

Subject Name: Information System

Unit No:03

Unit Name: Security Policies

Faculty: Mrs. Bhavana Alte

Mr. Prathmesh Gunjgur

Unit No: 3

Unit Name: Security Policies Models

Integrity Policies



Introduction

Integrity ensures that data and systems remain accurate, reliable, and unaltered by unauthorized users.

- **Commercial vs. Military Integrity:** Commercial systems focus on **data integrity** and preventing corruption or unauthorized modifications.
- **Military systems** focus on **data confidentiality** (who can access classified information).



Commercial Integrity Goals (Lipner's Five Requirements)

- **Lipner (571) identified five key integrity requirements for commercial systems:**
- **1. Users do not write their own programs**
- ✓ **Explanation:** Employees must use pre-approved software and databases to ensure security.
 ✦ **Example:** A financial analyst cannot install unauthorized spreadsheet software; they must use the company-approved tool.
- **2. Programmers test software on non-production systems**
- ✓ **Explanation:** Developers must use a separate test environment instead of real business systems.
 ✦ **Example:** A bank does not allow software developers to test new code directly on live customer accounts.
- **3. Controlled installation of new software**
- ✓ **Explanation:** A **strict process** is followed when moving software from testing to live use.
 ✦ **Example:** An IT manager must **approve and document** any software update before installing it in the company's production system.



Commercial Integrity Goals (Lipner's Five Requirements)

- **4. Auditing the installation process**
- ✓ **Explanation:** Every software update must be **tracked and audited** to prevent unauthorized changes.
 - ◆ **Example:** If an employee installs unauthorized software, the audit logs help **identify who did it and when.**
- **5. Managers and auditors must have system access**
- ✓ **Explanation:** Managers and auditors need visibility into system logs and configurations to ensure compliance.
 - ◆ **Example:** A company's security team regularly **reviews system logs** to detect suspicious activities.



Principles Derived from These Requirements

- ◆ **Separation of Duty**
- **Definition:** Critical tasks should require at least **two people** to prevent fraud/errors.
✓ **Example:** In banking, one employee initiates a large money transfer, and another approves it.
- ◆ **Separation of Function**
- **Definition:** Developers should not handle live (production) data.
✓ **Example:** A software engineer at a hospital tests new features **without** real patient data.
- ◆ **Auditing & Accountability**
- **Definition:** System logs track **who did what** to detect errors or fraud.
✓ **Example:** A company detects an employee altering financial reports **by reviewing audit logs.**



Military vs. Commercial Security Approaches

- **Military Systems (Bell-LaPadula Model)**
 - Focuses on **confidentiality** (who can read classified information).
 - Uses **security levels** (Top Secret, Secret, Confidential, etc.).
 - **Example:** Only authorized personnel can view military intelligence.
- **◆ Commercial Systems (Integrity Models like Biba)**
 - Focuses on **data accuracy and trustworthiness** (who can modify data).
 - Uses **integrity levels** (Low-Trust to High-Trust).
 - **Example:** A financial database prevents junior employees from modifying accounting records.



Key Challenge: Information Aggregation

- Even if small bits of **public information** are released, someone might piece them together to infer **sensitive data**.
 - ✓ **Example:** A company keeps employee salaries private, but if expense reports are public, one might deduce salaries from them.



Biba Integrity Model

- **What is the Biba Model?**
- Developed by **Biba in 1977** to ensure **data integrity**.
- Defines **Integrity Levels** (Low to High) for users and objects.
- **Prevents unauthorized modification** by enforcing rules on how data flows.



Biba's Rules

Rule	Description	Example
1. No Read Down ("Write Up, Read Same or Higher")	A subject (user/program) cannot read data from a lower integrity level.	A high-trust financial system cannot use data from an unverified website.
2. No Write Up ("Write Same or Lower, Read Up")	A subject (user/program) cannot modify data at a higher integrity level.	A junior employee cannot edit financial reports, but can view them.
3. Execution Control	A program can only execute another program of the same or lower trust level .	A secure government system cannot run unverified software .



Comparison with Bell-LaPadula:

- **Bell-LaPadula** protects **confidentiality** (who can see data).
- **Biba** protects **integrity** (who can change data).
- **Example:** A **military document** needs Bell-LaPadula to **restrict viewing**, while a **financial record** needs Biba to **prevent modifications**.



Biba Model in Action: Example Case Study (Pozzo & Gray)

- **Goal**
- Prevent untrusted software from altering **data or critical systems**.
- **◆ Implementation in the LOCUS OS**
- **Trust Levels for Software**
 - Software is assigned a **credibility rating** (0 = Untrusted, N = Highly Trusted).
 - Only trusted programs **can modify critical data**.
- **User Risk Levels**
 - Users start at the **highest trust level** they are authorized for.
 - If they run untrusted software, they **acknowledge the risk**.
- **✓ Example:**

A government worker using **classified systems** can only install pre-approved software. To run **unverified software**, they must confirm they understand the security risk.



Clark-Wilson Integrity Model

- Developed in **1987** by **David Clark and David Wilson**.
 - Unlike previous models (like Biba or Bell-LaPadula), which focused on data classification, this model is **transaction-based**.
 - It is **realistic** for commercial systems where data integrity is critical (e.g., banking, finance).
 - **Clark-Wilson Integrity Model** is used to ensure **data integrity** in secure systems, such as **banking systems, financial databases, and enterprise applications**. It defines **how data is handled and modified** using **Certification Rules (CR)** and **Enforcement Rules (ER)**.
-
- **Key Objectives**
 - Ensure **data integrity** (data remains accurate and consistent).
 - Prevent **unauthorized modification** or corruption of data.
 - Implement **separation of duties** to reduce fraud risks.



How the Model Works

- **A. Key Data Classifications**
- **Constrained Data Items (CDIs)** – Critical data requiring strict integrity controls (e.g., bank account balances).
- **Unconstrained Data Items (UDIs)** – Less critical data, does not require strict controls (e.g., customer-selected gifts).
- **B. Types of Procedures**
- **Integrity Verification Procedures (IVPs)**
 - Ensure CDIs meet integrity constraints (e.g., checking that all accounts balance).
- **Transformation Procedures (TPs)**
 - Well-defined transactions that move the system from one valid state to another (e.g., transferring money between accounts).



-
- These rules ensure that **data remains valid** and that **transactions do not introduce corruption**.
 - **CR1: Initial Valid State**
 - **Rule:** Every **Constrained Data Item (CDI)** (trusted data) must start in a **valid** state.
✓ **Example:** A bank account cannot start with a negative balance.
 - **CR2: Valid Transactions**
 - **Rule:** Every **Transformation Procedure (TP)** (process that modifies data) must keep CDIs in a valid state.
✓ **Example:** A **fund transfer** process should always balance debit and credit, preventing accidental losses.



Certification Rules (CR) – Ensuring Data & Transaction Integrity

- **CR3: Separation of Duties**
 - **Rule:** Different users must handle different parts of a transaction to prevent fraud.
 - ✓ **Example:** In a financial system, the **person approving a transaction cannot be the same person executing it.**
- **CR4: Audit Logging**
 - **Rule:** Every transaction must be **logged** to allow audits and detect fraud.
 - ✓ **Example:** Every bank transaction is recorded, showing **who transferred how much to whom.**
- **CR5: Handling Untrusted Data (UDI to CDI Conversion)**
 - **Rule:** Any **untrusted data (UDI)** must be either **rejected** or **validated before entering the system.**
 - ✓ **Example:** A **bank deposit slip** must be **verified** before adding the amount to a customer's account.



Enforcement Rules (ER) – Controlling Access & Execution

- These rules **restrict access** to data and ensure **only authorized users** perform transactions.
- **ER1: Certified Transactions Only**
 - **Rule:** Only **certified TPs** can modify **CDIs**.
 - ✓ **Example:** Only a **verified payroll system** can update employee salaries.
- **ER2: User Authorization**
 - **Rule:** Users can only perform transactions they are **authorized** for.
 - ✓ **Example:** A **cashier can process payments** but **cannot approve refunds**.
- **ER3: User Authentication**
 - **Rule:** Users must be **authenticated** before executing any transaction.
 - ✓ **Example:** A bank teller must **log in** before accessing customer accounts.
- **ER4: Separation of Certifiers & Executors**
 - **Rule:** People who **approve transactions (certifiers)** cannot execute them.
 - ✓ **Example:** A **manager approves a large bank transfer**, but the **accountant processes it**.



Comparison with Other Models

Feature	Biba Model	Clark-Wilson Model
Integrity Focus	Prevents lower-integrity data from corrupting higher-integrity data.	Ensures only authorized transactions modify critical data.
Data Classification	Multiple integrity levels.	CDIs (high integrity) and UDIs (low integrity).
Verification	Trust in "trusted subjects."	Certification of TPs & IVPs ensures correctness.
Separation of Duty	Not explicitly enforced.	Strictly enforced.
Logging & Auditing	Not a key component.	Mandatory logging of all transactions.



Unit No: 3

Unit Name: Security Policies Models

Hybrid Policies



Hybrid policies

- **Hybrid policies** combine **two or more security models** to provide **better protection** in complex systems.
- Since no **single security model** can cover all scenarios, organizations often **merge policies** to balance **confidentiality, integrity, and availability**.
- **Why Use Hybrid Policies?**
 - Different security models have different strengths:
 - **Bell-LaPadula Model** (Confidentiality) → Prevents unauthorized reading (**no read up, no write down**).
 - **Biba Model** (Integrity) → Prevents data corruption (**no write up, no read down**).
 - **Clark-Wilson Model** (Integrity & Audit) → Ensures data is modified **only by trusted processes**.
 - **RBAC (Role-Based Access Control)** → Grants access based on **user roles**.



Common Security Models Used in Hybrid Approach

- 1 **Bell-LaPadula Model (Confidentiality)**
- ✦ **Focus:** Prevent unauthorized users from accessing confidential data.
 - 💡 **Rules:**
 - ✓ **No Read Up (NRU)** – A user **cannot read** data at a higher security level.
 - ✓ **No Write Down (NWD)** – A user **cannot write** data to a lower security level.
 - ⚡ **Used in:** Military, Government, Intelligence Agencies.
- ⚡ **Example:**

A **military officer** with "Secret" clearance **cannot access** "Top Secret" files.
- 2 **Biba Model (Integrity)**
- ✦ **Focus:** Prevent unauthorized modifications to critical data.
 - 💡 **Rules:**
 - ✓ **No Write Up (NWU)** – A lower-level user **cannot modify** higher-level data.
 - ✓ **No Read Down (NRD)** – A higher-level user **cannot read** lower-level data.
 - ⚡ **Used in:** Banking, Healthcare, Financial Systems.
- ⚡ **Example:**

A **junior accountant** cannot modify company financial records, ensuring only **authorized personnel** can update critical files.



- **3□ Clark-Wilson Model (Integrity & Authentication)**

- ✦ **Focus:** Ensures that data is changed **only through secure and verified processes**.

- **💡 Rules:**

- ✓ Data can only be modified by **trusted programs**.
- ✓ **Separation of duties** is enforced (different users handle different steps).
- ✓ All transactions must be **logged** for audits.
- ◆ **Used in:** Banks, E-commerce, Payment Systems.

- ◆ **Example:**

In a **bank**, tellers can **input transactions**, but only a **manager** can approve large transfers.

- **4□ Role-Based Access Control (RBAC)**

- ✦ **Focus:** Limits access to users based on their **job roles**.

- **💡 Rules:**

- ✓ Users **only access what they need** based on their role.
- ✓ Reduces the risk of unauthorized access.
- ◆ **Used in:** Corporate Networks, Hospitals, Cloud Computing.

- ◆ **Example:**

A **nurse** can view **patient medical records**, but **only doctors** can prescribe medication.



How Hybrid Models Work – Real-World Examples

- **Example 1: Military Security System (Bell-LaPadula + Biba)**
- ✦ **Challenge:** The military must protect **both confidentiality & integrity**.
- ✓ **Solution:**
- **Bell-LaPadula** prevents lower-clearance users from accessing secret documents.
- **Biba** ensures that only high-ranking officers can modify critical files.

- **Example 2: Banking System (Clark-Wilson + RBAC)**
- ✦ **Challenge:** Preventing fraud while allowing efficient banking operations.
- ✓ **Solution:**
- **Clark-Wilson** ensures transactions are processed **only through secure programs**.
- **RBAC** restricts access: tellers can check balances, but only managers approve loans.



How Hybrid Models Work – Real-World Examples

- **Example 3: Cloud Computing (DAC + MAC + RBAC)**
- ✦ **Challenge:** Businesses need **both flexibility & strict access control**.
- ✓ **Solution:**
- **DAC** allows employees to share work files.
- **MAC** protects sensitive company data from unauthorized changes.
- **RBAC** ensures employees can only access information **based on their role**.



Chinese Wall Model

- The **Chinese Wall Model** is a **security model** designed to prevent **conflicts of interest** in industries where employees or organizations deal with **sensitive information from competing companies**. It ensures that a person who has accessed **confidential data** from one company cannot access similar data from its competitor.
- ✓ **Key Purpose:** Prevent information leakage and insider trading.
- ⚡ **Also Known As:** Brewer-Nash Model
⚡ **Used In:** Financial institutions, law firms, consulting firms, investment banking, auditing firms, stock markets, etc.





How Does the Chinese Wall Model Work?


- The model divides information into **conflict-of-interest classes** to ensure separation.
- ♦ **Three Key Concepts:**
- 1 **Objects (Data):** The sensitive files or information that must be protected.
- 2 □ **Company Datasets:** Information about different companies grouped into "Conflict of Interest (COI) classes."
- 3 □ **Access Control Rules:** Users can only access **one company's data** from a COI class.



Three Key Concepts of the Chinese Wall Model

- 1  **Objects (Data):**
 - These are the sensitive files, documents, or records that need to be **protected** from unauthorized access.
 - Example: A financial report of **Company A** in the telecom sector.
- 2  **Company Datasets:**
 - Companies in the same industry are grouped into **Conflict of Interest (COI) classes**.
 - If a person accesses **one company's data**, they cannot access data from other companies in the same **COI class**.
 - Example:
 - **COI Class: Telecom Industry**
 - Company A
 - Company B
 - Company C
 - **COI Class: Banking Industry**
 - Company X
 - Company Y
 - Company Z



-
- 3  Access Control Rules:
 - Users can only access data from **ONE** company in a COI class.
 - If they access **Company A's** data, they **CANNOT** access **Company B or C's** data.
 - However, they **CAN still access** data from another industry (e.g., banking sector)



Main Rules of the Model:

- **Rule 1: No Read Conflict**
 - A user **cannot read** data from a competitor if they have already accessed a company's data in the same industry.
- **✓ Rule 2: No Write Conflict**
 - A user **cannot write** information for a company if they have read access to a competitor's data.
- **✓ Rule 3: Access to Unrelated Data Allowed**
 - Users can still access data from **other industries** where no conflict of interest exists.



◆ Main Rules of the Model

- **Rule 1: No Read Conflict**
- A user **CANNOT read** data from **two competing companies** in the same COI class.
- **Example:**
 - If a financial analyst **reads** confidential data from **Company A (Telecom)**
 - They **CANNOT read** data from **Company B (a competitor in Telecom)**.
 - But they **CAN read** data from a company in the Banking industry (Company X).



◆ Main Rules of the Model

- **Rule 2: No Write Conflict**
- A user **CANNOT write (modify or create)** data for a company if they have read access to a competitor's data.
- **Example:**
 - If a consultant **reads** financial data from **Company A (Telecom)**
 - They **CANNOT write** a report for **Company B (a competitor in Telecom)** because it creates a **conflict of interest**.
 - But they **CAN write** for a company in a different industry, like a **Banking company (Company X)**.



◆ Main Rules of the Model

- **Rule 3: Access to Unrelated Data Allowed**
- A user **CAN access data from other industries** where no conflict of interest exists.
- **Example:**
 - If an auditor accesses **Company A's data (Telecom)**,
 - They **CANNOT access** Company B's data (another Telecom company),
 - But they **CAN access** data from **Company X (Banking)** because it's an unrelated industry.



Chinese Wall Model

- Introduction
- The **Chinese Wall Model** is a security model designed to prevent **conflicts of interest** in industries such as **investment firms, banks, and law firms**.
- It ensures that an individual **cannot access sensitive data from competing companies** that belong to the same conflict of interest (COI) class.
- **Example Scenario:**
 - ⚡ **Anthony** is an investment analyst.
 - ⚡ He advises **Bank of America**.
 - ⚡ To **prevent a conflict of interest**, he **cannot** advise **Citibank** because it is a competing bank.



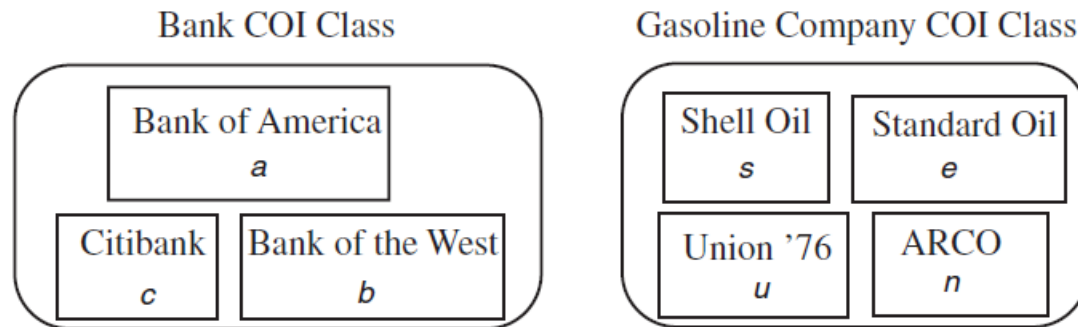


Figure 7–1 The Chinese Wall model database. It has two COI classes. The one for banks contains three CDs. The other one, for gasoline companies, contains four CDs. Each (COI, CD) pair is represented by a lowercase letter (for example, (Bank COI, Citibank) is *c*). Susan may have access to no more than one CD in each COI, so she could access Citibank's CD and ARCO's CD, but not Citibank's CD and Bank of America's CD.

A. Chinese Wall Model vs. Bell-LaPadula Model

Feature	Chinese Wall Model	Bell-LaPadula Model
Focus	Conflict of Interest	Confidentiality
Access Control	Based on past access history	Based on security levels
Security Labels	Not used	Security levels & categories
Key Weakness	Difficult to track past access over time	Cannot enforce access restrictions dynamically



Chinese Wall Model vs. Clark-Wilson Model

Feature	Chinese Wall Model	Clark-Wilson Model
Focus	Conflict of Interest	Data Integrity
Access Control	Based on past accesses	Based on authorized transactions
Users	No specific user authentication required	Requires user authentication for actions
Security Weakness	Difficult to track real-world COI cases	Requires strict auditing & enforcement



Unit No: 3

Unit Name: Security Policies Models

Non-Interference & Policy Composition



Overview of Noninterference and Policy Composition

- **Noninterference:** Refers to the principle that actions performed at a high-security level should not be detectable by lower-security-level users.
- If information can be inferred by a lower-level user from a high-level operation, it violates the principle of noninterference.
- **Policy Composition:** This topic covers how different security policies from different systems can be combined to form a new, coherent policy.
- It addresses challenges like what happens when one system's security policy allows certain accesses and another one forbids them.



The Problem of Composition

- The difficulty of composing security policies from different systems. If two systems with different security levels need to be integrated, there's a challenge in defining how the security levels and labels of one system interact with those of another.

Example:

- **Systems with Different Labels:** One system uses labels such as LOW and HIGH, while another uses labels like EAST and WEST. To compose these systems securely, we need to define how these labels correspond to each other (e.g., does HIGH correspond to EAST, and does LOW correspond to WEST?).
- **New Systems Must Reflect Relationships:** This requires a careful mapping of how different security levels and categories from each system interact.



Example of a Covert Channel

- **A covert channel**, a hidden way for users to **secretly communicate** without violating the system's traditional communication channels. Here, CPU utilization is used to signal information.
- **Example for Presentation:**
- **Matt** (high-security) runs a CPU-heavy program.
- **Holly** (low-security) runs a light program.
- To send a "1", Matt runs the program, making the CPU usage rise above 60%.
- To send a "0", Matt does not run the program, keeping the CPU usage below 40%.
- **Holly** observes the CPU usage and decodes the message.



-
- Even though there is no explicit data transfer (like writing to a file), **Matt** has successfully communicated with **Holly** through a **covert channel**, which is a **security vulnerability** that **violates the Bell-LaPadula model**.



Policy vs Mechanism

- It's often difficult to separate the **policy** (rules for accessing data) from the **mechanism** (how the system enforces those rules). In this example, CPU usage becomes a communication channel, violating the **Bell-LaPadula model's -property* (No Write Down) because it allows information to be transmitted from high-level to low-level users.
- **Example:**
- The policy might say that Lara cannot access Holly's data, but the mechanism (CPU usage) allows information to flow indirectly, violating the policy's intent.



Composition of Bell-LaPadula Models (Lattice Approach)

- **Lattices** are used to represent the hierarchy of security levels and categories in each system.
- When combining two systems, you need to merge their **lattices** into a new one.
- **Example:**
 - System 1 (Windsor) has:
 - **Security Levels:** LOW, HIGH
 - **Categories:** EAST, WEST
 - System 2 (Scout) has:
 - **Security Levels:** LOW, S, TS
 - **Categories:** EAST, SOUTH



Composition of Bell-LaPadula Models (Lattice Approach)

- The question is: How do we combine these two lattices?
- **How does S compare to HIGH?**
- **How does SOUTH compare to EAST and WEST?**
- To combine the systems, you need to create a new lattice with levels and categories that reflect these relationships.



Principles of Composition

- When two systems are composed, we need to establish principles that govern how access rights are handled. The **Principles of Autonomy** and **Principles of Security** help define how access is granted or denied.
- **Principles:**
- **Principle of Autonomy:** If a component allows access to a subject, the composed system must allow that access as well.
- **Principle of Security:** If a component denies access, the composed system must deny that access.



Principles of Composition-EXAMPLE

- **System A** denies **Bob** access to **Alice's files**.
- **System B** allows **Eve** and **Lilith** to access each other's files.
- When these systems are composed, **Bob** still cannot access **Alice's files** (due to Principle of Security), but **Bob** may be allowed to access **Eve's files** (due to the composition policy).



Issues in Composition

- When composing security systems, the relationship between the security labels (e.g., LOW, HIGH) must be defined. If the security levels are the same, composition is trivial. If they differ, a new system must be defined to manage the relationships between these levels.
- **Example:**
- If one system uses **LOW** and **HIGH** security levels, and another uses **SOUTH** and **EAST**, the composition of these systems must determine how **SOUTH** relates to **HIGH**, and so on. This requires a careful definition of relationships between different security levels.



Same Policies

- If both systems follow the same security policies, composing them is straightforward. However, if the policies differ, the composition becomes challenging because it must respect both sets of rules.
- **Example:**
- If two systems have the same access control policies (e.g., both systems allow Bob to access certain data), combining them is easy. But if one system allows Bob to access some files while the other does not, combining them becomes difficult.



Different Policies

- When systems with different policies are composed, the challenge is determining what "secure" means in this new context. The combined system needs to ensure that all policies are respected.
- **Example:**
- If **System X** allows **Bob** to access **Alice's** files, but **System Y** allows **Eve** and **Lilith** to access each other's files, the combined policy must decide whether **Bob** can access **Lilith's** files, and so on.



Implications

- When combining systems, the resulting system must satisfy the security requirements of each original system. If the system doesn't explicitly allow or forbid an action, default policies must be followed (e.g., allow or deny based on the principle of least privilege).
- **Example:**
- If **Bob** can access **Eve's** files but **Lilith** can't, and the combined system allows Bob to access Eve's files, it needs to define whether Bob can access Lilith's files.



Deterministic Noninterference

- **Noninterference** ensures that high-level actions do not interfere with or reveal anything to low-level users. In simpler terms, **low-level users** should not be able to observe high-level data or deduce anything from it.
- **Example:**
 - Consider a system with **Holly** (high-security) and **Lucy** (low-security). There are two bits:
 - **H** (high-security bit).
 - **L** (low-security bit).
 - The system should ensure that **Holly's** actions do not affect **Lucy's** ability to observe or modify the **L bit**.
 - If **Holly** changes the **H bit**, **Lucy** should not be able to tell that change from the **L bit**. This is **noninterference**.



Deterministic Noninterference

- In **deterministic noninterference**, we ensure that **one user's actions do not interfere** with what another user can observe in a system.
- A **state machine** helps us understand how a system changes when users execute commands.
- **What is a State Machine?**
 - A **state machine** consists of:
 - **States** (ξ_0, ξ_1, \dots): Different configurations of the system at different times.
 - **Users** ($S = \{\text{Holly, Lucy}\}$): People who interact with the system.
 - **Commands** ($Z = \{\text{xor0, xor1}\}$): Actions that users can perform.
 - **Outputs** (O): What the users can see.
 - Each **command** changes the system's **state** and produces an **output**.



Example: The Two-Bit Machine

Imagine a simple computer system that stores **two pieces of information**:

- **H (High bit)** → **Confidential data** (only Holly can see this).
- **L (Low bit)** → **Public data** (both Holly and Lucy can see this).

The system allows two commands:

1. **xor0** → No change (bits stay the same).
2. **xor1** → Flips both bits (0s become 1s, 1s become 0s).



How Commands Change the System

Command	Before (H, L)	After (H, L)
xor0	(0,0) →	(0,0) (No change)
xor0	(0,1) →	(0,1) (No change)
xor0	(1,0) →	(1,0) (No change)
xor0	(1,1) →	(1,1) (No change)
xor1	(0,0) →	(1,1) (Flips both)
xor1	(0,1) →	(1,0) (Flips both)
xor1	(1,0) →	(0,1) (Flips both)
xor1	(1,1) →	(0,0) (Flips both)



Noninterference-Secure System

- A system is considered noninterference-secure if executing commands within that system does not cause any information to flow from high-security states to low-security states. Here the concept of **output-consistency** in noninterference security is introduced.
- **Example:**
- Imagine a system where **Heidi** (high security) and **Lucy** (low security) are interacting with a 2-bit machine. Heidi modifies only the **H bit** (high security), and Lucy modifies the **L bit** (low security). These actions should not affect each other's data. If Heidi changes the **H bit**, Lucy should not be able to deduce any change in the **H bit** because she is not authorized to observe it.



Proof of Non-interference-Secure

- The **Unwinding Theorem** is a fundamental concept in proving **noninterference security** in computer systems.
- It simplifies the process by allowing us to check **small steps** (state transitions) instead of verifying the entire system at once.
- **What is the Unwinding Theorem?**
 - Instead of **analyzing the entire system**, we "**unwind**" the problem into smaller pieces.
 - If **each individual step** (state transition) follows the security policy, then the **entire system** is secure.



Key Definitions for Understanding the Theorem

Definition 1: Local Respect for Policy

- A system **locally respects a policy (r)** if a command **c** does not cause any visible effect on a protected domain **d**.
☞ **If a command is not supposed to interfere with d, its execution should not be observable in d.**
- **Example:**
- Imagine **Alice (High-Security User)** and **Bob (Low-Security User)** using the same system.
- Alice is modifying **confidential files**, while Bob is using a **public web page**.
- If Bob **notices no change** in his system when Alice modifies files, then the system **locally respects the policy**.



Key Definitions for Understanding the Theorem

- **Definition 2: Transition Consistency**
 - A system is **transition-consistent** if running **the same command** on **two equivalent states** produces **equivalent new states**.
 - **Example:**
 - Suppose two **identical bank accounts** start with **\$1000**.
 - If a transfer of **\$100** is applied to **both accounts**, they should both end up with **\$900**.
 - If one account updates differently, the system is **not transition-consistent**.



Statement of the Unwinding Theorem

- **If a system is output-consistent, transition-consistent, and locally respects a policy r , then it is noninterference-secure.**
- **This means low-security users (L) cannot infer or be affected by high-security users' (H) actions.**



Step-by-Step Proof of the Unwinding Theorem

Step 1: Base Case (Starting Point)



- Suppose there are **no commands** ($cs = \emptyset$).
- Since the system **does nothing**, the final state remains the **same as the initial state**.
- ✓ The system is trivially secure.

Step 2: Induction Hypothesis (Assume the Rule Holds for n Steps)

- Assume that after **n commands**, the system remains **secure**.
- This means executing **n secure commands** does **not reveal high-security actions to low-security users**.



Step-by-Step Proof of the Unwinding Theorem

- **Step 4.3: Induction Step (Prove for $n+1$ Steps)**
- Now, we consider adding **one more command (c_{n+1})**.
-  **Two Cases Arise:**
- **Case 1: The New Command (c_{n+1}) Affects the Domain d**
- The new command **modifies** a state variable.
- We check if **transition consistency holds**.
- Since previous commands were secure, and the new command follows the rules, the system remains **secure**.
-  **Example:**
- Alice (High-Security) edits a **confidential document**.
- Bob (Low-Security) should **not** see any effect.
- If Bob's view remains the **same**, transition consistency holds.



Step-by-Step Proof of the Unwinding Theorem

- **Case 2: The New Command (c_{n+1}) Does Not Affect the Domain d**
- The command **does not interfere** with d .
- The previous security remains **unchanged**, so the system is still **secure**.
- **Example:**
- Alice **changes her password**.
- Bob should **not** be affected.
- Since Bob's state does not change, the system remains **secure**.
- **Step 4: Conclusion**
- Since we have proved that security **holds for both cases**, the **induction is complete**, meaning: ✓ **The entire system is noninterference-secure!**



Step 5: Applying the Unwinding Theorem to Access Control Matrices

- Now, we apply the **unwinding theorem** to **access control matrices (ACM)** to check if they enforce **noninterference security**.
- **What is an Access Control Matrix?**
- A table that **defines what each user (subject) can do with each file (object)**.
- Controls **read/write** permissions in a system.

User	File 1	File 2	File 3
Alice	Read/Write	Read	-
Bob	-	Read	Write
Eve	Read	-	Read



Step 5.2: Security Conditions

- For an ACM system to be **noninterference-secure**, it must satisfy these five conditions:
- **Output Consistency**
 - Commands **depend only on allowed inputs**.
 - ✓ Example: If **Bob can only read File 2**, the system **must not depend on File 1**.
- **Read-Only Dependence**
 - If a command **modifies an object**, it should use **only readable data**.
 - ✓ Example: Bob **cannot write** to File 1 **because he cannot read it**.
- **Write Permission Requirement**
 - If a command **modifies an object**, the user must have **write access**.
 - ✓ Example: Alice **can write** to File 1 but **not to File 3**.



Step 5.2: Security Conditions

- **Read Interference Prevention**

- If **one user can interfere** with another, they must have **shared read permissions**.

- ✓ Example: If Bob writes to **File 3** and Eve **reads** it, Bob can interfere with Eve.

- **Write-Read Interference Prevention**

- If **one user writes to a file** and **another reads it**, they must have a **security relationship**.

- ✓ Example: Bob cannot **write to a file** that **Alice can read** unless security policy allows it.

- **Step 6: Final Conclusion**

- The **Unwinding Theorem** proves that **if each state transition follows security rules**, the **whole system is secure**.

- It **reduces complexity** by checking **small steps** instead of **entire system behavior**.

- **Access Control Matrices** can be checked using the **five security conditions** to ensure **noninterference security**.

- ✓ **If all rules hold, high-security actions remain hidden from low-security users!**



Real-World Example of the Unwinding Theorem in Action: Banking System Security

- **Step 1: Defining the Security Model**
- Imagine a bank has three types of users:
- **Bank Manager (High Security - H)**
 - Can **approve loans** and **modify high-level financial data**.
- **Bank Teller (Medium Security - M)**
 - Can **view account balances** and **perform transactions for customers**.
- **Customer (Low Security - L)**
 - Can **only view their own account balance**.



Step 2: Defining Security Goals

- ☞ The bank follows a **strict security policy** to prevent information leaks:
- **A bank manager's actions (H)** should **not** affect what a customer (L) sees.
- **A teller (M) should not see loan approvals (H actions)** unless explicitly allowed.
- **Customers (L) should only see their own transactions and nothing from tellers (M) or managers (H).**
- 💡 **Example of a Potential Security Leak:**
- If a customer **requests a loan status update**, but the system **pauses or slows down** whenever a manager is reviewing loans, the customer might **guess** whether they were approved or denied.
- This is a **covert information leak** violating **noninterference security**.



Step 3: Checking the Unwinding Theorem Conditions

- The **Unwinding Theorem** tells us that **if we verify three conditions**, the system is **noninterference-secure**.
- **Condition 1: Output Consistency**
- Any command **should produce the same output** when executed in **equivalent system states**.
- ✓ **Banking Example:**
- Two **customers with the same balance** should see **the same response** when checking their balance.
- If a manager **approves a loan**, it **should not change** what the customer sees until the customer is officially notified.
- ⚠ **Potential Violation:**
- If the customer **notices a delay** whenever a loan is being approved in the background, they might infer their loan status.



Condition 2: Transition Consistency

- If two system states are **equivalent**, applying the **same command** should lead to **equivalent new states**.
- ✓ **Banking Example:**
 - Suppose two customers **request a balance check** at the same time.
 - If their accounts have **the same starting balance**, they should see **the same balance update** after a deposit.
 - If one customer's balance updates **before the other's**, the system **breaks transition consistency**.
- ⚠ **Potential Violation:**
 - If customers notice **account balance updates happening at different speeds**, they might **infer** when a teller is handling transactions.



Condition 3: Local Respect for Policy

- If an action **should not affect a low-security user**, it must **not be visible** to that user.
- ✓ **Banking Example:**
 - A manager **approving a loan** should not **affect a customer's experience** until the decision is officially communicated.
 - The **customer should not observe delays, loading screens, or error messages** related to loan processing.
- ⚠ **Potential Violation:**
 - If the **loan approval system** slows down the **customer's online banking** experience, they might **guess** that their loan is being processed.



Step 4: Applying the Proof by Induction

- We use **induction** to prove that if each small step is secure, the **entire banking system is secure**.
- **Base Case (No Actions)**
- If **no transactions occur**, the system remains **unchanged** and trivially secure.
- **Inductive Hypothesis (Assume Security Holds for n Transactions)**
- Assume that after **n transactions**, customers can **only see their own data** and no **covert channels** exist.
- **Inductive Step (Prove for $n+1$ Transactions)**
- Consider a **new banking transaction**:
 - If it affects **only authorized users**, the system remains **secure**.
 - If it affects an **unauthorized user**, we must check if this **violates local respect for policy**.
 - If **noninterference is maintained**, the system remains **secure**.
- ✓ **Final Conclusion:**
If every small step follows the security rules, then the **entire banking system remains secure**.



Step 5: How the Bank Can Fix Security Violations

- If the **Unwinding Theorem** identifies security risks, the bank can **modify its system** to prevent **information leaks**.
- **Fix for Output Consistency Issues**
- ✓ **Solution:** Add **deliberate delays** to ensure **uniform response times**, preventing customers from guessing when a manager is reviewing loans.
- **Fix for Transition Consistency Issues**
- ✓ **Solution:** Implement **atomic transactions**, ensuring all customers see **consistent account updates**.
- **Fix for Local Respect for Policy Issues**
- ✓ **Solution:** Separate **loan processing** from **customer interactions** so that customers cannot **observe system slowdowns**.



Changing Security Policies and System Composition

- The concept of **noninterference security** to situations where **security policies change over time** and explores **how multiple secure systems, when combined, may become insecure**.
- **1: Security Policies That Change Over Time**
- We have assumed that **security policies remain fixed**. However, in real-world systems:
- **Permissions change over time** (users gain or lose access).
- **Users may transfer rights to others** (delegation of access).



-
- **What Happens When Policies Change?**
 - In **real-world systems**, security policies are dynamic.
 - **Example: Discretionary Access Control (DAC) in a File System**
 - **Holly (User)** owns a file "**confidential.txt**".
 - She grants "**read**" **access** to **Matt**.
 - Later, she revokes Matt's access.
 - The system must ensure that **Matt cannot use past information** to infer changes.



Handling Dynamic Changes in Policies:

To modify the **noninterference model** to handle the fact that policies may change.

We do this by introducing a **predicate function** that determines whether a subject **has permission** to execute a command at any given point.

Predicate Function: **cando(w, s, z)**

- **Definition:** $\text{cando}(w, s, z)$ evaluates whether subject **s** can execute the command **z** at state **w**.
- If $\text{cando}(w, s, z)$ is **false**, subject **s** does not have permission to execute command **z**, and thus, it cannot interfere with the system.

Example:

- Suppose **Holly** has permission to **read file f**, but after a policy change, **Holly** no longer has permission to do so.
- In this case, $\text{cando}(w, \text{Holly}, \text{"read f"})$ would evaluate as **false**, and Holly cannot execute the "read" command.
- Thus, Holly's inability to read **file f** will **not interfere** with the system's other users.



Modifying Command Sequences Based on Permissions

- When we incorporate dynamic changes to security policies, we modify the sequence of commands (or actions) being executed by a subject.
- This modification ensures that only commands the subject is **authorized** to execute will affect the system.
- **How It Works:**
 - If a subject attempts to execute a command that they are **no longer allowed** to execute (due to a policy change), the sequence is **modified** to exclude that command.
- **Example:**
 - Suppose **Holly** initially had permission to execute a command to **read file f**.
 - If the policy changes, and **Holly** no longer has permission to execute this command, the modified sequence of commands would **remove** the "read file f" command, preventing **Holly** from interfering with the system or other subjects.



Policy Amplification: Passing Rights to Other Users

- In more complex systems, **subjects (users)** may have the ability to **pass rights** to other subjects, granting them permission to execute certain commands they would not otherwise be able to.
- **Explanation:**
- Imagine a situation where there are multiple users in a system, each with different **security levels** (e.g., **high** and **low**). Some systems allow high-security subjects to **pass** certain rights to low-security subjects.



Policy Formalization:

- To make sure the **system** remains **secure** when rights are passed:
- A subject **cannot execute a command** unless they had the right to do so in the **previous state** or the right has been explicitly **passed** to them.
- The system must ensure that rights are **explicitly passed** from one subject to another, and that no **unauthorized subjects** can access resources.



Example: Complex Security with Passing Rights

- **Scenario:**
- **Lucy** (low security) wants to **read a file** but doesn't have permission to do so initially.
- **Holly** (high security) has the **pass(s2, z)** command, which means **Holly** can pass permission for **Lucy** to execute the command "**read file**".
- **Steps:**
- **Before passing the right:** **Lucy** cannot read the file, since she doesn't have the right.
- **Holly grants the right:** **Holly** uses **pass(s2, z)** to pass the "**read**" right to **Lucy**.
- **After the right is passed:** Now, **Lucy** can read the file, because **Holly** explicitly allowed her to do so.



Key Rules to Ensure Security

•Output Consistency

- A command **only depends on the values it has permission to read.**
- If Bob can **only read File 2**, then a command **must not depend on File 1 or 3.**

•Read-Only Dependence

- If a command **changes a file**, it should use **only readable data** to determine the new value.
- Bob cannot **write to File 1** if he cannot read it.

•Write Permission Requirement

- If a command **modifies** a file, the user must have **write access.**
- Bob cannot **change File 1's content** because he **lacks write access.**

•Read Interference Prevention

- If **one user can interfere with another**, they must share **read permissions.**
- If Bob writes to File 3 and Eve reads File 3, Bob can interfere with Eve.

•Write-Read Interference Prevention

- If **one user writes to a file** and **another reads it**, they must have a security relationship.
- Otherwise, **unintended data leaks** might happen.



-
- **If These Five Rules Hold, the System is Noninterference-Secure**
 - This means **low-level users cannot infer high-level operations.**
 - The **unwinding theorem** proves that **if each step respects these rules, then the whole system is secure.**



Composition of Policies

- **Main Idea:** Two organizations with different security policies want to combine their systems. The challenge is how to combine their separate policies into a coherent, unified security policy.
- **Example:**
- Two companies with different security protocols want to collaborate on a shared project. Company A has stricter data security measures than Company B. The challenge is to create a unified security policy that works for both companies without compromising security.



The Problem Example

- The scenario involves a system with two users, Holly and Lara, operating on separate virtual machines (VMs). Despite being isolated in terms of direct communication, they can still "communicate" through CPU load, forming a covert channel.
- **Example:**
- **Holly** (high-security) can alter CPU utilization to "signal" information to **Lara** (low-security).
 - To send a "1," Holly runs a program that raises CPU utilization to over 60%.
 - To send a "0," she doesn't run the program, and the CPU usage stays below 40%.
 - Lara observes the CPU usage to deduce Holly's message.
- This scenario demonstrates **covert channels**, where sensitive data leaks through unintentional side channels like CPU usage.



Policy vs Mechanism

- It's often difficult to separate the **policy** (rules for accessing data) from the **mechanism** (how the system enforces those rules).
- In this example, CPU usage becomes a communication channel, violating the **Bell-LaPadula model's -property* (No Write Down) because it allows information to be transmitted from high-level to low-level users.
- **Example:**
- The policy might say that Lara cannot access Holly's data, but the mechanism (CPU usage) allows information to flow indirectly, violating the policy's intent.



Issues in Composition

- When composing security systems, the relationship between the security labels (e.g., LOW, HIGH) must be defined. If the security levels are the same, composition is trivial. If they differ, a new system must be defined to manage the relationships between these levels.
- **Example:**
- If one system uses **LOW** and **HIGH** security levels, and another uses **SOUTH** and **EAST**, the composition of these systems must determine how **SOUTH** relates to **HIGH**, and so on. This requires a careful definition of relationships between different security levels.



Same Policies

- If both systems follow the same security policies, composing them is straightforward. However, if the policies differ, the composition becomes challenging because it must respect both sets of rules.
- If two systems have the same access control policies (e.g., both systems allow Bob to access certain data), combining them is easy. But if one system allows Bob to access some files while the other does not, combining them becomes difficult.



Different Policies

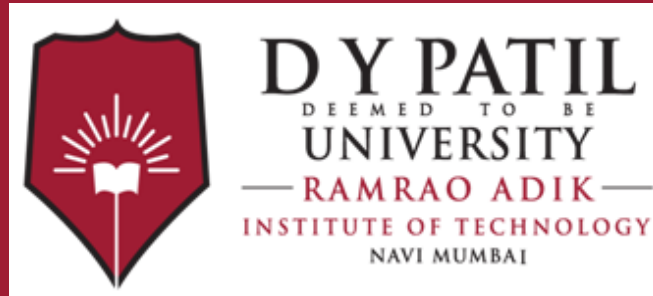
- When systems with different policies are composed, the challenge is determining what "secure" means in this new context. The combined system needs to ensure that all policies are respected.
- **Example:**
- If **System X** allows **Bob** to access **Alice's** files, but **System Y** allows **Eve** and **Lilith** to access each other's files, the combined policy must decide whether **Bob** can access **Lilith's** files, and so on.



Interference

- Interference refers to a situation where high-level actions (e.g., Holly's actions) indirectly affect low-level observations (e.g., Lara's ability to deduce information), which violates noninterference.
- **Example:**
- **Holly** (high level) affects the **CPU load**, which **Lara** (low level) can observe, even though they are not supposed to communicate.





Thank You