

Asymmetric Cryptography

SIO

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Asymmetric Ciphers

- Use key pairs
 - **One private key**: personal, not transmittable
 - **One public key**: available to all
- Enable more interesting use cases
 - **Confidentiality** without any previous exchange of secrets
 - **Authentication**
 - Of contents (data integrity)
 - Of the data origin (source authentication, or digital signature)

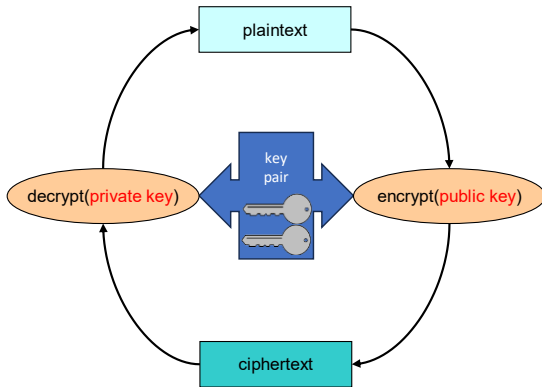
In symmetric cryptography, Alice and Bob must first share one secret, and getting that secret to each other without an eavesdropper is the whole game. For two people it is awkward; for a group of n people it is $n(n-1)/2$ different keys, and every new participant means re-keying everyone. Asymmetric ciphers remove the pre-shared secret entirely: each person generates a mathematically linked pair, a public key they can publish anywhere and a private key they keep. No secret ever has to cross the network first.

The pair works in two opposite directions, and both come from one fact: what one key does, only its partner can undo. In the confidentiality direction, Alice encrypts with Bob's public key and only Bob's private key decrypts, so the traffic may be fully public and still private. In the authenticity direction, Alice applies her private key to data, and anyone holding her public key can check the result; that is the digital signature. Feel it concretely: Bob publishes his public key in a directory, and a stranger who has never met Bob can still send him a message nobody else can read, while Bob can sign a document that any clerk with the published key can verify.

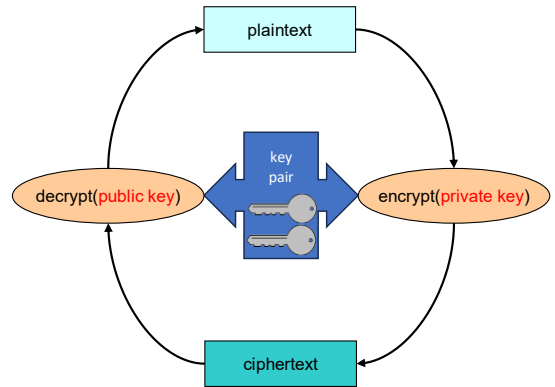
The classic student error is a direction mix-up: encrypting with your own public key does not make a message secret, because the matching private key can read it, and it proves nothing about who sent it either. The second trap is treating the private key like a password. It is not a credential you can reset; it is an identity. Whoever holds it can impersonate the owner, and the owner cannot credibly deny what was signed, so a leak is a compromised identity, and the remedy is re-issuing everything the key certified, not changing a password.

Operations of an asymmetric cipher

Confidentiality



Authenticity



Asymmetric ciphers come as two mirror-image operations on the same key pair, and keeping the directions straight is the single most important habit in this lecture. In the confidentiality direction the sender encrypts with the receiver's public key and only the receiver's private key decrypts: a secret moves from a public channel into a private one. In the authenticity direction the sender applies their own private key and anyone can check the result with the sender's public key: a private operation produces a public proof.

Walk the two loops. Confidentiality: Alice has Bob's public key, which is public by definition, so she can start immediately over a fully open channel, and the output is ciphertext that only Bob's private half undoes. Authenticity: Alice applies her private key to her message, and a verifier, a stranger holding no secret, applies Alice's public key and checks whether the original comes back. Notice the asymmetry of secrets: in the first case the receiver must keep the private key safe, in the second the sender. Neither side needs the other to hold any secret at all, which is why strangers can bootstrap a relationship with no meeting and no courier.

The most common failure is a direction mix-up, and it is a logic error. Typical confusions: believing that encrypting with your own public key hides the message (it does not; your own private key reads it), or that signing is just encryption in reverse with the same guarantees (it is not; both the message and the signature are public there, and the protected thing is the claim of origin). When in doubt, ask who holds the private key in that direction, because that person is the one whose secret is doing the work, and every confusion in this area resolves once you name that person.

Real protocols use both directions constantly, usually in combination: TLS authenticates the server with a signature and then negotiates a symmetric session key; PGP signs and encrypts in one message; code signing is pure authenticity. Takeaway: public key in, secret out, is confidentiality, and private key in, public proof out, is authenticity, and if you can name which direction each protocol step is using, the rest of the lecture reads itself.

Use cases: confidential communication

- Secure communication with a target (Bob)

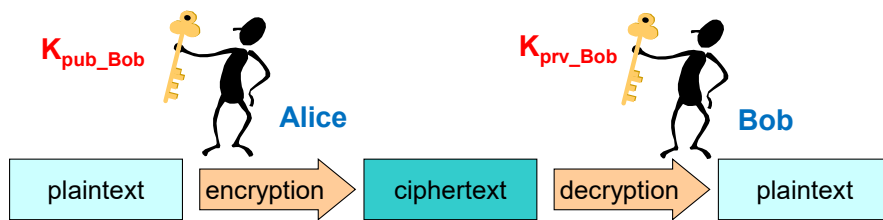
- Alice encrypts plaintext P with Bob's public key K_{pub_Bob}

$$\text{Alice: } C = \{P\}_{K_{pub_Bob}}$$

- Bob decrypts ciphertext C with his private key K_{priv_Bob}

$$\text{Bob: } P' = \{C\}_{K_{priv_Bob}}$$

- P' should be equal to P (requires checking using integrity control)
- K_{pub_Bob} needs to be known by Alice



This is the fundamental promise of public-key encryption: two strangers can talk privately over a completely public channel, with no meeting, no courier, no pre-shared secret. Alice simply looks up Bob's public key, which is allowed to be public, encrypts her message, and sends the ciphertext anywhere; Bob is the only person who can read it, because only he holds the matching private key. The security does not come from hiding the traffic, it comes from one-way math, and everything about symmetric key distribution is eliminated.

Concretely: Alice computes C from the plaintext P and Bob's public key, and sends C over the open internet; Bob applies his private key to C and expects to recover exactly P . Two real requirements hide behind that simple picture. First, Alice must be sure the key she calls Bob's really is Bob's, because if an attacker substitutes their own key, the attacker reads everything while Bob just gets garbage. Second, the recovered text must actually be checked against what was expected, because decryption with the wrong key, or a corrupted or tampered ciphertext, can return garbage that looks like plaintext, which is why real systems always pair encryption with an integrity control.

The key-identity problem is the big pitfall and the cipher cannot answer it: the math cannot tell Alice whether a public key belongs to Bob, and that gap is filled by certificates and trust structures, not by the algorithm. A second, more mechanical pitfall: raw asymmetric primitives encrypt only small blocks relative to the key size, so a long message must be split into chunks, and each chunk is exposed to the deterministic, malleable behavior of the raw operation. Students also forget that Bob must verify equality of the recovered and the expected text, not merely that the decryption ran, because a tampered input can produce output that is garbage but looks plausible.

This exact pattern, in hybrid form, is what you use every day: TLS protects the session key with the server's public key, PGP encrypts mail to a recipient's key, and S/MIME does the same inside signed mail. The diagram is the idealized pure version; the deployed version almost always wraps a short symmetric key. Takeaway: confidentiality between strangers reduces to one question, do I really have your public key, and the rest of the lecture is about answering it.

Use cases: authenticated communication

- Authenticate the communication from Alice

- Alice encrypts plaintext P with her private key $K_{\text{prv_Alice}}$

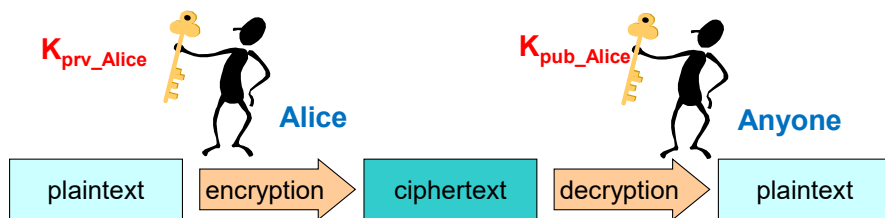
$$\text{Alice: } C = \{P\}_{K_{\text{prv_Alice}}}$$

- Anyone can decrypt ciphertext C with Alice's public key $K_{\text{pub_Alice}}$

$$\text{Anyone: } P' = \{C\}_{K_{\text{pub_Alice}}}$$

- If $P' = P$, then C is Alice's signature of P

- $K_{\text{pub_Alice}}$ needs to be known by the message verifiers



The authenticity direction answers a different question: how can Alice prove to the world, not just to one person, that a message really comes from her, in a way anyone can check at any time without contacting her? The answer is to apply her private key to the message. Because only she holds that key, anyone who can undo the operation with her public key has evidence the message came from her. Compare with the confidentiality case: there the message is secret and the key is the mechanism; here the message is public and the protected thing is the operation itself, the claim of origin.

Walk it: Alice computes C from the plaintext P using her private key, and publishes both P and C . Any verifier takes C , applies Alice's public key, and gets P' . If P' equals P , then C is accepted as Alice's signature on P , because only the holder of the private key could have produced a value that the public key undoes back to P . The verifier needs no secret, no session, no access to Alice; the check is a self-contained computation. That is why the same mechanism can verify a software update on a machine in another country, a contract in a filing cabinet, or a commit message in a repository, long after the signing happened.

The big caution: encrypting with the private key is a teaching model, and real signature schemes do not literally do that, because a raw private-key operation on arbitrary data has real flaws, including malleability and no tamper detection. Production schemes sign a hash of the data with purpose-built mathematics, but the properties are the same: integrity, origin, non-repudiation. The human pitfall is on the signing side: signing is a strong act, so never sign data you have not read and understood, and treat the private key as an identity. Whoever signs speaks for the owner of the key, and a careless signature is a statement you cannot take back.

Everywhere you see a claim of authorship with verifiable proof, this direction is underneath: signed software updates, signed commit messages, signed certificates, signed contracts, and the signature part of S/MIME mail. Takeaway: the private key is a stamp of identity that anyone can check but only the owner can apply, and non-repudiation, the author's inability to deny what they signed, is what turns a technical check into evidence.

Asymmetric ciphers

Issues

- Advantages

- They are a fundamental authentication mechanism
- They allow to explore features that are not possible with symmetric ciphers

- Disadvantages

- Performance: 2 or 3 orders of magnitude slower than AES
- Very inefficient and memory consuming: Large keys ($2^{2048} = 617$ digits)

- Problems

- Trustworthy distribution of public keys: how to know if the public key is the correct one?
- Lifetime of key pairs: How to make sure that we can deal with lost/deprecated/leaked keys?

Why do we pay the price of asymmetric cryptography? Because it gives two things symmetric keys cannot. First, authentication without a shared secret: a key shared between two people can only prove that one of them sent the message, while a signature proves which one. Second, non-repudiation: the sender cannot deny what they signed, because only they held the key that made it. These properties underpin certificates, signed updates, signed mail, and every trust structure built on public keys, and none of them is possible with a symmetric key alone.

The price, in the numbers given here, is performance: one asymmetric operation is two or three orders of magnitude slower than an AES block, and the keys are large, a 2048-bit modulus is 617 decimal digits. The design rule that follows: use asymmetric operations to set up and authenticate (wrap one short key, sign one short digest) and use symmetric operations to move data (encrypt the bulk). The hybrid pattern is not a workaround for a weakness; it is the correct division of labor, and it is why you will almost never see a long message encrypted directly with a public key.

The two listed problems are organizational, and they are the hard part. Problem one, key authenticity: how does Alice know that the public key she found really belongs to Bob and not an attacker? No algorithm answers that; the answer is a trust structure, certificates, authorities, revocation lists, or out-of-band fingerprint exchange. Problem two, key lifetime: what happens when a key is lost, leaked, or its owner leaves the organization? The mathematics does not age gracefully, so you need policies, rotation, revocation, re-issuance. Students underestimate these because they are not in the equations, but in practice this is where most public-key deployments actually fail.

These two problems are exactly what the public-key-infrastructure topic exists to solve: bind a key to an identity, and retire keys that must no longer be trusted. The same trade-off shows up in every deployment: HSMs protect private keys because the cost of a leak is an identity, and hybrid protocols exist because the cost of pure asymmetric throughput is unacceptable. Takeaway: asymmetric cryptography is a bargain, you give up speed and key-management simplicity in exchange for authentication and non-repudiation, and you should know the price before you spend it.

Asymmetric ciphers

Overview

- Approaches: complex mathematic problems
 - Discrete logarithms of large numbers
 - Integer factorization of large numbers
- Most common algorithms
 - RSA
 - ElGamal
 - Elliptic curves (ECC)
- Other techniques with asymmetric key pairs
 - Diffie-Hellman (key agreement)

Every practical asymmetric scheme stands on a deliberately chosen hard problem: easy to compute in one direction, infeasible to invert, with difficulty growing fast with size. Two classical problems dominate. Integer factorization: multiply two large primes, trivial; given the product, find the primes, infeasible for large inputs; RSA stands here. The discrete logarithm: pick a base g and a secret a , compute g to the a modulo q , trivial; given the result, recover a , infeasible for large q ; Diffie-Hellman, ElGamal, and DSA stand here. The shape is always the same: cheap forward, expensive backward.

Elliptic curves add a third family with the same one-way shape but on the points of a curve rather than in a modular ring, and the practical payoff is size: about 256 bits of curve gives roughly the security of a 3072-bit RSA modulus, so keys and handshakes are smaller and faster at equal security. Note where Diffie-Hellman sits: it is not an encryption algorithm at all, it is a key agreement protocol, two parties deriving a shared secret from public values without ever sending the secret. ElGamal shows one problem powering both an encryption scheme and a signature scheme: the hard problem is the currency, the schemes are just different ways of spending it.

Two confusions. One: treating factorization and the discrete logarithm as the same problem; they are different, and the attacks, the key sizes, and the quantum story differ by family. Two: thinking Diffie-Hellman encrypts; it does not, it agrees a key, and by itself it authenticates no one, which is why TLS pairs it with certificates. A mnemonic covers the whole landscape: factorization gives RSA; the discrete log gives Diffie-Hellman, ElGamal, DSA; the curve version gives ECC, ECDH, ECDSA. Once you can name which problem a scheme stands on, its security level, its key sizes, and its quantum vulnerability all follow from the problem, not the name.

This map is why the quantum discussion lands the way it does: Shor's algorithm attacks exactly these two problem families, so every scheme here, RSA, Diffie-Hellman, ECC, is on the migration path toward post-quantum designs, while hash-based and lattice schemes stand on different ground. Takeaway: when you meet a new asymmetric scheme, ask which hard problem it spends, because that answer determines its key sizes, its performance, its known attacks, and its future.

RSA

Rivest, Shamir, Adelman, 1978

- Keys: Private: (d, n) Public: (e, n)
- Public key encryption (confidentiality) of P
 - $C = P^e \bmod n$
 - $P = C^d \bmod n$
- Private key encryption (authenticity) of P
 - $C = P^d \bmod n$
 - $P = C^e \bmod n$

P, C are numbers!
Message is converted to/from numbers
 $0 \leq P, C < n$

RSA, from Rivest, Shamir, and Adleman in 1978, is the most deployed asymmetric scheme in the world, and it is strikingly simple, which is both its charm and its trap. The entire algorithm is one arithmetic operation in each direction: modular exponentiation. The public key is the pair (e, n) , the private key is (d, n) , and both share the modulus n . Encrypting means raising a message number to the power e , modulo n ; decrypting means raising the result to the power d , modulo n . No tables, no loops, no state; if you can do repeated squaring, you can do RSA.

Both directions follow from one identity: e and d are chosen so that raising to e and then to d , or the other way around, returns the original number, because e times d is congruent to 1 modulo $\phi(n)$. The confidentiality direction is $C = P$ to the $e \bmod n$, undone by $P = C$ to the $d \bmod n$; the authenticity direction swaps the roles, $C = P$ to the $d \bmod n$, undone by raising to e . The same exponent pair does two jobs. The catch, stressed in the side note, is that P and C must be numbers, with P between 0 and $n-1$: RSA never sees bytes or text directly, the message is first converted to an integer and the output back, and in practice that conversion is done by a padding scheme, never by a naive byte read.

Conceptually, treating raw RSA as a safe cipher is wrong: it is deterministic, so the same message always gives the same ciphertext and an attacker can build a dictionary of likely plaintexts, and the operation is malleable, a modified ciphertext is a modified plaintext in a predictable way. That is why every real deployment wraps raw RSA in a padding layer; textbook RSA is the engine and the padding is the safety system.

RSA appears in the places you already trust: server certificates in TLS, PGP and S/MIME signatures, SSH host keys, code-signing blobs, and HSMs that store the private exponent. Takeaway: RSA in one line is exponentiate with e to encrypt, exponentiate with d to sign, and everything else in the scheme, key generation, padding, key sizes, attacks, is engineering around that single arithmetic core.

- Computational complexity: **Discrete logarithm** and **Integer factoring**
- Key selection
 - Large **n** (hundreds or thousands of bits)
 - **$n = p \times q$** with **p** and **q** being large (secret) prime numbers
 - Choose an **e** co-prime with $(p-1) \times (q-1)$
 - Compute **d** such that **$e \times d \equiv 1 \pmod{(p-1) \times (q-1)}$**
 - Discard **p** and **q**
 - The value of **d** cannot be computed out of **e** and **n**
 - Only from **p** and **q**

coprime $\rightarrow \gcd(a, b) = 1$

$\times \rightarrow$ multiplication

mod $\rightarrow$ modulo operation

$\equiv \rightarrow$ modular congruence

$a \equiv b \pmod n$ iff $\text{rem}(a,n) = \text{rem}(b,n)$

All of RSA's security rests on one number: n , the product of two large primes p and q that are kept secret. The public key is (e, n) . If you know p and q , you can compute the private exponent d from the public key in seconds; if you know only e and n , you must factor n , and for a large product that is, as far as classical computing is concerned, infeasible. The entire security argument is that one direction is cheap and the other is not, and the key-generation procedure is built to make the cheap direction available only to its owner.

Walk the construction: pick two large primes p and q from a cryptographically secure random source, then primality-check them; compute $n = p$ times q and $\phi = (p-1)$ times $(q-1)$; choose e , co-prime with ϕ , in practice almost always 65537, which is prime, small, and makes encryption fast; compute d as the modular inverse of e modulo ϕ , the number with e times d congruent to 1 mod ϕ ; and then discard p and q . After discarding, the only things that exist are (e, n) and (d, n) , and the discarded factors are precisely what make the private key private: d is computable from e and n only if you also have the factors.

Three pitfalls live in this step. One, co-prime means the greatest common divisor of e and ϕ is 1; if they share a factor, no inverse d exists and the pair is unusable, so e is not a free choice. Two, the primes must come from a good random source; weak, small, or reused primes are a classic real-world failure, because factoring then becomes easy or the keys become guessable. Three, the security is a statement about what is known today: no efficient classical factoring algorithm is known, but a large quantum computer running Shor's algorithm would factor n directly, which is why key sizes keep growing and long-lived secrets are already being planned for migration.

Every RSA key you will ever see, in a certificate, an SSH host key, or an HSM, was made by exactly this recipe, and any audit of a deployment starts by asking how the primes were generated and whether they were kept secret. Takeaway: RSA's one-wayness is not a property of the exponentiation but of the discarded factors, so the security of the scheme is only as good as the secrecy of p and q after key generation.

Playing with RSA

- $p = 5$ $q = 11$ (prime numbers)
 - $n = p \times q = 55$
 - $(p-1) \times (q-1) = 40$
- $e = 3$ (public key = e, n)
 - Coprime of 40
- $d = 27$ (private key = d, n)
 - $e \times d \equiv 1 \pmod{40}$ $\rightarrow$ $d \times e \pmod{40} = 1$ $\rightarrow$ $(27 \times 3) \pmod{40} = 1$
- For a message to encrypt, $P = 26$ (notice that $P, C \in [0, n-1]$)
 - $C = P^e \pmod{n} = 26^3 \pmod{55} = 31$
 - $P = C^d \pmod{n} = 31^{27} \pmod{55} = 26$

A tiny worked case is the fastest way to stop RSA from feeling like magic, and it uses the numbers on this page: $p = 5$, $q = 11$, so $n = 55$ and $\phi = (5-1) \times (11-1) = 40$. Choose $e = 3$, which is co-prime with 40; its inverse modulo 40 is $d = 27$, because $3 \times 27 = 81$, which is congruent to 1 mod 40. Public key (3, 55), private key (27, 55). Every real RSA key is this exact procedure with numbers hundreds of digits long; nothing new is added at scale.

Now run a message through. Take $P = 26$, which lies in the allowed range 0 to $n-1$. Encrypt: $C = 26^3 \pmod{55} = 17576 \pmod{55} = 31$. Decrypt: 31 to the 27th power mod 55 = 26. The round trip works, and so does the reverse direction, raise to d first then to e , which is the signature direction. Notice the scale for a moment: with two-digit primes anyone can factor 55 by hand, so this toy is a teaching case, not a security claim. The security appears only when p and q are hundreds or thousands of bits, at which point the same two lines of arithmetic become infeasible to invert.

Two things to take from the case. First, the range restriction is real: RSA only accepts message numbers smaller than the modulus, so long messages are chunked, and every chunk needs the padding layer to be safe; a student who tries to just split the bytes has reinvented an insecure variant. Second, determinism is visible in the toy: encrypt 26 again and you get 31 again, every time. That is the property that lets an attacker precompute a dictionary of likely messages and recognize them in the traffic, and it is why no real system uses raw RSA on data without randomization. The toy shows the engine; the next pages show the safety system.

If you can reproduce this case by hand, you can read any RSA documentation without fear: the padding, the key sizes, the attacks, all of it is engineering around three steps, pick two primes, compute an exponent pair, exponentiate modulo n . Takeaway: RSA is not a black box, it is arithmetic you can do with a pencil, and the whole discipline is deciding which layer, padding, key generation, or size, actually carries the security.

Hybrid Encryption

- Combines symmetric with asymmetric cryptography
 - Use the best of both worlds, while avoiding problems
 - Asymmetric cipher: Uses public keys (but it is slow)
 - Symmetric cipher: Fast (but with weak key exchange methods)
- Method:
 - Obtain K_{pub} from the receiver
 - Generate a random K_{sym}
 - Calculate $C1 = E_{sym}(K_{sym}, P)$
 - Calculate $C2 = E_{asym}(K_{pub}, K_{sym})$
 - Send **C1 + C2**
 - C1 = Text encrypted with symmetric key
 - C2 = Symmetric key encrypted with the receiver public key
 - May also contain the IV

Hybrid encryption is how the two halves of the course finally cooperate, and it is the pattern behind almost every system you use. The public-key half solves key distribution: anyone can send a key to the right person over a public channel. The symmetric half solves throughput: it encrypts the actual data fast. Neither is sufficient alone, asymmetric is slow and size-limited, symmetric is fast but has no convenient way to share its key, so the hybrid uses each exactly where it is strong. This is not a compromise; it is the design.

The protocol, step by step: obtain the receiver's public key K_{pub} ; generate a fresh random symmetric key K_{sym} ; compute $C1 = \text{encrypt}(K_{sym}, P)$, the message under the fast cipher, often together with an IV; compute $C2 = \text{encrypt}(K_{pub}, K_{sym})$, the short key wrapped under the public key; send C1 plus C2. Only the receiver can unwrap K_{sym} with the private key, and only K_{sym} opens C1. An eavesdropper who captures both pieces holds two locked boxes with no key: the public-key box opens only with a secret they do not have, and without the symmetric key the bulk is unreadable noise.

The subtlety students miss is why K_{sym} must be fresh for each message. Reusing one symmetric key for many messages under one public key leaks relationships between the messages, repeated patterns, identical keystream positions, and it destroys forward secrecy: one later compromise of the public-key material exposes everything wrapped under that reused key. A fresh key per message also bounds the blast radius: if one symmetric key is ever exposed, only the data under it is exposed. The random K_{sym} is the most security-relevant value in the whole protocol, and its quality comes from a true random source, not a counter.

Randomization of asymmetric encryptions

- RSA is a deterministic algorithm: same message result in same output
 - Note: we do not have block cipher modes like in symmetric ciphers. Therefore, no feedback mechanisms
- What we need: Non-deterministic result of asymmetric encryptions
 - **N** encryptions of the same value, with the same key, should yield N different results
 - **Goal: prevent the trial & error discovery of encrypted values**
- Approaches
 - Concatenation of value to encrypt with two values
 - A fixed one (for integrity control)
 - A random one (for randomization)

Raw RSA has an embarrassing property: it is deterministic. The same plaintext, the same key, always the same ciphertext. An attacker does not need to break the math; they just need to guess what you are likely to send. Common business phrases, predictable headers, repeated form values: compute their ciphertexts once, build a dictionary, recognize them in the traffic whenever they appear. Symmetric ciphers solved this long ago with modes and nonces; asymmetric has no such built-in feedback, so it needs an explicit randomization layer.

The goal: N encryptions of the same value under the same key must yield N different ciphertexts, all still decrypting to the original. The classical trick mixes fresh randomness into the input before the raw operation: concatenate the plaintext with two values, a fixed one that supports integrity checking, and a random one that is the actual source of variation. Because the input now differs every time, the output differs every time, while the decryption procedure still knows how to peel the structure off and recover the message. The random value must be generated fresh per encryption and must be recoverable or checkable at decryption, which constrains how it is placed.

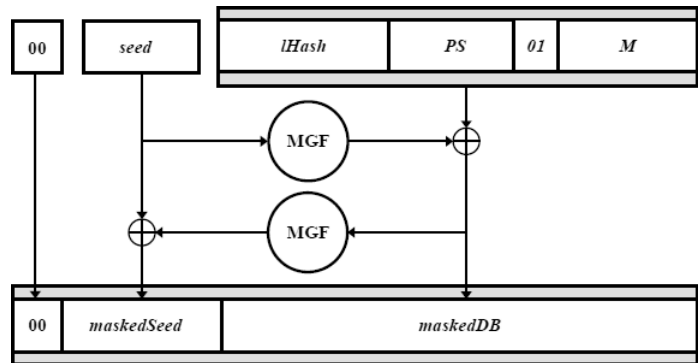
Two consequences people frequently underestimate. First, the random source is load-bearing: a weak or predictable generator quietly degrades the whole scheme no matter how strong the padding is, because randomized with a guessable seed is deterministic in disguise. Second, randomization alone protects confidentiality, not tamper detection: a good padding must also verify its own structure on decryption, because if an attacker can flip ciphertext bits and learn from whether decryption succeeds, they can recover secrets, the padding-oracle family of attacks that has broken real deployments. The standard answer that handles both needs at once is the scheme on the next page.

This is why every deployed RSA encryption label includes a padding name, and why raw RSA appears in protocol catalogs only as a warning. The same idea, randomness plus structure, reappears in signatures, where a fresh random nonce is part of the signed value, and in transport protocols, where fresh nonces and IVs guard the symmetric layer. Takeaway: a secure encryption must be unpredictable, and since the raw operation is not, the unpredictability has to be engineered in, and the quality of that engineering is part of the security.

Randomization of asymmetric encryptions

OAEP (Optimal Asymmetric Encryption Padding)

- iHash: digest over Label
- seed: random value
- PS: zeros
- M: plaintext
- MGF: Mask Generation Function
 - Similar to Hash, but with variable size



OAEP, Optimal Asymmetric Encryption Padding, is the standard answer to the previous problem: it makes RSA encryption non-deterministic and tamper-detecting at the same time, with a proof of why. The idea is to build a padded block of exactly the size RSA needs out of the message, a fresh random seed, a run of zero padding, and a hash of an optional context label, then bind all the parts together with a mask generation function so that no part can be moved without the others breaking. Decryption reverses the construction and checks the structure; if anything is off, the ciphertext was tampered with and the result is rejected, not a misleading plaintext.

Read the parts as a pipeline: M is the plaintext, seed is the random value that makes each encryption different, PS is the zero padding that fills the block, iHash is a digest of the label that binds this encryption to its context, so a ciphertext cannot be moved between two uses of the same key in different settings, and the MGF is a mask generator, a hash-like function that stretches a short value into a long mask. The MGF is what entangles the seed with the message: flip a bit anywhere in the block and the recovered padding will not check out, which is the tamper detection working.

Two points to hold. First, OAEP's security is proven relative to standard assumptions about the underlying hash and the raw RSA operation; that is a much stronger claim than it has not been broken yet, though it is still a statement about assumptions, not an absolute guarantee. The older ad-hoc padding was replaced precisely because it has known weaknesses in practice, including padding-oracle exposure. Second, the choice of the inner hash matters: a weak or deprecated hash degrades the whole padding, so the common guidance is a strong, current hash in the SHA-256 class, and the scheme name should name that hash so the deployment is auditable.

OAEP is what RSA encryption means in TLS, PGP, and S/MIME today, and the scheme identifier you see in a certificate, RSA with OAEP and a named hash, is the visible signature of this engineering. The same lesson applies to the signing side: never bare RSA, always a named padding or encoding. Takeaway: the padding layer is not decoration around RSA, it is the part that makes RSA a secure cipher, and if a protocol name omits it, that is a red flag.

Diffie-Hellman Key Agreement (1976)



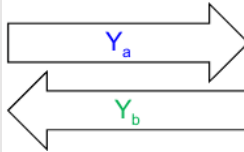
q (large prime)
 α (primitive root mod q)



$a = \text{random}$

$$Y_a = \alpha^a \bmod q$$

$$K_{ab} = Y_b^a \bmod q$$



$$K_{ab} = K_{ba}$$

$b = \text{random}$

$$Y_b = \alpha^b \bmod q$$

$$K_{ba} = Y_a^b \bmod q$$

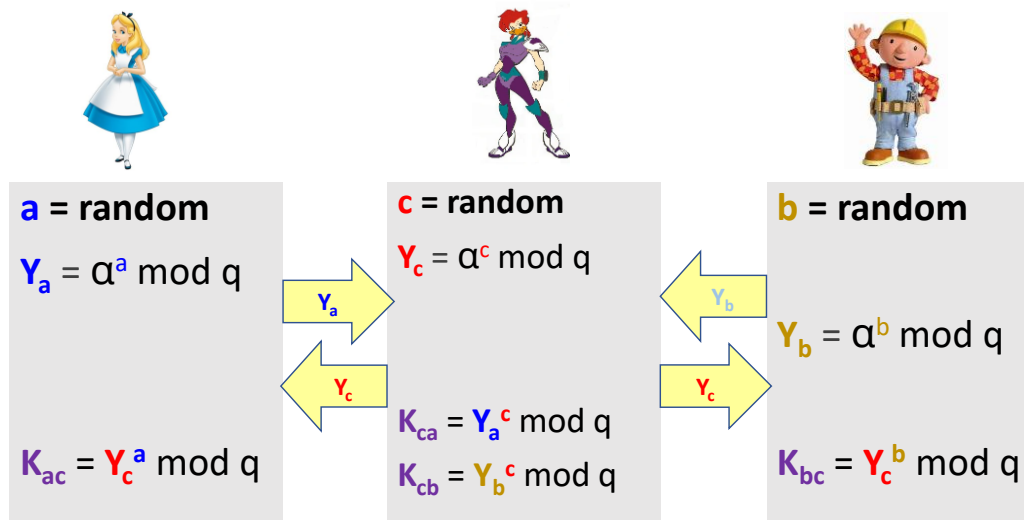
Diffie-Hellman, from 1976, answered a question that looked impossible before: how can two people agree on a shared secret over a channel they do not trust, without ever transmitting that secret, and without having shared anything before? The trick is to use public-key mathematics not to encrypt data but to derive a key. Each party picks a private random exponent and publishes a value computed from it; both parties then combine the two public values with their own private exponent, and arithmetic guarantees they land on the same secret, which never crossed the wire.

The small-number version makes it click. Agree on a large prime q and a base g . Alice picks a random a , publishes $Y_a = g$ to the a , mod q . Bob picks a random b , publishes $Y_b = g$ to the b , mod q . Alice computes $K_{ab} = Y_b$ to the a , mod q , which is g to the a times b . Bob computes $K_{ba} = Y_a$ to the b , mod q , which is g to the b times a , the same value. The eavesdropper sees q , g , Y_a , and Y_b , but to get the secret they would have to recover a from Y_a , the discrete logarithm, which is infeasible for large q . Two strangers, one secret, no prior exchange, no trusted channel.

The classic pitfall, the one to drill: the basic protocol authenticates no one. An active attacker in the middle runs two separate exchanges, one with Alice and one with Bob, and relays the public values between them. Both parties believe they share one key; the attacker holds two, and the channel is fully readable and writable for them. That is why every real protocol, TLS above all, pairs key agreement with authentication, using certificates or a pre-shared secret to bind the public values to identities. A second, subtler point: when the per-session exponents are fresh and discarded, a later compromise of long-term keys does not expose past sessions, which is forward secrecy.

Diffie-Hellman is the skeleton of modern transport security: the key exchange in TLS, in SSH, and in IPsec all agree keys this way, and the ephemeral curve version, ECDH, is the default in current deployments. Takeaway: Diffie-Hellman is key agreement, not encryption, and by itself it gives secrecy against passive eavesdropping but zero protection against an active middleman until you add authentication.

Diffie-Hellman Key Agreement (1976)



This generalizes the two-party handshake to three participants, and the point is that the mathematics composes. Each of a , b , and c picks a secret exponent and publishes the corresponding power of the base; then any two of them can compute a shared key exactly the way the two-party case does: combine your own secret with the other party's public value. So a and c derive one key, b and c another, and a and b a third, while the outsiders only ever see public values.

Concretely: c publishes $Y_c = g$ to the c , $\bmod q$. To get the key between a and c , Alice computes Y_c to the a , $\bmod q$, and c computes Y_a to the c , $\bmod q$; both are g to the a times c . For the key between b and c , Bob and c do the same with b . The key property is that no pair's key is derivable from any other pair's: the group structure lets each subset form its own secret while everyone else holds only public values. That is the seed of group key agreement: instead of one shared key for everyone, each pair or subgroup has its own, and adding a participant does not force everyone else to renegotiate from scratch.

The caveats carry over, amplified. First, authentication is still absent: an active attacker can still interfere with any of the exchanges, so the multi-party version has the same middleman exposure as the two-party one, and real group protocols add signatures or certificates to the public values. Second, the combinatorics grow: with n participants there are $n(n-1)/2$ possible pairwise keys, which is exactly the quadratic key-management cost that killed naive symmetric group design. Practical protocols tame it with structure, hierarchies, subgroups, key servers, or tree-based constructions, so the raw composition is the mathematical core but not the whole engineering story.

The idea reappears wherever a group needs per-pair or per-subgroup secrecy: group messaging, some broadcast encryption schemes, and the group stages of larger key-distribution protocols all build on the fact that any subset can derive its own secret from public values. Takeaway: Diffie-Hellman is not a one-shot handshake between two people, it is a composable primitive, and the moment you see a group protocol, look for this per-pair derivation underneath.

Elliptic curves: the big picture

An elliptic curve is a set of points (x, y) that satisfy an equation like

$$y^2 = x^3 + ax + b$$

- **Points form a group:** adding two points on the curve geometrically gives a third point
- Multiplying the generator G by an integer a ($a \cdot G$) is easy to compute
- **Finding a from $a \cdot G$ is infeasible:** the Elliptic Curve Discrete Logarithm Problem (ECDLP)
 - 256-bit ECC $\approx$ 128-bit security $\approx$ 3072-bit RSA: far smaller keys

SIO

Elliptic curves give public-key cryptography a smaller, faster group to work in. An elliptic curve is, loosely, the set of points (x, y) satisfying an equation like $y^2 = x^3 + ax + b$, and the useful part is not the drawing but the algebra: there is a geometric rule for adding two points on the curve, and the result is a third point on the curve. Because the rule is defined entirely on the coordinates, it is a precise computation, not a sketch, and the points, plus a special point at infinity that acts as zero, form a mathematical group.

The group operation is the whole story. To add two points, draw the line through them, find where the line meets the curve a third time, and reflect that point across the x-axis; the reflection is the sum. To add a point to itself, use the tangent line instead. A generator G is a fixed starting point, and multiplying G by an integer a , written a times G , means adding G to itself a times. Computing a times G is fast even for huge a , because repeated doubling gives the result in logarithmically many additions. The security comes from the other direction: given a times G and G , recovering a is infeasible for large, well-chosen curves. That one-wayness, the elliptic-curve discrete logarithm problem, is the curve-world twin of the discrete log from the Diffie-Hellman pages.

The practical payoff is size, and it is the reason ECC is everywhere: roughly 256 bits of curve matches the security of about 3072 bits of RSA modulus, so public keys, signatures, and handshakes are much smaller and faster at equal security, which matters on constrained devices and in latency-sensitive handshakes. The pitfall is curve choice: an elliptic curve is a family, not a product, and only specific, publicly vetted curves with clean parameters should be used. Inventing a curve, or using an obscure one with unreviewed constants, is how implementations get hidden weaknesses; the rule is the same as for hashes, pick from the vetted list, never roll your own.

ECC appears in transport-security handshakes, in hardware security modules that store a small curve scalar instead of a huge RSA exponent, in code signing, and in cryptocurrency wallets. Takeaway: the elliptic-curve trick is fast group operation forward, infeasible discrete log backward, and the whole family of ECC schemes is that one sentence applied to encryption, key agreement, and signatures.

Elliptic Curve Cryptography (ECC)

- Elliptic curves are specific functions
 - They have a generator (G)
 - A private key K_{prv} is an integer with a maximum of bits allowed by the curve
 - A public key K_{pub} is a point $(x, y) = K_{\text{prv}} \times G$
 - Given K_{pub} , it should be hard to guess K_{prv}

- Curves

- NIST curves (15)
 - P-192, P-224, P-256, P-384, P-521
 - B-163, B-233, B-283, B-409, B-571
 - K-163, K-233, K-283, K-409, K-571

Other curves

- Curve25519 (256 bits)
- Curve448 (448 bits)

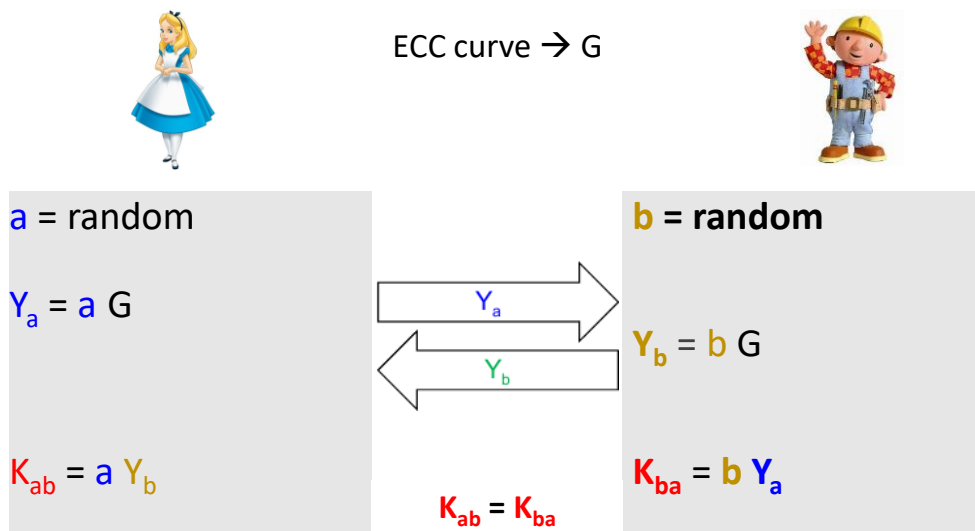
ECC applies the curve group to the key-pair idea in its most direct form, and the construction is almost embarrassingly small. The private key K_{prv} is just a random integer, bounded by the curve's order; the public key is the point $K_{\text{pub}} = K_{\text{prv}} \times G$, the generator scaled by that integer. Computing the public key from the private one is one fast scalar multiplication; going the other way, finding K_{prv} from K_{pub} , is the elliptic-curve discrete logarithm problem, and it is exactly that one-wayness that makes the pair usable as a key pair.

Feel the scale with numbers: on a 256-bit curve the private key is 32 random bytes, and the public key is a pair of 32-byte coordinates, so the whole key pair fits in about 64 bytes, compared with roughly 384 bytes for a 3072-bit RSA public key at similar security. A scalar multiplication that would be enormous in a big integer modulus is just a few hundred point operations, which is why handshakes and signatures on curves are faster and smaller. The curve list is the practical point: use the standardized, publicly vetted families, the P curves over prime fields such as P-256, and the K and B curves over binary fields, plus the modern heavily audited Curve25519 and Curve448, which are now defaults in many protocols and libraries.

The pitfall is choosing a curve, and it has two faces. First, size: 192-bit and 163-bit curves are considered weak by current guidance, sized for an era when 1024-bit RSA was acceptable, so new deployments should start at 256 bits. Second, pedigree: vetted means the parameters have been reviewed by many independent groups for hidden structure, and binary-field curves in particular have fewer implementations and slower performance than the prime-field and Curve25519 families, which is why newer protocols quietly dropped them from their default suites. In asymmetric cryptography the parameter choice is part of the security decision, not an implementation detail.

The curve is doing quiet work everywhere: TLS handshakes, SSH, code signing with ECDSA, and hardware security modules that store a small scalar. Takeaway: an ECC key pair is a random integer and its multiple of G , and the whole discipline is choosing a vetted curve at an adequate size, because the math is the same for everyone, but the parameters are where the security actually lives.

ECDH: DH with ECC



ECDH is Diffie-Hellman with the finite field replaced by the curve group, and the protocol shape does not change at all. The point of doing it is efficiency at equal security: smaller keys, faster operations, shorter handshakes, and a noticeable win on devices with limited CPU or battery. The security argument is the same one-wayness, now the curve discrete log problem, and the vulnerability profile is the same too: key agreement without authentication, and forward secrecy when the exponents are ephemeral.

The exchange: agree on the curve and its generator G . Alice picks a random a and publishes $Y_a = a$ times G . Bob picks a random b and publishes $Y_b = b$ times G . Alice computes $K_{ab} = a$ times Y_b , which is a times b times G ; Bob computes $K_{ba} = b$ times Y_a , which is b times a times G , the same point, because scalar multiplication on the curve is commutative. Neither a nor b ever crosses the channel, and an eavesdropper holding G , Y_a , and Y_b faces the curve discrete log to recover either exponent. Compare the sizes: two 32-byte public values carry a shared secret with roughly 128-bit security, where the finite-field version needs multi-hundred-byte values.

The pitfalls are the Diffie-Hellman pitfalls, in curve clothing. First, no authentication: a middleman can still run two separate ECDH exchanges and read and write the whole session, so real protocols authenticate the public points with certificates or signatures, and TLS does exactly this. Second, ephemeral or not: if the curve keys are fresh per connection and discarded, past sessions survive a later long-term key compromise, forward secrecy; if a long-term key is reused, they do not, and the deployed form, usually written ECDHE, is the ephemeral one. Third, a small practical note: point validation matters, because a malformed public point from a peer can be an implementation hazard, which is another reason to use a vetted library.

ECDH is the default key exchange in modern TLS and SSH, it appears in pairing and authentication flows, and hardware security chips store the small curve scalar instead of a huge RSA exponent. Takeaway: ECDH is Diffie-Hellman with a smaller, faster group, and when you read a modern handshake, the curve is almost certainly doing the heavy lifting in exactly this shape.

ECC public key encryption

Combines hybrid encryption with ECDH

- Obtain $K_{\text{pub_recv}}$ from the receiver
- Generate a random $K_{\text{prv_send}}$ and the corresponding $K_{\text{pub_send}}$
- Calculate $K_{\text{sym}} = K_{\text{prv_send}} K_{\text{pub_recv}}$

$$C = E(P, K_{\text{sym}})$$

- Send $C + K_{\text{pub_send}}$
- Receiver calculates $K_{\text{sym}} = K_{\text{pub_send}} K_{\text{prv_recv}}$

$$P = D(C, K_{\text{sym}})$$

Unlike RSA, the basic curve operation does not hand you encrypt-with-public, decrypt-with-private directly, so ECC encryption borrows the hybrid pattern from earlier and wraps ECDH inside it. The scheme shown here, in the ECIES family, is the standard answer: use the curve for key agreement, use a fast symmetric cipher for the data, and get forward secrecy for free because the agreement key is fresh per message. The curve is used only for the job it is good at.

Walk it. The sender obtains the receiver's public key $K_{\text{pub_recv}}$. The sender then generates a fresh ephemeral key pair, a random $K_{\text{prv_send}}$ and the matching $K_{\text{pub_send}}$. Then K_{sym} is derived as the shared point between $K_{\text{prv_send}}$ and $K_{\text{pub_recv}}$, the sender's ephemeral secret times the receiver's public point, and the message is encrypted with the fast symmetric cipher: $C = \text{encrypt}(P, K_{\text{sym}})$. The sender transmits C plus $K_{\text{pub_send}}$. The receiver derives the same K_{sym} as $K_{\text{pub_send}}$ times their own $K_{\text{prv_recv}}$, and decrypts. Both sides computed the same shared point, and the ephemeral secret is thrown away after the message.

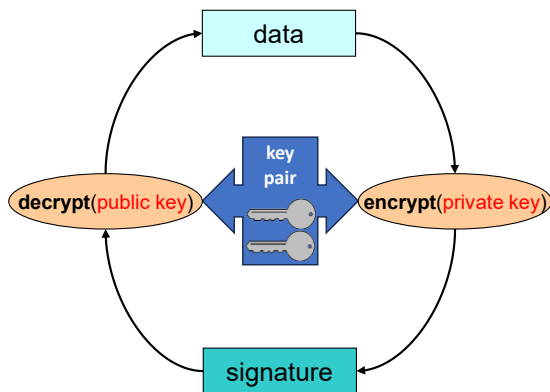
The trade-offs are honest and worth naming. The transmission grows by the size of the ephemeral public key, and the protocol has more steps than plain RSA encryption, which is why RSA is still used where simplicity and small envelopes matter more than forward secrecy. The deeper pitfall is conceptual: because a fresh ephemeral key is used per message, a later compromise of the receiver's long-term key does not expose past messages, forward secrecy, a property that plain RSA encryption to a long-term key does not give you. The two schemes are functionally interchangeable in most applications, and the choice is a design decision, not a correctness one.

ECIES-style encryption is what PGP and S/MIME use when the recipient is a curve key, and it is the same key-agreement-plus-symmetric-cipher theme as TLS. The unifying pattern across the whole lecture: RSA with OAEP, ECIES, and TLS are all variations on one design, a public-key step that protects a short symmetric key, and a symmetric step that moves the data. Takeaway: in modern practice, asymmetric encryption almost always means this hybrid, and once you see the pattern, the specific curve or modulus is a detail.

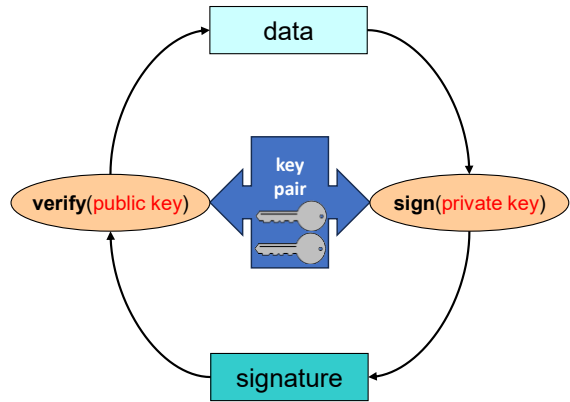
Digital signatures

Different algorithms, different approaches

Encrypt/Decrypt (RSA)



Sign/Verify (ElGamal, EC)



Digital signatures split into two structural families, and they are best learned side by side. The first family, the RSA style, treats signing as the private-key operation and verification as the public-key operation: the signature is literally encrypted data, and checking it is decrypting it and comparing. The second family, the ElGamal-based and elliptic-curve schemes, uses purpose-built sign and verify algorithms that never use encryption at all: the sign algorithm produces a signature mathematically tied to the data and the key, and a separate verify algorithm checks it.

The two mental models. Left: data goes in with the private key, a signature comes out; the signature plus data go in with the public key, and if the recovered data matches the original, the signature is valid. Right: data plus private key go into the sign function, a signature comes out; data plus signature plus public key go into the verify function, which returns true or false. Note what both share: the signature is computed from the data and the private key, verification needs the data, the signature, and the public key, and the verifier holds no secret. That last point is what makes verification scalable to millions of independent checkers.

The practical difference to care about: the sign-verify family tends to give shorter signatures, is easier to analyze, and is easier to implement correctly, which is why it dominates modern deployments, ECDSA and EdDSA. The encrypt-decrypt family is conceptually simpler, maps neatly onto the key-pair picture, and remains everywhere RSA is already deployed, in certificates and legacy tooling. The common mistake is to think signing is just encryption backwards in the sign-verify family; it is not, the signature is not a decryption of the data, and treating it as one leads to wrong conclusions about what is protected. When you read a scheme name, ask which family it is, because it tells you how verification works and what to expect of size and speed.

Both families appear in the systems you use: RSA signatures in X.509 certificates and TLS, ECDSA in code signing and mobile platforms, EdDSA in SSH and git commits. Takeaway: a signature is a claim of origin that anyone can check, and whether the machinery underneath is an inverse encryption or a dedicated verify function is a family detail, the properties, integrity, origin, non-repudiation, are the same.

Digital signatures

- Authenticate the contents of a document
 - Ensure its integrity (it was not changed)
- Authenticate its author
 - Ensure the identity of the creator/originator
- Prevent repudiation of signatures
 - Non-repudiation property
 - Genuine authors cannot deny authorship
 - Only the identified author could have generated a given signature

This is the property checklist for what a digital signature must deliver, and a careful analysis of any signed artifact should name all three and check each independently. Integrity of content: any change to the signed data invalidates the signature. Origin: the signature ties the content to a specific key, and through the key to a specific person. Non-repudiation: the originator cannot deny authorship, because only they could have produced the signature. The signature is a bundle of three guarantees, not one, and it verified is not the same as it verified for all three properties.

The mechanism behind non-repudiation is the private-key monopoly: the signature can only be produced by whoever holds the private key, so a valid signature is public evidence of that person's act. The other side of the coin is the verifier's position: they need no secret, no session, no access to the signer, just the public key and the data, which is what makes verification scalable and asynchronous. Millions of independent checkers can each run the same tiny computation, at any time, without ever contacting the author or any central authority. That asymmetry, one secret to make, none to check, is the structural reason signatures work as evidence.

The subtlety: a signature authenticates exactly the bytes it was computed over. A single-bit change in the document, or in the signing metadata, which algorithm, which key identity, which context label, breaks verification, which is why signatures are computed over the exact final bytes including the headers and context the scheme defines. The failure modes map one-to-one onto the three properties: tampering attacks integrity, forgery attacks origin, and denial attacks repudiation. When a signature fails to verify, the productive first question is not why it is broken but what exactly was covered, because a signature that verified over a subset of the document is not a signature over the whole thing.

The same checklist applies whether the artifact is a certificate, a signed update, a legal document, or a code commit, and it applies to both algorithm families, the encrypt-decrypt and the sign-verify. Takeaway: treat a signature as three separate claims, the data is unchanged, it is from this key, and that key cannot disavow it, and verify each one, because a system that checks only the first is checking a checksum with extra steps.

Practical Considerations

- Encryption with private key is vital for authentication
 - Only the author can make it, everyone can verify it
- But... sending secure authenticated texts will require two (slow) encryptions
 - Remember: Asymmetric ciphers are slow and inefficient
- Preferred Approach: **Encrypt(Hash(T))**
 - This is the base concept of a Digital Signature

The engineering tension: to send a message that is both confidential and authenticated with raw private-key operations, you would do two slow asymmetric operations on the full data, one for the signature, one for encryption, and asymmetric operations are orders of magnitude slower than symmetric ones. For a large file that is not just wasteful, it is impractical. The question is how to keep all the properties, integrity, origin, non-repudiation, without paying the asymmetric cost on the whole document.

The answer, and the base concept of a digital signature: sign the digest of the content instead of the content itself. A hash compresses the document to a fixed short value, so the asymmetric operation happens on that short value only, and its cost is now constant, independent of message size. Verification is symmetric: recompute the hash over the received document, and check the signature against that digest. If the document changed by one bit, the recomputed digest differs, and the check fails. The three properties survive intact, because a signature over the digest is a signature over the document in the only way that matters: any change to the document changes the digest, and the signature no longer matches.

The pitfall that makes the hash choice load-bearing: a collision in the hash, two different documents producing the same digest, lets an attacker substitute one document for another while keeping the signature valid. This is not hypothetical; it is exactly how real-world attacks on older weak hashes have played out, and it is why signature schemes name their digest in the algorithm identifier, so the signature's strength is auditable from the name alone. The related mistake: thinking the digest is there only for performance. It is a performance optimization, yes, but it is also the security boundary of the signature; the digest is what is signed, what is checked, and the glue between a big document and a small signature.

Every real signature you will meet, RSA with PSS, ECDSA, EdDSA, is hash-then-sign with a named digest, and the same pattern unifies the course: the hash function that gave you integrity in the symmetric part is now a load-bearing component of asymmetric security. Takeaway: a digital signature is a signature over a digest, and the strength of the whole construction is the strength of the weakest of the three, the hash, the key, and the padding or encoding that binds them.

Digital Signatures

- Approaches

- Digest function of the Text (only for performance)
- Asymmetric encryption/decryption or signature/verification

Signing:

$$A_x(\text{doc}) = \text{info} + E(K_x^{-1}, \text{digest}(\text{doc} + \text{info}))$$

$$A_x(\text{doc}) = \text{info} + S(K_x^{-1}, \text{digest}(\text{doc} + \text{info}))$$

info = signing context, signer identity, K_x

Verification:

$$D(K_x, A_x(\text{doc})) \equiv \text{digest}(\text{doc} + \text{info})$$

$$V(K_x, A_x(\text{doc}), \text{doc}, \text{info}) \rightarrow \text{True} / \text{False}$$

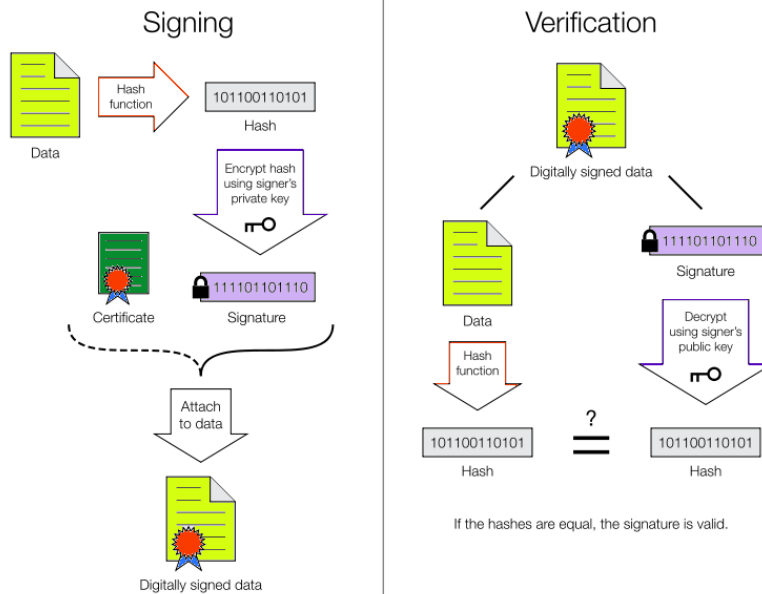
The formulas put hash-then-sign in compact form, separating the signing side from the verifying side. Signing takes the document, the signer's private key, and context information, and produces a signature $A_x(\text{doc})$. Verification takes the signature, the document, and the public key K_x , and returns a boolean, true or false: no partial results, no ambiguity. The boolean outcome is a design feature: a verifier must never have to interpret mostly valid.

The context term, info, carries which key was used, the signer's identity, sometimes a policy or timestamp; it is bound into what gets digested and signed. That binding is not decoration; it is what lets a verifier check that the signature was made with the expected key in the expected setting, not with some other key the person holds. Two signatures by the same person over the same document can differ in info and carry different meanings; an attacker who strips or swaps the context has changed what the signature claims. The two lines show the two families again: the encrypt-decrypt style applies the inverse key to the digest of document plus context; the sign-verify style applies the sign function to the same digest. Both sign digest plus context, both are checked against the document and the public key.

The practical trap: if the context is not bound into the signed digest, the signature is vulnerable to context attacks, the same signature value accepted in a different setting, with a different key identity, or over a different document framing. This is why the digest covers document plus context, not the document alone. The second trap is on the verification side: the verify function must be a strict boolean over all its inputs, signature, document, key, context; a check that silently ignores a field is a signature that does not cover what you think it covers. When you read a signature format, the first thing to locate is exactly which bytes feed the digest.

Every signature format you will meet, a certificate, a signed mail part, a code-signing blob, is an instance of these two lines with a specific binding of context. The same structure appears in transport security, where the handshake transcript is signed so the context cannot be swapped, and in git commits, where the commit object itself is what is hashed. Takeaway: a signature is a tuple, data, bound context, signature value, and if you can point to all three parts in any format, you can reason about what is protected and what a failure means.

Encryption / decryption signatures



This slide shows the encrypt-decrypt signature flow visually: the sender applies the private-key operation to the hashed data to produce the signature, and the verifier applies the public-key operation and compares the result against the original data. The diagram is the RSA-style family, where signing and verifying are literally inverse operations of each other.

What the picture reinforces: the signature and the data travel together, and verification needs both. The public-key operation is the inverse of the private-key operation - that is what makes the comparison work, and it is the entire mechanism in the RSA-style family. There is no shared state, no challenge, no second round trip: one operation on each side, and a comparison.

If you can explain this picture out loud - 'private key makes it, public key checks it, the check compares against the data' - you understand the core of how every certificate and signed update works, regardless of the specific algorithm underneath. The companion slide shows the same loop from the verifier's side; together they close the circle.

Digital Signature on a mail message

Multipart content, signature w/ certificate

```
From - Fri Oct 02 15:37:14 2009
[...
Date: Fri, 02 Oct 2009 15:35:55 +0100
From: User A <usera@domain.com>
MIME-Version: 1.0
To: User B <userb@domain.com>
Subject: Teste
Content-Type: multipart/signed; protocol="application/x-pkcs7-signature"; micalg=sha1; boundary="-----ms050405070101010502050101"
```

This is a cryptographically signed message in MIME format.

```
-----ms050405070101010502050101
Content-Type: multipart/mixed;
boundary="-----060802050708070409030504"
```

```
This is a multi-part message in MIME format.
-----060802050708070409030504
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: quoted-printable
```

Corpo do mail

```
-----060802050708070409030504--
-----ms050405070101010502050101
Content-Type: application/x-pkcs7-signature; name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7s"
Content-Description: S/MIME Cryptographic Signature
```

```
MIAGCSqGS1b3DQEHAQCAMIACAQExCzAJBgUrDgMCQgUAMIAGCSqGS1b3DQEHAQAoIIamTCCBUkwggSyoAMCAQICBACnIaEwDQVJYKoZIHvcNAQEFBQAwdTELMakGA1UEBhMCVVMxGDAWBgNV
[...
KoZIHvcNAQEFBQAEGYCoFks852BV7NWuww53vSx01XtI2JhC1CDlu+tcTPoMD1wq5dc5v40Tgsaw0N8dqVLk8aC/CdGMbRBu+J1LKrcVZa+khnjtB66HhDRLrjmEGDNttrEjbbqvpd2Q02
vx831PTIU+vCGxo47e6GyRydqTpbq0h49Zqmx+I36Z7iigAAAAAAA==
-----ms050405070101010502050101--
```

João Paulo Barraca, André Zúquete

SIO

27

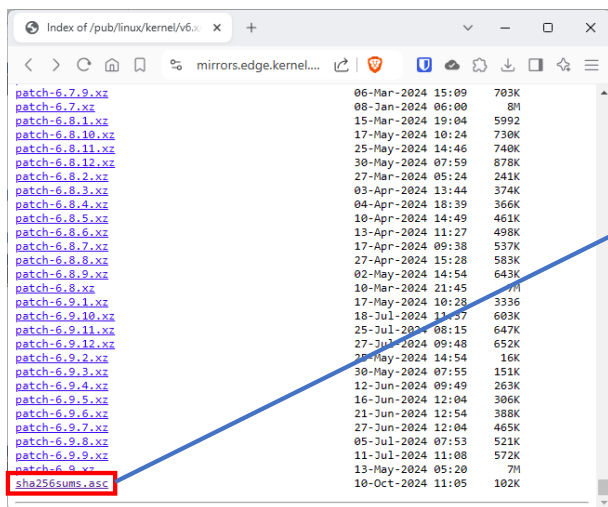
This is a real S/MIME signed email, the kind your mail client verifies silently when the signature banner appears. The header declares the message as multipart signed, which means the mail is a MIME container with two roles: the actual body as one part, and the cryptographic signature as another part, an attached file of type application/x-pkcs7-signature. The micalg field names the digest algorithm that was used, which tells you exactly which hash the signature is bound to, and therefore what a successful check actually guarantees. Reading these headers is reading the security contract of the message.

The flow the client performs: it splits the multipart container, extracts the body part and the signature part, recomputes the digest over the body bytes, and checks the signature against the signer's certificate, which is embedded inside the signature part. That embedding is the key design point: a first-time recipient gets the key material in the same message and can verify without any prior setup, no pre-exchanged keys, no directory lookup. The signature and the content travel together as a self-contained unit, which is what makes signed mail work between organizations that have never met.

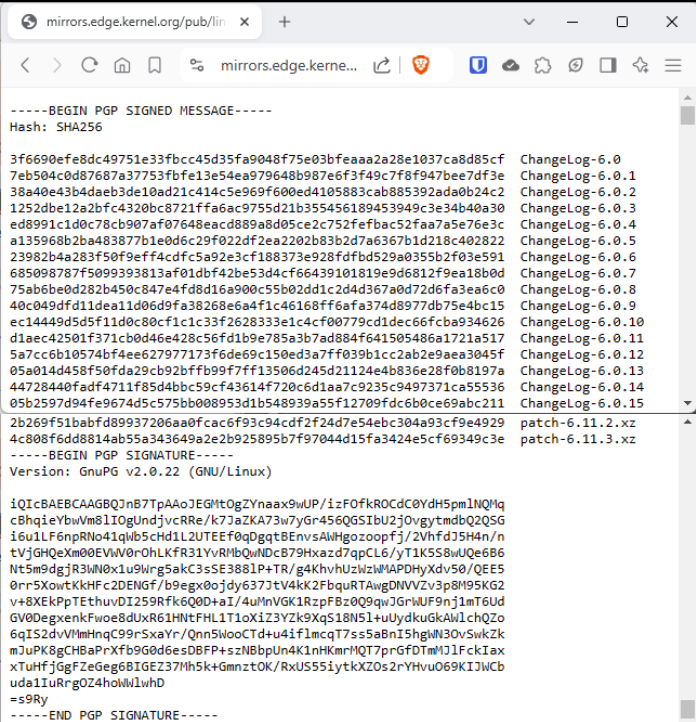
The detail that carries the lesson: the micalg here is SHA-1, by today's standards. This message dates from 2009, and it illustrates exactly why digest choices age: the structure is still sound, but a digest that can now be collided degrades the whole signature, because a collision lets an attacker substitute content while the check still passes. The fix is not a new container format; it is a stronger digest, modern clients use SHA-256 or better, and the general lesson is that the weakest component sets the floor: a sound structure built around a weak hash is still a weak signature.

The broader point is what mail signing does and does not do: it separates the message arrived from the message is from who it claims. A signed mail can still be forwarded, copied, delayed, or read by the forwarding party; the signature protects content and authorship, not delivery or privacy, that would require the encryption part of S/MIME. The same distinction applies to every signed artifact you will meet, a commit, a container, an update: the signature is evidence about the bytes and the signer, and nothing more.

Digital Signatures at kernel.org



patch-6.7.9.xz	06-Mar-2024 15:09	703K
patch-6.7.xz	08-Jan-2024 06:00	8M
patch-6.8.1.xz	15-Mar-2024 19:04	5992
patch-6.8.10.xz	17-May-2024 10:24	730K
patch-6.8.11.xz	25-May-2024 14:46	740K
patch-6.8.2.xz	30-May-2024 07:59	878K
patch-6.8.3.xz	27-Mar-2024 05:24	241K
patch-6.8.4.xz	03-Apr-2024 13:44	374K
patch-6.8.5.xz	04-Apr-2024 18:39	366K
patch-6.8.6.xz	10-Apr-2024 14:49	461K
patch-6.8.7.xz	13-Apr-2024 11:27	498K
patch-6.8.8.xz	17-Apr-2024 09:38	537K
patch-6.8.9.xz	27-Apr-2024 15:28	583K
patch-6.8.xz	02-May-2024 14:49	643K
patch-6.9.xz	10-Mar-2024 21:45	7M
patch-6.9.1.xz	17-May-2024 10:28	333K
patch-6.9.10.xz	18-Jul-2024 13:57	603K
patch-6.9.11.xz	25-Jul-2024 08:15	647K
patch-6.9.12.xz	27-Jul-2024 09:48	652K
patch-6.9.2.xz	20-May-2024 14:54	16K
patch-6.9.3.xz	30-May-2024 07:55	151K
patch-6.9.4.xz	12-Jun-2024 09:49	263K
patch-6.9.5.xz	16-Jun-2024 12:04	306K
patch-6.9.6.xz	21-Jun-2024 12:54	388K
patch-6.9.7.xz	27-Jun-2024 12:04	465K
patch-6.9.8.xz	05-Jul-2024 07:53	523K
patch-6.9.9.xz	11-Jul-2024 11:08	572K
patch-6.9.xz	13-May-2024 05:20	7M
sha256sums.asc	10-Oct-2024 11:05	102K



```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

3f6690efe8dc49751e33fbcc45d35fa9048f75e03bfaa2a28e1037ca8d85cf ChangeLog-6.0
7eb504c0d87687a37753bf13e54ea979648b987e6f349c7f8f947bee7df3e ChangeLog-6.0.1
38440e43b4daeb3de10ad21c414c5e969f500ed4105883cab885392ada0b24c2 ChangeLog-6.0.2
1252dbe12a2bfc4320bc8721ffa6ac9755d21b355456189453949c3e34b40a30 ChangeLog-6.0.3
ed8991c1d0c78cb907af07648eacdd889a8d05ce2c752fefbac52faa7a5e76e3c ChangeLog-6.0.4
a135968b2ba48377b1e0d6c29f022df2ea220283b2d7a6367b1d218c402822 ChangeLog-6.0.5
23982b4a283f50f9eff4cdcf5a92e3cf188373e928fddbd529a0355b2f03e591 ChangeLog-6.0.6
685098787f5099393813af01dbf42be53d4cf66439101819e9d6812f9ea18b0d ChangeLog-6.0.7
75ab6be0d282b450c847e4fd8d1d6a900c55b0d2d1c2d4367a0d72d6fa3ea6c0 ChangeLog-6.0.8
40c049dfd1dea11d06d9fa38268e6a4f1c46186ff6afa374d8977db75e4bc15 ChangeLog-6.0.9
ec14449d5d5f11d0c80cf1c1c3f2628333e1c4cf00779cd1dec66fcb934626 ChangeLog-6.0.10
diaec42501f371cb0d46e428c56fd1b9e785a3b7ad884f641505486a1721a517 ChangeLog-6.0.11
5a7cc6b10574bf4ee627977173f6de69c150ed3a7ff039b1cc2ab2e9aea3045f ChangeLog-6.0.12
05a014d458f50fda29c9b2bffb99f7f13506d245d21124e4b836e28f0b8197a ChangeLog-6.0.13
44728440fadf4711f85d4bbc59cf43614f720c6d1aa7c9235c9497371ca55536 ChangeLog-6.0.14
05b2597d94fe9674d5c575bb08953d1b548939a55f12709fcd6b0ce69abc211 ChangeLog-6.0.15
2b269f51babfd89937206aa0cac6f93c94cdf2f24d7e54ebc304a93cf9e4929 patch-6.11.2.xz
4c808f6dd8814ab55a343649a2e2b925895b7f97044d15fa342ae5cf69349c3e patch-6.11.3.xz
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.22 (GNU/Linux)

iQIcBAEBCAAGBQJnB7TAAoAEGHtOgZynaax9wUP/1zF0fkROCD0Ydh5pm1NQHQ
cBhQieYbVvm8II0gUndjvRRe/k7JaZAK73wYGr456QGSIBU2j0vgymdbQ2QSG
i6u1LF6npRNo41qwb5cHd1L2UTEEf0qDgt8ENvsAlHgozoopfj/2Vhfd5H4n/n
tVjGHQeXm00EVW0rOhLFR31YVRhQWnDC879Hxazd7qpCLy/1T1K558uUQeB6E
Nt5m9dgjR3Wn0x1u9Wrg5akC3s5E3881P+TR/g4KhvhuZuZwMAPDHyXdv50/QEE5
0rr5XowtKkHF2DENGf/b9egx00jdY637jT4Vkk2FbquRTAwgDNVZv3p8M95KG2
v+8XekPpTEthuvDI259RfK6Q0D+a1/4uMnVGK1RzpfBz009qWjGrWUF9nj1mT6Ud
GV0DegxenKfwoe8dUxR61HntFHL1T1oXIZ3YZK9XqS18W51+uJydkuGKA1chQzO
6qIS2dvVMmHnqC99rSxaYr/Qnn5WocCdT4u4iFlmcqT7ss5aBnI5ghW30vSwkZk
xTuHfjGgFZeG68IGE237Hh5k+GmnzOK/RxUS55y1ikXZ0s2rYvhu069K1JwCb
uda1IuRrgOZ4holWlwhD
=s9Ry
-----END PGP SIGNATURE-----
```

João Paulo Barraca, André Zúquete

SIO

28

This slide shows digital signatures where they matter most in practice: the kernel.org project signs its release tarballs and publishes the signatures (and the signing keys). Anyone downloading the kernel can verify that the file is exactly what the maintainers released, and that it came from them - not from a mirror that was compromised, and not from an attacker sitting on the download path.

The workflow is the standard one: download the tarball, download the signature, obtain the maintainers' public key from a trusted channel, and run the verification command. If all three match, you have integrity plus origin for the artifact you are about to build and deploy. The verification command is the same shape as everything in this lecture: digest the data, check the signature against the public key.

The subtlety is the trust in the key: the public key itself must be obtained out of band (published on the project's site, cross-signed, or announced through a known channel). A signature is only as strong as your trust in the key it was checked against - the public-key distribution problem from earlier, in action. A mirror can serve a valid signature for a tampered file only if it also holds the private key, which is why the key, not the file, is the thing to protect.

This is the everyday face of non-repudiation and integrity: you are not just checking a checksum (which anyone can recompute and publish alongside a tampered file), you are checking a claim only the key holder could make. It is the same mechanism as signed updates and signed containers, applied to source code - and it is why kernel developers sign every release they ship.

Security considerations: key sizes and the quantum horizon

- **RSA**
 - 1024-bit is deprecated
 - 2048-bit is the practical minimum
 - 3072-bit+ for long-term needs
- **ECC**
 - 256-bit gives about 128-bit security
 - 192-bit is considered weak
- **Shor's algorithm on a large quantum computer would break RSA and ECC alike**
 - Post-quantum (lattice-based) alternatives are being standardized now

SIO

Key size is not a formality; it is the dial that sets your real security level, and it should be read in security bits, not raw length. The RSA side of the table: 1024-bit is deprecated, 2048-bit is the practical minimum for new systems, and 3072-bit or more is the common guidance for long-lived secrets. The ECC side: 256-bit gives roughly 128-bit security, and 192-bit is considered weak. The same security-level thinking applies to both families: comparing a 256-bit curve to a 2048-bit modulus directly is a category error, you compare implied security levels, which is why the table exists.

The rule of thumb the table operationalizes: match the key size to how long the secret must stay secret. A session that lasts an hour does not need the same key as a document that must stay confidential for 20 years, because the threat model includes the growth of attacker capability over time. The table converts how long must this stay secret into how big must the key be, and it is a planning tool that gets updated as attacks improve. That is why a size that was strong a decade ago can be the minimum today and deprecated in a decade; new deployments should pick sizes with runway, not just the current minimum.

The quantum horizon is why the tables keep moving, and students should not skip it. Shor's algorithm, run on a sufficiently large fault-tolerant quantum computer, would break both factoring and discrete logarithms, which means RSA, Diffie-Hellman, and ECC alike, regardless of key size; there is no bigger-key escape inside those families. No such computer exists today, but long-lived secrets are already at risk from harvest-now-decrypt-later attacks: encrypted traffic is stored now, cheaply, and decrypted later when the hardware exists. That asymmetry, cheap storage now, expensive break later, is what makes the migration urgent for anything that must remain secret for years.

The practical response is migration, not panic: post-quantum cryptography, mostly lattice-based schemes, is being standardized and deployed in hybrid mode, classical plus post-quantum together, so a break in either half alone does not compromise the session. You will see hybrid key exchanges appearing in new protocol versions and server configurations, and that is the transition in action. Takeaway: treat every classical key size as having an expiration date, pick sizes with runway, and expect hybrid exchanges to become the norm within a few protocol generations.