Security

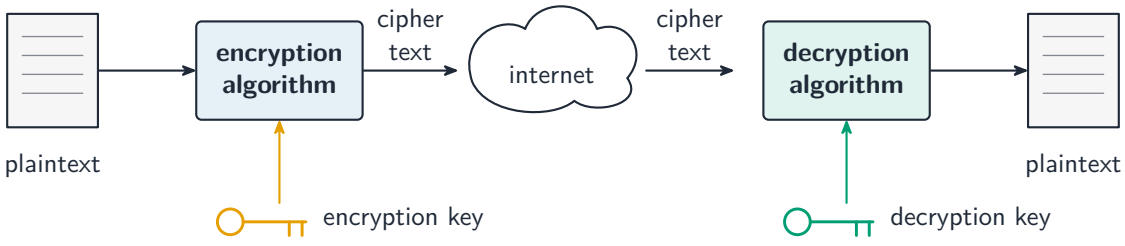
A-Level Computer Science

How encryption works

Encryption 加密 turns readable **plaintext** 明文 into unreadable **ciphertext** 密文 using a maths operation that depends on a **key**. Only someone with the right key can reverse it —**decryption** 解密—to get the plaintext back. An attacker who intercepts the ciphertext without the key sees only meaningless data, because trying every possible key would take far too long.

The Enigma machine encrypted messages in the Second World War —an early, mechanical cipher device

Image: Andy Li, CC0 (commons.wikimedia.org)



Encryption scrambles plaintext with a key; decryption reverses it

Symmetric encryption

Symmetric encryption 对称加密 uses the **same key** for both encryption and decryption, so sender and receiver must both hold the secret key. It is **fast** and good for **bulk data** (a whole disk, a video stream). Its problem is **key distribution** 密钥分发: how do you share the key safely in the first place? Asymmetric encryption solves this.

Asymmetric encryption (public-key)

Asymmetric encryption 非对称加密 gives each user a **pair** of related keys: a **public key** 公钥 they publish, and a **private key** 私钥 they keep secret. Data encrypted with the public key can be decrypted **only** with the matching private key, and vice versa.

encrypt with the recipient's **public key** — only their matching **private key** can decrypt

Each user has a public key to share and a private key to keep secret

To send a secret message to Alice: get her published public key, encrypt with it, and send. Only Alice —holding the matching private key —can decrypt. No prior key exchange is needed. The trade-off is that it is **much slower** than symmetric, so it is not used for large data.

A private key must stay secret, so it is sometimes kept on a small **hardware security key** 硬件安全密钥. You plug it in or tap it to prove who you are, and the secret key never leaves the device.

A hardware security key stores a secret key to prove who you are

Image: Yubico, Product image (www.yubico.com)

Hybrid approach (used by almost every real system)

Use asymmetric encryption to exchange a fresh **session key** 会话密钥, then use that symmetric key for the data:

1. the client makes a random session key.
2. it encrypts the session key with the server's public key.

3. the server decrypts it with its private key.
4. both ends now share the session key and use fast symmetric encryption for the rest.

This is how HTTPS and SSH work.

Hashing (related, not encryption)

A **cryptographic hash** 密码散列 function takes any input and gives a fixed-size **digest** 摘要 such that the same input always gives the same digest, it is infeasible to find two inputs with the same digest, and a tiny change in input changes the digest completely. Hashing is **one-way** —you cannot get the input back. It is used for storing password checks, **integrity** 完整性 checks, and digital signatures.

SSL / TLS

TLS 传输层安全 (Transport Layer Security, the successor to SSL) is a **protocol** that gives encryption and authentication for data sent over a network. It **encrypts** the data in transit, **authenticates** the server with a certificate, and provides **integrity** (detecting tampering).

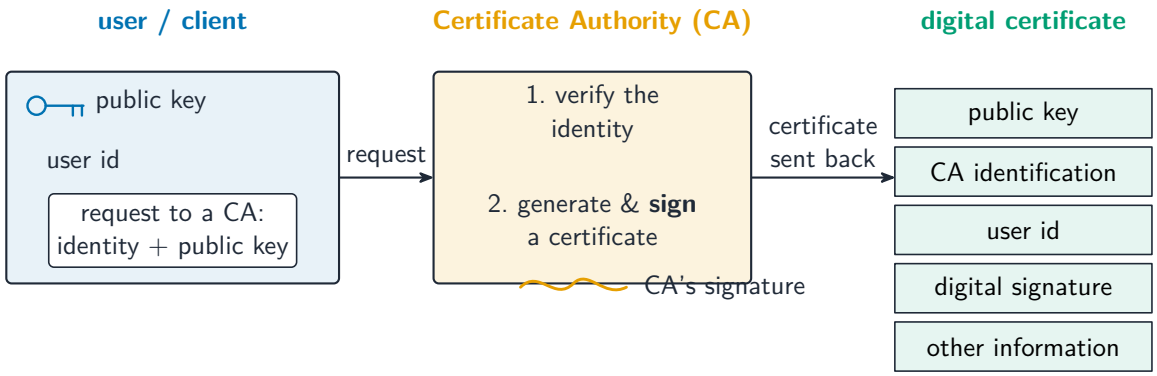
Outline of a TLS handshake:

1. the client connects and proposes cipher options.
2. the server picks one and sends its **digital certificate** (with its public key).
3. the client checks the certificate.
4. the two ends exchange a fresh **session key** using asymmetric crypto.
5. all later traffic uses fast symmetric encryption with the session key.

The result is an encrypted, authenticated, integrity-checked tunnel for higher-level protocols (HTTP, SMTP). It is appropriate wherever **sensitive information** is sent: HTTPS web browsing, online banking and payments, secure email, and VPNs.

Digital certificates

A **digital certificate** 数字证书 binds an identity (a domain, an organisation) to a public key, and is signed by a trusted **Certificate Authority** 证书颁发机构 (CA). It contains the subject (who it identifies), the subject's public key, the issuer (the CA), a validity period, and the CA's signature over all of it.



A Certificate Authority issues a digital certificate binding an identity to a public key

To verify one, the client (which holds a list of trusted root CAs):

1. checks the expiry dates.
2. checks the subject name matches the URL.
3. checks it is **signed by a trusted CA**, using the CA's public key to verify the signature.
4. follows the certificate chain up to a trusted root.

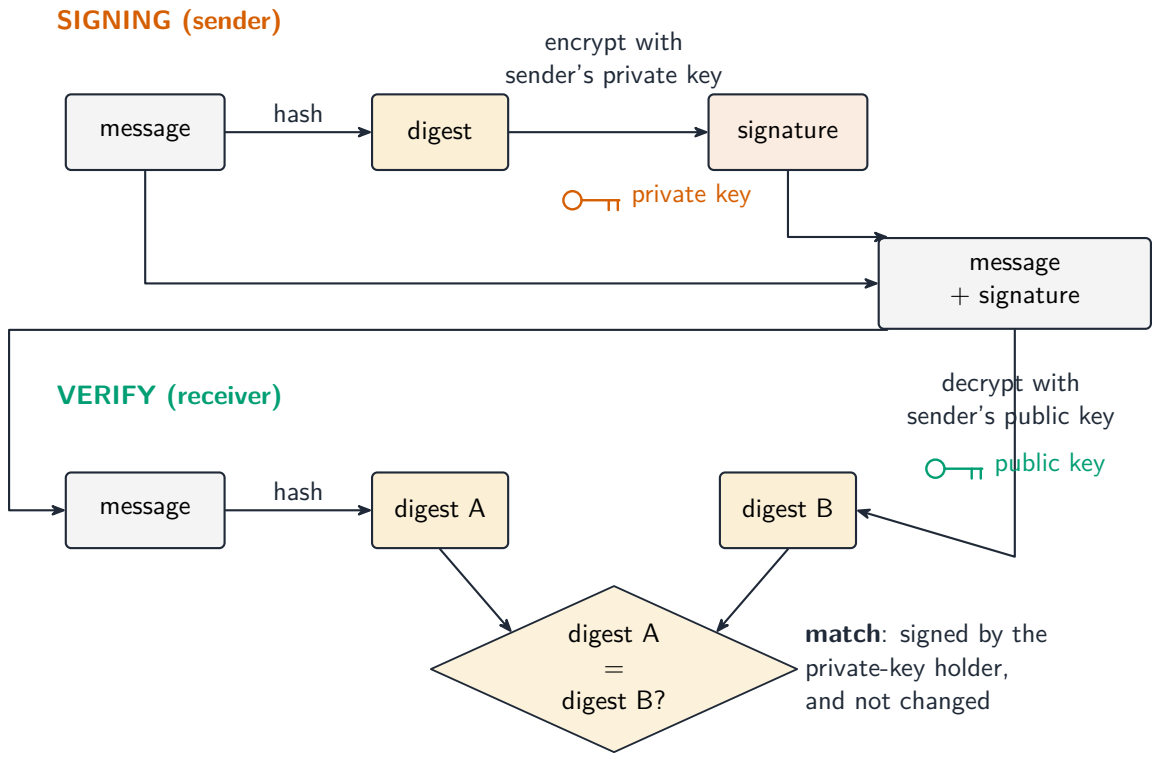
If anything fails, the browser shows the "Your connection is not private" warning. When it verifies cleanly, the client knows the identity was vetted by a trusted CA, the public key really belongs to that identity, and the certificate is current.

Digital signatures

A **digital signature** 数字签名 proves who signed a message **and** that it was not changed. To sign:

1. compute a cryptographic hash of the message.
2. **encrypt the hash with the sender's private key** —that is the signature.
3. send the message and the signature.

To verify: compute the hash of the received message; **decrypt the signature with the sender's public key** to get the sender's hash; compare. If they match, the message was signed by the holder of the private key (**authentication** 身份验证) and was not changed (integrity). A signature does **not** hide the message —for confidentiality as well, encrypt **and** sign.



Signing hashes the message and encrypts the digest with the private key; the receiver checks it with the public key

Putting it together

A secure request to `https://www.bank.com`: the server sends its certificate; the client verifies it against trusted CAs; the client uses the server's public key to exchange a session key; then data flows encrypted with that key. Encryption stops eavesdroppers, the certificate proves the server's identity, and integrity checks stop a **man-in-the-middle** 中间人攻击 altering the data.