
Computer Security and Privacy

BA5 • 2026-2027

Contents

1	Lecture 1 : Basics	5
1.1	Security properties	5
1.2	Security policy	5
1.3	The adversary	6
1.4	Security mechanisms	6
1.5	Showing a system is secure	7
1.6	Security engineering process	7
2	Lecture 2 : Principles of computer security	8
2.1	1 – Economy of mechanism	8
2.2	2 – Fail-safe defaults	9
2.3	3 – Complete mediation	9
2.4	4 – Open design	10
2.5	5 – Separation of privilege	11
2.6	6 – Least privilege	11
2.7	7 – Least common mechanism	12
2.8	8 – Psychological acceptability	12
2.9	Extra principles from physical security	13
3	Lecture 3 : Access control basics and DAC	14
3.1	What is access control?	14
3.2	Implementing access control	15
3.3	DAC vs. MAC	15
3.4	The access control matrix	15
3.5	Access Control Lists (ACLs)	16
3.5.1	Role Based Access Control (RBAC)	16
3.5.2	Group Based Access Control	17
3.6	Capabilities	17
3.7	Ambient authority and the confused deputy	18
3.8	DAC in practice: UNIX	19
3.9	DAC in practice: Windows	21
4	Lecture 4 : Mandatory Access Control	23
4.1	Bell-La Padula (BLP): protecting confidentiality	23
4.1.1	Security levels and dominance	24

4.1.2	The three BLP properties	25
4.1.3	Covert channels	26
4.1.4	Declassification	26
4.2	BIBA: protecting integrity	27
4.2.1	BIBA variants	27
4.2.2	Sanitization	28
4.2.3	Principles to support integrity	28
4.3	Chinese Wall model: conflicts of interest	29
5	Lecture 5 : Applied cryptography	30
5.1	Module 1 – Introduction	30
5.2	Module 2 – Adversaries in cryptography	33
5.3	Module 3 – Perfect secrecy and the one-time pad	34
5.4	Module 4 – From OTP to stream ciphers	37
5.5	Module 5 – Public-key cryptography	39

These notes cover the core vocabulary and reasoning framework of computer security: adversaries, threat models, security policies, mechanisms, and how to argue that a system is secure.

CHAPTER 1

Lecture 1 : Basics

In a nutshell

What is this lecture about? The vocabulary of security: what we want to protect (*assets*), against whom (the *adversary*), what “secure” means (the *security policy*) and with what tools (*mechanisms*).

Why is it useful? Every later lecture reuses these words. Before building or attacking anything, you must be able to say clearly *what* must hold and *against which attacker* — otherwise “secure” means nothing.

Why a course on security?

Ordinary engineering asks for **correctness**, **safety** and **robustness** against accidental errors. Security additionally requires that properties hold in the presence of a **deliberate, strategic adversary**.

1.1 Security properties

Properties of a computer system must hold in the presence of a **resourced strategic adversary**.

CIA triad

- **Confidentiality** — prevention of unauthorized *disclosure* of information.
- **Integrity** — prevention of unauthorized *modification* of information.
- **Availability** — prevention of unauthorized *denial of service*.

CIA on a bank account

The adversary should not be able to *read* my statement (confidentiality), *change* my balance (integrity), or *block* my access to the account (availability).

Beyond the CIA triad

Modern systems also care about **authenticity**, **anonymity**, **isolation**, and more.

1.2 Security policy

Security policy

A high-level description of the **security properties** that must hold, in relation to **assets** and **principals**.

- **Assets (objects)** — anything of value that needs protection (data, files, memory, ...).
- **Principals (subjects)** — people, programs, services, ... (never the adversary).

Properties in terms of principals and assets

- **Confidentiality** — authorized **users** may read a **file**.
- **Integrity** — authorized **programs** may write a **file**.
- **Availability** — authorized **services** can access a **file**.

1.3 The adversary

Threat model

Describes the **resources** available to the adversary and its **capabilities** (observe, influence, corrupt, ...).

Adversary

A malicious entity aiming to breach the security policy. The adversary is **strategic**: it always picks the **optimal** way to use its resources to violate a security property.

Key vocabulary

- **Vulnerability** — a specific weakness an adversary could exploit (*e.g. a password shown in plaintext*).
- **Threat** — the feared event, i.e. the adversary's goal (*e.g. a hacker wants to steal money from the bank*).
- **Harm** — what actually happens if the threat materializes (*e.g. the adversary steals the money*).

If the properties hold within the threat model, no vulnerability can be exploited to realize a threat, so no harm occurs.

1.4 Security mechanisms

Security mechanism

A technical mechanism (software, hardware, cryptography, procedures, ...) used to ensure that the security policy is not violated by an adversary within the threat model.

Policy → mechanism

Policy: the transaction log must not be tampered with by a single employee.

Mechanism: replicate the log across multiple machines so that no employee has access to all copies.

Composing mechanisms

Security does **not** increase linearly with the number of mechanisms.

- **Defence in depth** (AND) — secure as long as *one* mechanism holds.
- **Weakest link** (OR) — insecure as soon as *any one* mechanism fails.

Humans are part of the system

Social engineering, phishing, and weak/reused/written-down passwords are frequently the weakest link — but that is no excuse to neglect the rest of the system.

1.5 Showing a system is secure

Attacker vs. defender

Attacker: one way to violate one property is enough (within the threat model).

Defender: must show **no** adversary strategy violates the policy — security is always relative to a **specific threat model**.

Secure

A system is **secure** if an adversary constrained by a specific threat model cannot violate the security policy.

Security argument

A rigorous argument (verbal or mathematical) that the security mechanisms in place are effective in maintaining the security policy — **subject to the assumptions of the threat model**.

Composition is hard

A useful threat model must constrain the adversary, otherwise no security argument is possible. Security arguments for the **composition** of mechanisms are notoriously hard to get right.

1.6 Security engineering process

Three-stage process

1. **High-level specification** — architecture, security policy (principals, assets, properties), threat model.
2. **Security design** — select/design mechanisms; state the security argument (which mechanism maintains which property).
3. **Secure implementation** — implement the mechanisms, ensure conformance to the design, perform security testing.

Summary

- Security problems always involve an **adversary**, who is **strategic**.
- The adversary's capabilities define a **threat model**.
- **Security mechanisms** aim at fulfilling a **security policy within a threat model**.
- Showing security means providing a **security argument**.

CHAPTER 2

Lecture 2 : Principles of computer security

In a nutshell

What is this lecture about? Nobody knows how to build a system without bugs, so experience gave us **8 (+2) rules of thumb** (Saltzer & Schroeder) to design security mechanisms: keep it simple, refuse by default, check every access, do not rely on secrecy of the design, never trust a single person, give only the rights needed, share as little as possible, make security easy to use.

Why is it useful? These principles are a **toolbox to judge a design**: when you look at a system (or an attack), you can ask “which principle was violated?”. Most real-world breaches (Heartbleed, MongoDB, S3 leaks, ...) are a principle that was broken because of a bad tradeoff.

Where do the principles come from?

Saltzer & Schroeder, *The Protection of Information in Computer Systems* (1975), abbreviated [SS75]: **8 (+2) principles** at the core of security engineering practice. Experience has shown that they **guide** the design and contribute to an implementation without security flaws. They are still used today (e.g. to analyse the threat model of machine learning systems).

Heartbleed (CVE-2014-0160) — why simplicity matters

In OpenSSL (used for HTTPS), the *Heartbeat* feature lets a client send a small piece of data that the server echoes back. The attacker sends 1 byte but **claims** to have sent 64 000 bytes; the server, **without checking the actual length**, replies with the byte plus 63 999 bytes of its private memory.

- **Feature complexity** — Heartbeat was a non-essential feature: the added complexity created a **new attack surface**.
- **Codebase complexity** — the bug was a single missing bounds check, hidden for two years because the huge codebase made auditing difficult.

2.1 1 – Economy of mechanism

Economy of mechanism

“Keep the [security mechanism / implementation] design as **simple and small** as possible.” (KISS principle)

Why?

The mechanism needs to be **easy to audit and verify**. Operational testing is not appropriate to evaluate security (penetration testing is nevertheless valuable).

Trusted Computing Base (TCB)

Every component of the system (hardware / firmware / software) on which the security policy relies. The TCB is **trusted** to operate correctly for the security policy to hold.

Consequences of the TCB

- If something goes wrong **within** the TCB, **the security policy may be violated**; a failure **outside** the TCB should not matter for the policy.
- The TCB must be kept **small** to **ease verification** (economy of mechanism) and **diminish the attack surface**.

The verb “to trust”

The only proper use of “to trust” in security engineering is “**X trusts Y will do Z**”.

2.2 2 – Fail-safe defaults**Fail-safe defaults**

“Base access decisions on **permission** rather than **exclusion**.” [SS75]

If something fails, be as secure as if it does not fail: errors / uncertainty should err **on the side of the security policy**. Do **not** try to fix!

Whitelist, not blacklist

Use a **whitelist** (list what is allowed), not a blacklist (list what is forbidden): a **lack of permission is easy to detect and solve**, whereas a forgotten exclusion silently grants access.

Fail-safe defaults

- Security door: if no permission, do not open.
- Form input: if no permission to write in X , do not write anywhere.

MongoDB ransomware (2017) — ease of use vs. security

For a while, the default installation had **authentication turned off** and was **bound to all network interfaces** (0.0.0.0) instead of localhost. Attackers scanned the whole internet, connected without password, stole/deleted the data and left a ransom note. Unsafe defaults + users who do not read the manual = no access control.

Ransomware

A ransomware **denies you access to your own data**, most commonly by **encrypting your files**; the attacker then demands a **ransom payment** in exchange for the key needed to decrypt them. Its prevalence exploded with the arrival of cryptocurrencies.

2.3 3 – Complete mediation**Complete mediation**

“**Every access to every object** must be checked for authority.” [SS75]

Reference monitor

The component that **mediates ALL actions from subjects on objects** and ensures they are according to the **policy**. It takes as input the request (subject $\rightarrow$ object) and the policy, and writes to an **audit log**.

Difficult to implement

- **Performance** — checking everything is slow.
- **Time to check vs. time to use** — the permission may change between the check and the use.
- **Modern distributed systems** — you can only check what you see!

Missing mediation — the “Gold Member” cookie

A streaming platform checks the subscription in the database **only once at login**, then stores `{'user': 'Alice', 'status': 'paid'}` in a session cookie and only looks at the cookie afterwards. Alice subscribes, logs in, unsubscribes (the DB is correctly updated to “unpaid”), and can still watch premium content until her session expires: **violation of complete mediation**.

2.4 4 – Open design

Open design

“The design should **not be secret**.” [SS75]

When you design, **algorithms are public**; only **key elements** are kept secret:

- **Crypto** — only keep the key secret;
- **Authentication** — only keep the password secret;
- **Obfuscation** — only keep the used noise secret.

Historical formulations

- **Kerckhoffs** (1883): the design of a system should not require secrecy.
- **Shannon** (1949): “the enemy knows the system” — design systems assuming the enemy immediately gains full familiarity with them.
- **Baran** (1964): the paradox of secrecy about secrecy — without widespread scrutiny, weaknesses are more likely to be overlooked.

Why open design?

- Open design results in **better & easier auditing**. **Linus’ law** (Raymond, 1997): “given enough eyeballs, all bugs are shallow” — more of a mantra: it needs many eyeballs, and defect density is high.
- **Secrecy is unrealistic**: assuming the design stays secret is a way to build a **bad threat model**.
- Key principle behind the academic discipline devoted to understanding computer security.

Famous failure of a closed design: **GSM encryption**.

Security through obscurity does not work

- **Hidden instance (2017):** data of 198 million US voters stored on a public server with no authentication; the only “protection” was a “semi-secret” URL. A researcher found it.
- **Addresses are not safe:** the whole IPv4 space ($2^{32} \approx 4$ billion addresses) could be scanned in about 3 minutes in 2013 — picking a random port (0–65535) **is not hiding the service**.

2.5 5 – Separation of privilege**Privilege**

A **privilege** allows a user to perform an action on a computer system that may have security consequences (e.g. create a file in a directory, access a device, write to a socket for communicating over the Internet).

Separation of privilege

“No single accident, deception, or breach of trust is sufficient to compromise the protected information.” [SS75]

Requiring **multiple conditions** to execute an action improves security (e.g. two keys to open a safe, two factors to authenticate).

Problems

Availability? (more conditions = more ways to be blocked), **Responsibility?** (who is accountable?), **Complexity!**

Payment system of a business

Goal: the attacker must not steal money. **Threat model:** the attacker can corrupt *at most one* principal (no collusion). If a manager can both *create a vendor*, *create an order* and *approve* it, a single corrupted manager registers a fake shell company and pays it.

Solution: split roles — Procurement can **only** create/approve vendors; Finance can **only** approve payments for vendors already in the system. Reminiscent of **separation of powers** / “checks and balances”.

2.6 6 – Least privilege**Least privilege**

“Every program and every user of the system should operate using the **least set of privileges necessary** to complete the job.” [SS75]

Rights are **added as needed** and **discarded after use**.

Least privilege in practice

- **Damage control** — minimize high-privilege actions & interactions (related to economy of mechanism: small TCB).
- **“Need-to-know” principle** — e.g. guest accounts at EPFL, **data minimization** principle (data protection).

2.7 7 – Least common mechanism

Least common mechanism

“Minimize the amount of mechanism **common to more than one user** and depended on by all users.” [SS75]

Every shared mechanism represents a **potential information path between users** and must be designed with great care so that it does not unintentionally compromise security.

Link with economy of mechanism

- **(Design)** interactions make it hard to validate the security design;
- **(Implementation)** interactions may lead to unintentional leaks of information — **unintended channels**: use of `/tmp`, shared cache.

Mechanisms, not code

This principle refers to **mechanisms**, not to code: reusable code that has been tested many times is **helpful** for security. The problem is the **isolation of data**.

Symlink attack in `/tmp`

A **symbolic link** (symlink) is a “shortcut” file: when the OS accesses it, it is automatically redirected to the real file it points to.

A root program writes `echo "Service is starting..." > /tmp/starting.log`. A non-root user (corrupted by the adversary) also has write access to `/tmp` and runs `ln -s /var/log/dmesg /tmp/starting.log`: the root program now overwrites a system log, breaking **log integrity**. The common mechanism is the `/tmp` directory, shared between low-privilege users and root processes. (Modern Linux mitigates it in `/tmp`, but it still works e.g. in `/home/user`.)

2.8 8 – Psychological acceptability

Psychological acceptability

“It is essential that the human interface be designed for **ease of use**, so that users routinely and automatically apply the protection mechanisms correctly.” [SS75]

What it implies

- **Hide the complexity** introduced by security mechanisms: they should not make the resource more difficult to access than if they were not present.
- The **mental model** of the (honest) users must match the security policy and security mechanisms.
- **Cultural acceptability** — not all mechanisms are acceptable everywhere (face recognition in cultures that cover the face, register of everyone who sleeps in a dorm).

Amazon S3 “Any Authenticated AWS User”

Psychological acceptability is **not just for end users**: system administrators are also users of a configuration UI. Admins thought the option meant “any authenticated user *in my account*” whereas it meant “anyone with *any* AWS account” (free to create) $\Rightarrow$ several leaks from 2017. In 2023 AWS added a fail-safe default (block public access) — **safer defaults have tradeoffs** (users struggled to share buckets).

2.9 Extra principles from physical security**Difficult to transpose to computer security!**

The last two principles come from physical security and are hard to apply to computer systems.

9 – Work factor

“Compare the **cost of circumventing** the mechanism with the **resources of a potential attacker**.” [SS75]

It helps **refining the threat model**, but cost is **difficult to quantify** (cost of compromising insiders? of finding a bug? monetization?).

10 – Compromise recording

“**Reliably record** that a compromise of information has occurred [...] in place of more elaborate mechanisms that completely prevent loss.” [SS75]

Keep **tamper-evident logs**: they may enable recovery (integrity).

Logs are not magic

What if you cannot recover (e.g. confidentiality already lost)? How to keep the log’s integrity? Logs may be a vulnerability (privacy)? Logging the log (availability)? **Logging is not a guarantee that the compromise is detected.**

Summary

“Since no one knows how to build a system without flaws, the alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws.” [SS75]

- **Economy of mechanism** — keep it simple!
- **Fail-safe defaults** — safe defaults; if there is a problem, your move should comply with the policy.
- **Complete mediation** — verify *every* action.
- **Open design** — make the design (and implementation) available.
- **Separation of privilege** — never rely on *only one* entity or action.
- **Least privilege** — only the privileges needed, only while needed.
- **Least common mechanism** — minimize shared mechanisms (isolate data).
- **Psychological acceptability** — security must be easy to use correctly.

Principles help identify safe and unsafe **patterns**: do **not** use them as a blind checklist, but as tools to weigh design decisions. They are easy to violate — typically through **improperly evaluated tradeoffs**.

CHAPTER 3

Lecture 3 : Access control basics and DAC

In a nutshell

What is this lecture about? **Access control** answers one question: “is *this* user allowed to do *this* action on *this* file?”. In **DAC**, the **owner** of a file decides who can use it (like on your laptop or on Facebook). We see how to store these permissions (matrix, ACLs, capabilities, roles, groups), a classic trap (the *confused deputy*), and how Linux and Windows do it in practice.

Why is it useful? Access control is the **first line of defense** and is everywhere: apps, databases, operating systems, hardware. Understanding it lets you read and set file permissions correctly (`ls -l`, `chmod`, `sudo`) and spot programs that can be tricked into using their rights for an attacker.

3.1 What is access control?

Access control

Security mechanism that ensures that “all accesses and actions on objects by principals are **within** the security policy”.

Questions an access control system answers

Can Alice read file `/users/Bob/readme.txt`? Can Bob open a TCP socket to `http://www.abc.com/`? Can Charlie write to row 15 of the table GRADES?

The answer is either **authorized** (“has permission”) or **unauthorized** (“access denied”): only events within the security policy may happen.

Why is access control so important?

Access control is the **first line of defense**, thus **it is used everywhere**:

- **Applications** — online social networks, email servers, cloud storage;
- **Middleware** — database management systems (DBMS);
- **Operating system** — access to files, directories, ports, ...;
- **Hardware** — memory, registers, privileges.

Authentication vs. authorization

- **Authentication** — the system needs to **bind the actor to a principal** before authorization. A **principal** is an abstract entity that is authorized to act (users, connections, processes).
- **Authorization** — the system needs to decide whether the **principal is authorized**: this is **access control**.

TCB

The mechanisms that do authentication and authorization are in the **Trusted Computing Base**!

3.2 Implementing access control

“Checks soup” vs. reference monitor

- **What NOT to do: “checks soup”** — add checks all over the program, implementing the decision **in-line** based on the policy (`if user == Alice or user == Bob: open(...)`).
- **What you SHOULD do: systematic calls to a “reference monitor”** — all over the program add checks that **call the monitor**: they state the authorization required and provide evidence about principals and objects; a **“central” subsystem** establishes whether the checks pass or not (e.g. Apache Shiro: `subject.isPermitted("user:delete:jsmith")`).

Least common mechanism?

A central reference monitor is a mechanism shared by everybody — but this is fine: least common mechanism is about the isolation of data, and a single well-tested monitor is easier to verify (economy of mechanism, complete mediation).

3.3 DAC vs. MAC

Who decides the access policy? Two approaches

- **Discretionary Access Control (DAC)** — **object owners** assign permissions (ownership of resources). E.g. Windows, Linux, macOS, social networks.
- **Mandatory Access Control (MAC)** — a **central security policy** assigns permissions. For organizations with need for central controls: military (confidentiality), hospital (confidentiality and integrity), banking (integrity).

Origin of protection mechanisms (Lampson, 1971)

The original motivation was to keep **one user’s malice or error from harming other users**: by destroying/modifying another user’s data, by reading/copying it without permission, or by degrading the service another user gets (the extreme case being a crash of the system). All these reasons are just as strong if “user” is replaced by **“program”**.

3.4 The access control matrix

Access control matrix

An **abstract** representation of all **permitted** triplets (**subject, object, access right**) within a system. DAC policies are often conceptualized as an access control matrix (Lampson).

- **Subjects (principals)** — entities within an IT system (a user, a process, a service).
- **Objects (assets)** — resources that (some) subject may access or use (a file, a folder, a row in a database, the system’s memory, a machine in the network, a printer, a page in a website).
- **Operations** — in abstract, subjects can **observe** and/or **alter** objects (read, write, append, execute).

Access control matrix

$S = \{\text{Alice}, \text{Bob}\}$, $O = \{\text{file1}, \text{file2}, \text{file3}\}$, $A = \{\text{read}, \text{write}\}$.

	file1	file2	file3
Alice	read, write		read
Bob		read, write	read, write

Can Alice read file1? Can Bob write file1? Can file3 be written by Alice?

The matrix is an abstract concept

Not for direct implementation: size $O(f \cdot u)$ (f files, u users), which is huge with thousands of files or hundreds of users. Moreover it is:

- **usually very sparse** — wasteful;
- **inefficient to access** — need to keep the matrix in fast memory?
- **not extensible** — add a file? add a user?

3.5 Access Control Lists (ACLs)**Access Control List (ACL)**

Associate permissions to **objects**: each object stores the **column** of the matrix, i.e. the list of (subject, rights) pairs. **Blanks are not stored**.

file3: $\{(\text{Alice}, \text{read}), (\text{Bob}, \text{read/write})\}$

ACLs: pros and cons

Pros: can be stored close to / with the resource; easy to determine who can access a resource; easy to revoke rights by resource.

Cons: difficult to check at runtime; difficult to audit all rights of a user; difficult to remove all permissions from a user (better remove authentication!); difficult delegation.

3.5.1 Role Based Access Control (RBAC)**Role Based Access Control (RBAC)**

Systems have too many subjects that come and go (large dynamic ACLs), but subjects are similar to each other and need the same rights (a doctor has the same privileges as another doctor).

1. assign permissions to **roles**;
2. assign **roles** to subjects;
3. subjects select an **active role** — they have the permissions of the active role.

RBAC problems

1. **Role explosion** — temptation to create fine-grained roles, denying the benefits of RBAC.
2. **Limited expressiveness** of simple RBAC — problems implementing least privilege; some roles are relative (“Alice’s doctor” vs. “any doctor”).
3. **Separation of duty is difficult** — “two doctors are needed to authorize a procedure”: the mechanism needs to ensure they are distinct!

3.5.2 Group Based Access Control**Group Based Access Control**

Observation: some permissions are always needed together (e.g. access to sockets and network interface).

1. assign permissions to access objects to **groups**;
2. assign subjects to **groups**;
3. subjects have the permissions of **all** their groups.

Negative permissions (explicit denials for a subject) can be added to implement fine-grained policies.

RBAC vs. groups

With **roles**, a subject has the permissions of the *one active* role; with **groups**, a subject has the union of the permissions of *all* its groups.

Negative permissions are checked first

If group permissions were checked first, a group would grant the access and the negative permission would never be applied. Checking **negative permissions first** is the fail-safe choice.

3.6 Capabilities**Capabilities**

Associate permissions to **subjects**: each subject stores the **row** of the matrix, i.e. the list of (object, rights) pairs. **Blanks are not stored**.

Alice: {(file1, read/write), (file3, read)}

Capabilities: pros and cons

Pros: can be stored with the subject (portable!); easy to audit all subject permissions; delegating is “simple”.

Cons: revoking permission on one object is hard; transferability — once the capability is given, how can we prevent sharing?; authenticity — how to check it?

ACL vs. capabilities

	ACL	Capabilities
Stored with	the object (column)	the subject (row)
Who can access <i>o</i> ?	easy	hard
What can <i>s</i> access?	hard	easy
Revocation	easy per object	hard per object
Delegation	difficult	“simple”

History of capability-based security

Introduced by Dennis & Van Horn (1966); used in the Plessey System 250 (1972), CAP computer (1976), Intel iAPX 432 (1981). The industry then adopted UNIX and its ACLs (1982, Linux 1991). Revival: seL4 formally verified microkernel (2009), CHERI (2015), Google Fuchsia (2016), WASI (2019), ARM Morello (2022), Scala capabilities at EPFL.

3.7 Ambient authority and the confused deputy**Ambient authority**

Ambient authority is used by a subject if, for an action to succeed, it **only** needs to specify the **operation** and the **names** of the involved object(s). The **subject** (with authority) is **implicit**: in `open("file1", "rw")` the subject is missing, it is understood to be the process owner.

Ambient authority: pros and cons

Pro: no need to repeat the subject all the time (usability).

Cons: the program cannot check permissions for whom it acts; least privilege becomes harder to enforce; **confused deputy problem**!

Confused deputy problem

Problem with ambient authority: a **privileged program** can be **tricked to misuse its authority**.

Pay-per-use compiler

The compiler receives (`input`, `output`), compiles `input`, writes usage into `bill` and errors into `output`.

	<code>input</code>	<code>output</code>	<code>bill</code>
Alice	write	read	read
Compiler	read	read, write	read, write

Alice cannot write `bill`, but she calls the compiler with (`input`, `bill`): the compiler writes the errors into `output = bill` **with its own authority** $\Rightarrow$ `bill` is corrupted. The compiler is the **confused deputy**. (Same idea: a student with a card asks someone authorized to badge them into a room.)

How to avoid confused deputies

Ambient authority is used for convenience in many real systems (OS services, web servers, ...). Solutions:

1. re-implement access control in the privileged process;
2. let the privileged process check authorization for Alice;
3. **capabilities can help!** The compiler holds a capability for `bill`; to compile, Alice must give a capability for the `output` file — she cannot give a capability for writing `bill`, so she cannot confuse anyone.

3.8 DAC in practice: UNIX**UNIX principals and groups**

- **User Identities (UIDs)** and **Group Identities (GIDs)**: originally 16-bit (now 32-bit) numbers. Special UIDs: `-2`, `0`, `1`, ...
- Each user has its own directory `/home/username`; user accounts are in `/etc/passwd` (`username:password:UID:GID:info:home:shell`).
- Users belong to one or more groups: primary group (`/etc/passwd`), other groups (`/etc/group`).

UNIX security architecture

- **Everything is a file**; each user “owns” a set of files.
- Each file has a simple **ACL** expressing its access control policy. The owner can **modify the permissions of the file** (but cannot give away ownership). System files are owned by special users that can make system operations.
- All processes run by a user run **with that user’s privileges** $\Rightarrow$ **ambient authority!**

UNIX file ACLs

Each file is assigned an **owner UID** and a **GID**, and has **9 permission bits**: 3 actions (read, write, execute) $\times$ 3 subjects (**owner**, **group**, **other**). Plus 3 attributes: **suid**, **sgid** and **sticky**.

Different semantics for **directories**: Read $\rightarrow$ list files, Write $\rightarrow$ add files, Exec $\rightarrow$ traverse the directory (`cd`).

Reading `ls -l`

```
drwxrwxr-x 1 catronco catronco 4096 Sep 16 14:23 examplendir
```

First char: `d` = directory, `-` = file; then `rw` for owner, `rw` for group, `r-x` for others; then number of links, owner, group, size, last modified, filename.

The owner changes permissions with `chmod: +r, -w`, octal `666`, `662`, sticky `+t` or `1666`, `suid +s` or `4666`.

UNIX access control in action

Compare the **UID/GID** of the **process** trying to perform the action with the state of the file (owner, group, mode bits). **Order matters**:

1. if the **UID** says you are the owner: check the bits for **owner**;
2. if not owner, but your group is the file's group: check the bits for **group**;
3. otherwise check the bits for **other**.

The **root** user is **never denied access**.

Order matters

Only *one* class of bits is checked: an owner with **--** for owner but **rwX** for group is denied, even if they are in the group.

Super user (root)

Special user account with **User ID 0**: accesses system files and special operations, can access anything ((almost) all security checks disabled). **root is in the TCB!**

Never login as root: use **sudo** (run one command as root) or **su** (switch user).

suid / sgid

Setuid and setgid bits indicate that a file is **not run with the privileges of the launcher**, but **with the privileges of the owner** user/group. Useful for normal users to access system services that need system privileges, respecting least privilege, e.g. changing a password: `-rwsr-xr-x 1 root root .../bin/passwd`.

Setuid root programs are dangerous

They are in the TCB: how do you know that a **suid** program does what it is meant to do, and *only* what it is meant to do? (e.g. a **setuid-root python** gives root to anyone.)

Sticky bit

"Restricted deletion bit" (`chmod +t`).

- **Directories**: prevents unprivileged users from removing or renaming a file in the directory **unless they own the file** (e.g. `/tmp`: users can only edit their own files).
- **Files**: historically prevented a program from being moved from swap for fast load; Linux currently ignores the bit.

Nobody

Special user (**User ID -2**) that owns no files and belongs to no user. Safer user to execute code you do not know (particularly obfuscated code): limits damages if it misbehaves / gets compromised.

Exercise — ACL in Linux

```
drwxr-x-- 2 manager_A core_devs 4096 Sep 24 11:35 /project_beta
```

manager_A is the directory owner, a member of **core_devs**, and is **sudo**; **developer_B** is a member of **core_devs**; **intern_C** is **not** a member of **core_devs**.

For each user, determine which of `cd /project_beta`, `ls /project_beta` and `touch /project_beta/<file>` succeed or fail, and explain why based on the DAC rules.

Same state, with a new subdirectory and a file inside it:

```
drwxr-xr-x 2 developer_B core_devs 4096 Sep 24 12:00 /project_beta/Foo
```

```
-rw-r-r- 1 developer_B core_devs 512 Sep 24 12:05 .../Foo/intern_project.txt
```

Q1: `/project_beta` is owned by **manager_A**, but `Foo` is owned by **developer_B**. Find a scenario (sequence of commands) that would have led to this situation.

Q2: **intern_C** runs `cd /project_beta/Foo` then `cat intern_project.txt`. Do they work?

Q3: **intern_C** runs `cat /project_beta/Foo/intern_project.txt` from their home directory. Does it work?

Q4: Who do they need to contact to fix the problem?

3.9 DAC in practice: Windows

Windows access control

Principals = users, machines, groups, ...; objects = files, registry keys, printers, ...

- Each object has a **Discretionary Access Control List (DACL)**.
- Each process (or thread) has an **access token** with: the login user account (the process “runs as” this user), all groups of which the user is a member (recursively!), all privileges assigned to these groups.

The DACL is compared with the process’ access token **when creating a handle to the object**.

DACL

A list of **Access Control Entries (ACEs)**, each with a **type** (negative/positive, i.e. access denied/allowed), a **principal**, and **permissions** (more fine-grained than UNIX), plus flags and others. Negative entries come **first** so that a denial cannot be bypassed by an earlier allow.

DACL evaluation

ACE1: *deny* Andrew read/write/execute; ACE2: *allow* Group A write; ACE3: *allow* Everyone read/execute. Thread A (Andrew, groups A, B, C) is denied by ACE1. Thread B (Jane, group A) is granted write (ACE2) and read/execute (ACE3).

Least privilege by default

Windows runs with least privilege by default; elevated rights require “Run as administrator”.

Summary

- **Discretionary Access Control:** owners establish permissions.
- Conceptualized as an **access control matrix**.
- Implemented in two flavors: **access control lists** (permissions associated to objects) and **capabilities** (permissions associated to subjects).
- When relying on access control, always think about **confused deputies**.

CHAPTER 4

Lecture 4 : Mandatory Access Control

In a nutshell

What is this lecture about? In **MAC**, it is not the owner but a **central policy** that decides, and nobody can override it. We study three classic “recipes” (security models): **Bell-La Padula** to keep secrets from leaking down (army: no read up, no write down), **BIBA** to keep bad data from corrupting good data (bank: no read down, no write up), and the **Chinese Wall** to avoid conflicts of interest (a consultant cannot work for Pepsi and Coca-Cola).

Why is it useful? When an organization must guarantee a property no matter what users do (military secrets, bank records, hospitals), these models give a ready-made, proven design. They also explain deep real-world problems: hidden leaks (*covert channels*), why declassifying documents is hard, and why unchecked user input (*sanitization*) causes SQL injections and buffer overflows.

Mandatory Access Control (MAC)

Access to and operations on resources **are determined by the security policy** (a central security policy assigns permissions):

- the “owner” may not exist, or may not have the power to set permissions against the policy;
- the security policy **must** be enforced despite subjects trying to subvert it.

Security model

A **design pattern** for a specific security property or set of properties. When faced with a standard security problem → use a well-known model!

The devil is in the details

Many aspects are **not covered by the model**: who are the subjects? what are the objects? what mechanisms to use to implement it?

4.1 Bell-La Padula (BLP): protecting confidentiality

Bell-La Padula model (BLP)

Subjects S and objects O are associated to a **level** of confidentiality. Subjects’ access rights are defined (in an access control matrix) by four attributes:

- **Execute** — the subject cannot see or modify the object, but can run it;
- **Read** — the subject can only see the object but cannot modify it;
- **Append** — the subject cannot read the object, but can attach new content;
- **Write** — the subject can see the object and add content or modify existing content.

Observe vs. alter

Observe rights: read (r), write (w). **Alter** rights: append (a), write (w). Write is both observe and alter.

4.1.1 Security levels and dominance

Security level (classification of objects)

Objects are associated to a **security level**: they have a **label** and belong to one or more **categories**.

Security level = (Classification, {set of categories})

- **Classification** — **total order** of labels (e.g. Unclassified < Confidential < Secret < Top Secret).
- **Categories** — compartments of objects with a common topic (e.g. Nuclear, NATO, Crypto).

Dominance relationship

A security level (l_1, c_1) **dominates** (l_2, c_2) **if and only if**

$$l_1 \geq l_2 \quad \text{and} \quad c_2 \subseteq c_1.$$

Exercise — Dominance

Labels: Admin < Nurse < Surgeon < Doctor; categories: DEMOGRAPHICS, ANALYSIS, RESULTS. Which statements are true?

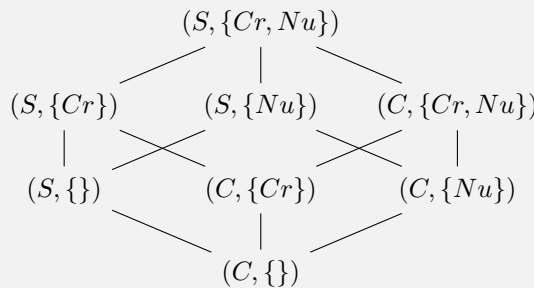
- $(D, \{\})$ dominates $(S, \{\})$
- $(S, \{\})$ dominates $(N, \{\text{RESULTS}\})$
- $(S, \{\text{DEMOGRAPHICS}, \text{RESULTS}\})$ dominates $(N, \{\text{DEMOGRAPHICS}\})$
- $(D, \{\text{ANALYSIS}, \text{RESULTS}\})$ dominates $(S, \{\text{DEMOGRAPHICS}\})$

What level dominates them all? What level dominates only itself?

Dominance lattice

Three key properties:

- dominates is **transitive**;
- there are **top and bottom** elements;
- it is only a **partial** order: two levels may be incomparable (e.g. $(S, \{\text{Crypto}\})$ and $(S, \{\text{Nuclear}\})$).



Labels $C < S$, categories Crypto, Nuclear: a higher node dominates every node reachable downwards.

Clearance of subjects

- **Clearance** — maximum security level a subject has been assigned: $\text{level}(S_i)$ (BLP also calls this “classification”).
- **Current security level** — subjects can operate at lower security levels: $\text{current-level}(S_i)$.

$\text{level}(S_i)$ **must dominate** $\text{current-level}(S_i)$!

4.1.2 The three BLP properties**Simple security property (ss-property) — No Read Up**

If (subject, object, r) is a current access, then $\text{level}(\text{subject})$ **dominates** $\text{level}(\text{object})$.

⇒ **No Read Up (NRU)**: a subject cleared Secret can read Secret, Classified, Unclassified, but not Top Secret.

Why is the ss-property not sufficient?

A Secret subject runs **malicious code** (Trojan) that reads Secret data (allowed by NRU) and **writes it into an Unclassified object**, which an Unclassified subject then reads (also allowed by NRU). Information flowed **down** without any read-up.

Star property (*-property) — No Write Down

If a subject has simultaneous “**observe**” (r, w) access to O_1 and “**alter**” (a, w) access to O_2 , then $\text{level}(O_2)$ **dominates** $\text{level}(O_1)$.

⇒ **No Write Down (NWD)**: a Secret subject can append to Top Secret, read Secret and below, but cannot append/write to lower levels.

Discretionary property (ds-property)

If an access (subject, object, action) takes place, it **must be in the access control matrix**. Information should only be accessed on a “**need-to-know**” basis: MAC needs **DAC** (least privilege inside the security level). Also important to protect integrity.

BLP in the army

- An Unclassified soldier reports “We’ve seen tanks moving!” to a Secret officer: **allowed** (write up).
- The Secret officer reads a Secret document: **allowed** (read at clearance level).
- The Secret officer orders “Attack!” to the Unclassified soldier: **not allowed** (write down, against the *-property).

Basic Security Theorem

If all state transitions are **security preserving**, and the **initial state is secure**, then every subsequent state is secure **regardless of the inputs**.

If, for any individual access, (1) the ss-property, (2) the *-property and (3) the ds-property are preserved, then security holds for any sequential composition: a system can be analyzed in terms of **single-step transitions** of states (not surprising: it is an induction).

4.1.3 Covert channels

What are the transitions, and what does “read” mean?

$\text{level}(s_1) = TS$, $\text{level}(s_2) = C$.

1. s_2 creates o_2 , so $\text{level}(o_2) = C$;
2. s_1 reads C and either raises the object level to TS or leaves it at C ;
3. s_2 attempts to access o_2 : **success or failure leaks 1 bit of information** from TS to C !

Covert channel

Any channel that allows **information flows contrary to the security policy**.

- **Storage channels** — e.g. shared counters, ID fields, file metadata, etc.
- **Timing channels** — e.g. use of CPU, load to memory (cache), queuing time, etc.

Covert channels and least common mechanism

The more resources are shared, the harder it is to eliminate covert channels (principle 7, least common mechanism).

Mitigation: isolation (communication with low level not possible) or adding noise to the communication.

- Hard to achieve less than 1 bit/sec.
- OK for documents, **NOT** OK for cryptographic keys (DoD policy: cryptographic keys must always be stored on dedicated hardware).

4.1.4 Declassification

Declassification

Remove the classification label of an object. It is very typical and necessary (e.g. the officer must eventually be able to tell the soldier “Attack!”), and happens **under the control of the security policy**.

Declassification cannot be made inherently safe

It is a **manual process** (rules about archives, historical records) and it is hard to rule out covert channels: how to know the object does not contain secrets? In practice: Word revision history retains deleted text; PDF “redaction” by overlaying black rectangles leaves the text in the file. **Strategic adversary!**

BLP limitations

- **Confidentiality-oriented:** does not consider integrity or availability.
- **State-based + single transition** model: too low-level, not expressive.
- The 3 security properties are **not sufficient** to ensure confidentiality: changes in clearance and classification can create covert channels, and a static system without changes is impractical.

4.2 BIBA: protecting integrity

Why integrity?

BLP focuses on confidentiality (military / government). For **commercial services** (banking, inventory, stock exchange, land registry, grades database, contracts, payments), preventing fraud is about **protecting integrity**: the adversary has not influenced the result; confidentiality is either secondary or unnecessary.

Integrity is key for computer security in general: the TCB has to have high integrity; public key cryptography requires high integrity for confidentiality.

BIBA model

Subjects and objects have an **integrity level** (from lowest to high integrity). Two operations: **read** (observe) and **write** (modify). Two key (strict) rules:

- **Simple integrity (no read-down)** — protects higher integrity principals from being corrupted by lower integrity level data.
- ***-integrity (no write-up)** — prevents lower integrity principals from corrupting high integrity data.

BIBA is the dual of BLP

BLP: no read *up*, no write *down* (secrets must not flow down).

BIBA: no read *down*, no write *up* (dirty data must not flow up).

BIBA illustrated

Bank: the director can establish a rule and every employee reads it; employees cannot rewrite rules. **Computer**: a web application open in the browser should not write to the file system (at most /tmp).

4.2.1 BIBA variants

Variant 1 — low-water-mark for subjects

Subjects start processes at their **highest** integrity level. When accessing an object, the subject's current level is **lowered to the lowest** of $\text{current-level}(s)$ and $\text{level}(o)$.

Temporary downgrade for the session (e.g. mitigates the impact of a network Trojan: after reading low network data, the subject can no longer write high OS data). Hard to avoid **label creep**.

Variant 2 — low-water-mark for objects

Once an object has been written to by a subject, it assumes the **lowest** level of the object or subject (a high-integrity database written by a process with network access is labelled "low").

Dangerous! It only allows integrity violation **detection**. Mitigation: replicate objects & sanitize / erase.

Invocation

- **Simple invocation** — only allow subjects to invoke subjects with a label **they dominate**. + protects high integrity data from misuse by low integrity principals; – what level is the output?
- **Controlled invocation** — only allow subjects to invoke subjects **that dominate them**. + prevents corruption of high integrity data; – hard to detect polluting information.

4.2.2 Sanitization

Sanitization

Process of taking objects with “**low**” integrity and “**lifting them**” to “**high**” integrity.

Root cause of many vulnerabilities

“Sanitization” problems are the root cause of large classes of real-world security vulnerabilities: malformed “low” (user) input can influence “high” (service) data and code.

- **Web**: web server (high) accepts input from a web client (low) → SQL interpreter → **SQL injection**.
- **OS**: UNIX suid program (high) accepts input from a user (low) → short buffer → **buffer overflow**.

Fundamental principle of sanitization (fail-safe defaults)

Positively verify that “low” objects are within a valid set before elevating their integrity to “high”.

- **Whitelist**: check that all properties of good objects hold (is it in the universe of good things? accept).
- **Do not blacklist**: do not just check for bad objects or properties (the set of bad things is large and undefined).

XSS

Insert a photo in a web album? Ensure the caption is from a restricted set of Unicode, or “escape”/“encode” any character not in that safe set. Do not simply check that it does not contain `<script>`.

4.2.3 Principles to support integrity

Three principles to support integrity

- **Separation of duties** — require **multiple principals** to perform an operation (harder for an adversary: they have to corrupt two principals).
- **Rotation of duties** — allow a principal only a **limited time** on any particular role and limit other actions while in this role (harder for an adversarial insider to tamper with the system).
- **Secure logging** — **tamper-evident log** to recover from integrity failures; consistency of the log across multiple entities is key (harder to make an integrity breach durable).

4.3 Chinese Wall model: conflicts of interest

Inspiration

UK rules about handling **conflicts of interest** in the financial sector: a separation must exist at all times, even within the same firm, between people engaging in activities that conflict with each other. Cost of failure: large fines and reputation.

Chinese Wall model — entities

1. All objects are associated with a **label denoting their origin** (“Pepsi Ltd.”, “Coca-Cola Co.”, “Microsoft Audit”, “Microsoft Investments”).
2. The originators define “**conflict sets**” of labels ({Pepsi, Coca-Cola}, {Microsoft Audit, Microsoft Investments}).
3. Subjects are associated with a **history** of their accesses to objects, and in particular their labels.

Chinese Wall access rule

A subject can access an object (for either read or write) if the access **does not allow an information flow** between items with labels in the **same conflict set**.

Direct flow

Alice accesses Pepsi files (OK), then Microsoft Invest (OK), then tries Coca-Cola: **denied**. She has already accessed Pepsi, so an information flow between Pepsi and Coca-Cola might happen (she could work again with Pepsi).

Indirect flow

Alice is assigned to Pepsi; Bob to Coca-Cola and IBM. Alice tries to access IBM: **denied**. If she writes in IBM with her knowledge of Pepsi, the information *may* flow to Coca-Cola through Bob.

Sanitization is necessary for business

“Un-label” some items as long as the information cannot lead to any conflict of interest, e.g. extract some “general market information”.

Summary

- **Security models:** patterns to design MAC policies.
- **BLP:** confidentiality (no read up, no write down). Key concept: **declassification**.
- **BIBA:** integrity (no read down, no write up). Can bootstrap high confidentiality (PKI) or high availability (replication); can lead to low confidentiality or low availability. Key concept: **sanitization**.
- **Chinese Wall:** conflicts of interest (confidentiality & integrity).
- **Multilateral security:** conflicting properties.

CHAPTER 5

Lecture 5 : Applied cryptography

In a nutshell

What is this lecture about? How Alice can send a message to Bob so that Eve, who sees everything on the wire, **cannot read it**. We start from broken historical ciphers (Caesar, substitution) to learn how attackers think, define the different kinds of attackers, see the only *perfectly* secure cipher (one-time pad) and why it is impractical, then how real systems approximate it (**stream ciphers**), and finally how two strangers can agree on a secret key over a public channel (**Diffie-Hellman**).

Why is it useful? Cryptography protects every HTTPS page, every VPN, every encrypted backup and every messaging app. Understanding its basics tells you *what* a cipher guarantees (and what it does not, e.g. integrity), and why the golden rule is: **never design your own crypto, use vetted standards**.

5.1 Module 1 – Introduction

Why cryptography matters

- **Data in transit** — securing communication (e.g. HTTPS, VPNs).
- **Data at rest** — securing stored information (e.g. encrypted backups).

Cryptography is also the building block for **integrity** (ensuring data has not been tampered with: hashing, signatures), **authentication** (verifying identity: protocols), **anonymity**, etc. This lecture focuses on the basics of using cryptography for **confidentiality**.

Secure communication over an insecure channel

The core problem: we need a way for **Alice** to send a message to **Bob**, so that **Eve** (the adversary) **cannot read it**.

Confidentiality for data at rest can be seen as a particular case, where Alice = Bob (you “send” the data to your future self).

Encryption and decryption

Confidentiality: information cannot be accessed by unauthorized parties.

- **Encryption algorithm** — cryptographic instructions that convert a **plaintext** message into **ciphertext**.
- **Decryption algorithm** — cryptographic instructions that convert a ciphertext message into plaintext.

Both encryption and decryption algorithms require a **key**: encryption cannot be reversed without the key. As functions, with plaintext M (original message), key K (secret value) and ciphertext C (scrambled message):

$$C = E_K(M) \quad \text{and} \quad M = D_K(C).$$

Cryptographic algorithms for confidentiality

1. Get a key k shared between Alice and Bob.
2. Alice encrypts the message: $m \rightarrow \text{Enc}(k, m)$.
3. Alice sends $\text{Enc}(k, m)$ over the insecure channel.
4. Bob decrypts: $\text{Dec}(k, \text{Enc}(k, m)) \rightarrow m$.

Intuition

Think of the algorithm as a **lock design** and the key as **the key**: the lock design is public (open design, Lecture 2 / Kerckhoffs), only the key is secret. Step 1 (sharing the key) is the hard part, and comes back at the end of the lecture.

Invertibility (the core requirement)

The decryption function D_K **must** be the **inverse** of E_K :

$$\forall K, M, \quad D_K(E_K(M)) = M.$$

If not, Bob cannot recover the original message M .

Security requirement: to provide security, these functions must also be **hard to invert without knowing the key K** .

Two requirements that pull in opposite directions

Correctness: *with* the key, inverting is easy. **Security**: *without* the key, inverting is hard. The whole art of cryptography is to get both at the same time.

Hard to invert: the key space and brute force

Key space: the set of all possible keys.

Concrete meaning of “hard to invert”: to find the plaintext M , in an ideal cryptosystem, the only viable attack for Eve must be to **try every single possible secret key K** (a **brute-force attack**).

Goal: a secure scheme forces the attacker into this situation.

Caesar’s cipher

- **Encryption**: shift each letter by a fixed number K .
- **Decryption**: shift each letter by $-K$.
- **Key space**: only 25 possible keys.

Caesar with $K = 3$

ATTACK AT DAWN $\xrightarrow{K=3}$ DWWDFN DW GDZQ ($A \rightarrow D, T \rightarrow W, \dots$).

Security insight: $\log_2(25) \approx 4.6$ bits of security — **far too small** for real-world use (a brute force is trivial). It is still pedagogically useful to introduce a first real attack.

Frequency analysis attack

- Eve knows that Alice wrote a long-enough message for Bob in English.
- She knows that the most frequent letter in English is ‘e’.
- She identifies the most frequent letter in the ciphertext: it likely corresponds to ‘e’.
- She then directly recovers the key (a number between 0 and 25).

Why it works

Caesar maps every ‘e’ to the *same* ciphertext letter, so the ciphertext keeps the **statistics** of English. Eve does not even need to try the 25 keys: one look at the letter counts gives the shift.

Substitution cipher

Encryption/decryption by **permuting the alphabet**: each letter is mapped to a unique, different letter, defined by a **permutation** of the 26 letters (the key).

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 HOWBUGIACRYEVZXPJQMSNTFDKL

Key space: $26! \approx 4.03 \times 10^{26}$ possible keys, i.e. $\log_2(26!) \approx 88.4$ bits of security?

The flaw, again

This key space looks big enough, but the cipher is **still easily broken** by the same frequency analysis. **Vulnerability:** the cipher **preserves the statistical properties** of the plaintext language (e.g. single-letter frequencies). With a long enough ciphertext, statistical analysis breaks the cipher, rendering the full $26!$ key space **irrelevant**.

E.g. on the whole Harry Potter series encrypted with one key, Eve guesses the most likely letter mappings, uses a dictionary to rule out keys that give non-English text, and finds K in a few tries.

Goal of the adversary

Eve does not necessarily need the *full* key: depending on the scenario, recovering part of it (the mapping of some letters) may be enough. Sometimes the goal is even smaller: Eve may only want to know whether Alice sent “Yes” or “No”.

Lesson

A **big key space is necessary but not sufficient**. Security is about the *best* attack, not only brute force.

Bits of security

Goal: an N -bit key should offer security as close to N bits as possible, i.e. require 2^N attempts.

The break: if a 1024-bit key is broken with only 2^{40} attempts (40 bits of security), the algorithm is considered **broken**.

Caesar lesson: the 25-key space was cleverly reduced to **1 possibility**; similarly the ≈ 88 bits of the substitution cipher were reduced to a few tries.

Orders of magnitude

Each extra bit **doubles** the work. $2^{40} \approx 10^{12}$ operations is easy for a laptop; 2^{128} is far beyond all computers on Earth for billions of years. “Broken” does not mean “practically decrypted today”, it means **faster than brute force**.

5.2 Module 2 – Adversaries in cryptography

Back to threat models

As in Lecture 1, security is always measured **relative to the power of the adversary**. Against Caesar and substitution, Eve’s capabilities were very limited: she observed a long-enough ciphertext of English text (and had some compute power to test a few keys). This is not the only possible adversary model.

Types of adversaries

- **Passive eavesdropper** — Eve can only *read* the ciphertext.
- **Active attacker** — Eve can somehow **influence the system**, e.g. convince Alice to send Bob specific messages, observe the precise time Bob takes to run his algorithm, etc.: anything that goes beyond just observing communication.

Known Plaintext Attack (KPA)

Scenario: Eve is given access to multiple pairs (M, C) of messages and their corresponding ciphertexts, all encrypted with the secret key K .

Goal: from these pairs, guess the key K to decrypt other messages.

Why is KPA a realistic model?

- **Standard headers** — many encrypted messages contain predictable parts (“From:”, timestamps, packet destinations, ...).
- **Malicious guessing** — if Eve guesses a message (or part of it) and observes its encryption, she has an (M, C) pair.

KPA and the substitution cipher

Under KPA the substitution cipher becomes **even more broken**: if Eve knows that $M = \text{THE QUICK BROWN FOX}$, she immediately recovers the mapping of many letters, drastically reducing the key space without any statistical analysis.

Chosen Plaintext Attack (CPA)

Scenario: Eve can convince Alice (or the system) to encrypt **arbitrary messages chosen by Eve** with the secret key.

More abstractly: for any chosen plaintext m , Eve gets $E_K(m)$. We say that Eve has access to an **encryption oracle**.

KPA vs. CPA

In KPA, Eve does not choose which pairs she sees; in CPA she *chooses* m and gets $(m, \text{Enc}(k, m))$. Hence CPA is a **strictly stronger** model than KPA:

$$\text{passive (ciphertext only)} < \text{KPA} < \text{CPA}.$$

A scheme secure against a stronger adversary is also secure against the weaker ones.

Why is CPA a realistic model?

- **Messaging app** — with a few minutes of access to Alice’s phone (Signal, WhatsApp), Eve can send *any* data she chooses to Bob and obtain an (M, C) pair.
- **Encryption API** — Eve interacts with an endpoint where she provides data and receives the encrypted output.

CPA and the substitution cipher

The substitution cipher is **completely broken** by a CPA: Eve asks for the encryption of $M = \text{abcdefghijklmnopqrstuvwxyz}$; the ciphertext *is* the entire key (the full permutation), in just **one** query!

Side-channel attacks

Eve not only accesses an ideal oracle, but also **measures physical characteristics of the actual implementation** while it encrypts/decrypts.

- **Timing attack** — Eve measures **how long** it takes to encrypt a given message or decrypt a ciphertext.
- **Power analysis** — Eve observes the **energy consumed** by the device (e.g. a credit card) doing the crypto.

Why are side channels realistic?

- **Malicious ownership** — you often own a device (phone, smart card, embedded chip) holding keys of *another* principal (Google, your bank) that are meant to be unreadable to you.
- **DRM** — the decryption key of a movie/song is embedded in your media player or console; by measuring power or timing, an attacker could extract it.

Algorithm vs. implementation

A mathematically perfect cipher can still leak through its **implementation** (e.g. code that takes longer when a key bit is 1). Side channels attack the code and the hardware, not the math — a reminder that security is about the whole system. As always, new adversary models can be defined, but always as a **strategic adversary with limited, well-defined resources**.

5.3 Module 3 – Perfect secrecy and the one-time pad**Reminder: XOR ($\oplus$)**

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Key properties: $a \oplus a = 0$, $a \oplus 0 = a$, hence $a \oplus b \oplus b = a$ (XOR-ing twice with the same value cancels out). XOR with a random bit flips the input bit with probability 1/2: the output is itself random.

One-Time Pad (OTP)

How to remove frequency analysis? Make sure that letters are **not** encrypted to the same values.

Key: a string k of **random** bits, **as long as the message**, pre-shared between Alice and Bob.

$$\text{Enc}(k, m) = m \oplus k \qquad \text{Dec}(k, c) = c \oplus k$$

Decryption works because $(m \oplus k) \oplus k = m$.

OTP on “YEAH”

Message (ASCII 59454148)	01011001010001010100000101001000
$\oplus$ OTP key	01110101000111010100101001001010
Ciphertext	00101100010110000000101100000010

XOR-ing the ciphertext with the same key gives back the message. Encrypting “YEAH” twice with *different* keys gives *different* ciphertexts, and “YEAH” and “NOPE” can produce any ciphertext depending on the key: the ciphertext alone tells nothing.

Perfect secrecy

For any ciphertext C , **every possible plaintext M is equally likely** as before seeing C : the ciphertext gives the adversary **zero new information** about M . Formally:

$$\forall m, c, \quad \mathbb{P}(M = m \mid E_K(M) = c) = \mathbb{P}(M = m).$$

The OTP is perfectly secure

Messages m have some prior probability $\mathbb{P}(M = m)$ (“Yes” may be more likely than “Auto-bus”), and the key K is uniform on n -bit strings, independent of M . Since $E_K(m) = c \iff K = m \oplus c$:

$$\begin{aligned} \mathbb{P}(M = m \mid C = c) &= \frac{\mathbb{P}(M = m \cap K = m \oplus c)}{\mathbb{P}(C = c)} = \frac{\mathbb{P}(M = m) \mathbb{P}(K = m \oplus c)}{\mathbb{P}(C = c)} \\ &= \frac{\mathbb{P}(M = m) 2^{-n}}{2^{-n}} = \mathbb{P}(M = m), \end{aligned}$$

using independence of M and K , and $\mathbb{P}(C = c) = \sum_{m'} \mathbb{P}(M = m') \mathbb{P}(K = m' \oplus c) = 2^{-n}$.

Intuition

For a given ciphertext c , *every* message m is possible: there is exactly one key ($k = m \oplus c$) that maps it to c , and all keys are equally likely. So even with infinite computing power, Eve cannot tell which message was sent.

One-time: never reuse the key

One-time pads are called like this because they must **not** be reused. If the same key encrypts two messages:

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2,$$

the key cancels out. Eve cannot recover the full messages directly, but she learns information about the plaintexts, which can feed frequency analysis and help recover them.

Spaces vs. letters in ASCII

In ASCII, spaces start with 00 and letters with 01. In $M_1 \oplus M_2$:

- 2 letters: $01 \oplus 01 = 00$;
- 2 spaces: $00 \oplus 00 = 00$;
- letter + space: $01 \oplus 00 = 01 \Rightarrow$ one of the messages has a space at this position!

Exercise — Key reuse

Suppose Alice reuses the key K to encrypt **two single-word English messages of length 5**, M_1 and M_2 , producing $C_1 = M_1 \oplus K$ and $C_2 = M_2 \oplus K$.

Task: show that Eve can “likely” recover the key and then the two messages M_1 and M_2 . (Hint: Eve can look up an English dictionary of all the words of length 5.)

The key problem of the OTP

Practical limitation: OTP requires keys that are **truly random, message-length, and used only once**.

The showstopper: securely pre-distributing message-length keys for real-world applications is **impractical**.

Why is it a showstopper?

To send 1 GB securely you first need to share 1 GB of secret randomness *securely* — but if you had a secure way to share 1 GB, you could just send the message that way! And reusing keys is not an option (see above).

OTP: integrity is NOT included (malleability)

The OTP guarantees **confidentiality** (nobody can read it), but it **does not** guarantee **integrity** (nobody can tamper with it).

Bit-flipping attack

Eve flips bits of C with a mask Δ (e.g. only the first bit set): $C' = C \oplus \Delta$. Bob decrypts

$$M' = C' \oplus K = (M \oplus K \oplus \Delta) \oplus K = M \oplus \Delta.$$

Eve successfully changed the message (M' is M with one bit flipped) **without knowing K** . If Eve knows the format (e.g. “PAY 0100 CHF”), she can change the amount in a controlled way.

Confidentiality $\neq$ integrity

Encryption hides the content; it does **not** prove that the content was not modified. Integrity needs separate mechanisms (MACs, signatures — later in the course).

5.4 Module 4 – From OTP to stream ciphers

Symmetric encryption

Encryption of plaintext and decryption of ciphertext are done using **the same key**. Two main types:

- **Block ciphers** — operate on fixed-size blocks (e.g. 128 bits). Example: **AES** (Advanced Encryption Standard).
- **Stream ciphers** — operate one bit/byte at a time, like a **pseudo-OTP**. Example: **ChaCha20**.

Symmetric cryptographic key

A **fixed-size** input to symmetric cryptographic primitives; the size of the key influences the level of security provided. Key properties:

- **Known to both parties** — partners must agree on the key **before** starting to use the primitive (requires secure generation and sharing protocols).
- **It is reused** — the key is pre-shared once and then reused (keys do have a “duration”).
- **It must be secret** — revealing the key eliminates any protection provided by the primitive.

Difference with the OTP

OTP key: as long as the message, used once. Symmetric key: **short and fixed** (e.g. 128 or 256 bits) and **reused** for many messages. The next definitions show how to get OTP-like encryption from such a short key.

Stream cipher (pseudo-OTP)

A stream cipher attempts to emulate the OTP’s security while solving the key-length problem:

1. **Short key K** — Alice and Bob share a small, manageable key (kept secret).
2. **Key Stream Generator (KSG)** — uses K and an **Initialization Vector (IV)** (which does **not** need to be secret) to produce an arbitrarily long, *pseudo-random* bit stream $S = \text{stream}(K, IV)$.
3. **Encryption** — $C = M \oplus S$.

To **decrypt**, the receiver feeds the KSG with the shared key and the IV (sent with the ciphertext) to recreate the same stream S , then computes $C \oplus S = M$.

Security of the KSG

We generate potentially **infinitely long randomness** (the stream S) from a **finite secret** (K). Security depends entirely on S being **computationally indistinguishable from a truly random OTP key**: unless one knows the key, one cannot distinguish the stream from a random string.

Why does the KSG also take an IV?

Without an IV, the same key would always produce the **same stream S** , and two messages would be encrypted with the same “pad”: XOR-ing the two ciphertexts recovers $M_1 \oplus M_2$ (the two-time pad above), and Eve could recover the prefix of S and decode future messages. A fresh IV per message gives a **different stream for every message** while reusing the same key.

Stream ciphers: strengths and weaknesses

Speed (+)	Algorithms are linear in time and constant in space.
Low error propagation (+)	Errors in one bit do not affect subsequent symbols.
Low diffusion (–)	A change in one plaintext bit only affects one ciphertext bit.
Susceptibility to modification (–)	Low diffusion makes it easier to tamper with the message (like the OTP).

Diffusion

Diffusion = how much a small change in the input spreads to the output. Stream ciphers have none (bit i of C only depends on bit i of M), which is exactly why the bit-flipping attack works on them.

Periodicity of key stream generators

Key stream generators which have a **finite state** are eventually **periodic**: if the message is longer than the period, the stream **repeats**.

The attack: if the period is identified and too small, the key stream is known for the entire message.

Why finite state implies periodic

A KSG with L bits of internal state has at most 2^L different states. Each output step moves deterministically to a next state, so after at most 2^L steps a state repeats, and from then on everything repeats: a repeating stream is a **reused pad**.

Interlude on building KSGs

The next definitions illustrate design patterns for KSGs and the challenges of building secure ones; you are not expected to master the underlying math.

Linear Feedback Shift Register (LFSR)

A **linear recurrence relation**: the next output bit is a linear function (XOR) of previous bits. It extends an initial random sequence of bits into an arbitrary-length sequence of (maybe?) random-looking bits.

Advantage: uses only **shift registers** (state storage) and **XOR gates** (feedback), so it is extremely fast and efficient in hardware.

LFSR example

Initial state of 4 bits $a_0, \dots, a_3 = 1, 0, 0, 0$ and $a_n = a_{n-3} \oplus a_{n-4}$:

$$a_4 = a_1 \oplus a_0 = 1, \quad a_5 = a_2 \oplus a_1 = 0, \quad a_6 = a_3 \oplus a_2 = 0, \quad a_7 = a_4 \oplus a_3 = 1, \dots$$

Output: 1, 0, 0, 0, 1, 0, 0, 1, 1, ...

Randomness of LFSRs

- **Maximal period** — if the characteristic polynomial of the recurrence is *primitive*, an L -bit register reaches the maximum period of $2^L - 1$ states before repeating (all non-zero states).
- **Distribution property** — every non-zero state appears exactly once in the cycle, so the output is highly balanced: over one period it contains exactly 2^{L-1} ones and $2^{L-1} - 1$ zeros.

LFSR — math danger

Despite passing some statistical tests, an LFSR is purely **linear**. Techniques like the Berlekamp-Massey algorithm (beyond the scope of the class) recover the entire LFSR state and key stream from only a **short segment of the output**, breaking the cipher. **LFSRs should not be used directly as key stream generators**, though they have traditionally been used as a building block of KSGs.

Why linearity is fatal

Linear means “solvable with linear algebra”: each output bit gives one linear equation on the unknown initial state. With about $2L$ output bits (easy to get in a KPA), Eve solves the system and predicts the whole stream. “Looks random” (statistics) is not the same as “unpredictable” (security).

A5/1 and modern stream ciphers

A5/1, used historically to secure GSM mobile phone communications, combined **multiple LFSRs** in a non-linear way to overcome the linearity flaw while staying cheap. Despite this, mathematical weaknesses allow breaking it much faster than brute force: it is practically broken (recall the closed-design failure of GSM, Lecture 2).

There are modern stream ciphers with **no known practical cryptanalysis**, e.g. **Trivium** (based on NLFSRs, 3 shift registers glued with XOR and AND gates) and **Salsa20** (using different techniques).

Bit security and consequences

- **Cost of breaking** — exploiting a mathematical property or bug can break the scheme much faster than $2^{|K|}$ attempts, **reducing the bit security**.
- **The difficulty** — designing mathematically secure and subtly resistant primitives is incredibly hard.
- **Conclusion** — **never design your own** stream cipher (block cipher, or hash function).
- **Best practice** — use **well-vetted, peer-reviewed, standard algorithms and their implementations** (AES, ChaCha20, SHA-3).

5.5 Module 5 – Public-key cryptography**Key distribution problem**

Symmetric ciphers are fast, but require a **pre-shared secret key**. How can Alice and Bob agree on a secret key if **all their communication is observed by Eve**?

Public-key (asymmetric) cryptography

The paradox: before the 1970s, it was considered impossible to communicate securely without a pre-shared secret.

The revelation: public-key cryptography allows two parties who have **never met** to establish a secure channel.

Diffie-Hellman (DH) key exchange

Alice, Bob (and Eve) know large **public parameters**: a prime number p and a generator g .

- **Alice** chooses a **private** secret a , computes the **public value** $A = g^a \bmod p$ and sends A to Bob.
- **Bob** chooses a **private** secret b , computes the **public value** $B = g^b \bmod p$ and sends B to Alice.

Both compute the same **shared secret**:

$$K = B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \pmod{p}.$$

Alice computes B^a (she has a), Bob computes A^b (he has b). Eve sees A , B , g and p , but cannot find a or b (discrete logarithm problem), so she cannot compute K .

Toy example (real p has 2048+ bits)

$p = 23$, $g = 5$. Alice picks $a = 4$: $A = 5^4 \bmod 23 = 625 \bmod 23 = 4$. Bob picks $b = 3$: $B = 5^3 \bmod 23 = 125 \bmod 23 = 10$.

Alice: $K = 10^4 \bmod 23 = 18$. Bob: $K = 4^3 \bmod 23 = 64 \bmod 23 = 18$. Same key! Eve sees 23, 5, 4, 10 but must find a with $5^a \equiv 4 \pmod{23}$.

Paint analogy

Mix a public colour with your secret colour and send the mix; each side adds its own secret colour to what it receives: both end up with the same final colour. Un-mixing paint (finding the secret colour) is hard, like the discrete logarithm.

Trapdoor functions

DH is one of many ways to use public-key cryptography to establish a shared secret (a shared key) to then do symmetric cryptography. They boil down to mathematical functions that are **easy to compute in one direction** but **hard (high complexity) to compute in the other direction**.

Discrete Logarithm Problem (DLP)

Underlying difficulty of DH: computing $g^a \bmod p$ is cheap (fast modular exponentiation), but there is **no known fast (polynomial) algorithm** to recover a from $A = g^a \bmod p$ (and g , p).

Beyond DH

Other ways to set a shared secret (or even to directly communicate), all involving interesting mathematics:

- **RSA** — based on the **factoring** problem.
- **Elliptic Curve Cryptography (ECC)** — the discrete logarithm problem on elliptic curves instead of modulo p (smaller keys for the same security).
- **Post-quantum cryptography** — new primitives (like **lattice crypto**) to resist quantum computers.

Man-In-The-Middle (MITM) attack

Vanilla DH is vulnerable to an **active** attack: DH only guarantees key *agreement*, not *authenticity* (verifying *who* you are agreeing with).

1. Eve intercepts A from Alice and B from Bob.
2. Eve sends her own value E_B to Alice: Alice computes K_{AE} .
3. Eve sends her own value E_A to Bob: Bob computes K_{BE} .

Eve shares a key K_{AE} with Alice and a separate key K_{BE} with Bob: she can **read, modify, and relay** all communication.

Why it works

Alice believes E_B comes from Bob, but nothing in the protocol proves it. Each side does a perfectly correct DH — just with Eve. DH is secure against a *passive* eavesdropper, not against an *active* attacker (Module 2).

Identity verification problem

DH solves **key distribution**, but creates the **identity verification problem**.

- **Solution:** DH must be paired with a mechanism (like **digital signatures**) to verify identity (next week).
- **Setup:** parties still need to trust a third party (a **Certificate Authority**) for public key ownership.

Summary

- Encryption $C = E_K(M)$ must be **invertible with the key** and **hard to invert without it**; ideally the best attack is brute force.
- A big key space is not enough: **frequency analysis** breaks Caesar and substitution ciphers. Security is measured in **bits of security**.
- Adversary models: **passive** < **KPA** < **CPA**, plus **side channels** on implementations.
- The **one-time pad** is **perfectly secret** but needs a random, message-length, single-use key, and is **malleable** (no integrity).
- **Stream ciphers** expand a short key + IV into a pseudo-random stream; never reuse a stream, never use plain LFSRs, **never design your own crypto**.
- **Diffie-Hellman** solves key distribution thanks to the **discrete logarithm problem**, but needs **authentication** against MITM.