

## The seL4 microkernel

# Introduction to seL4 Microkernel

## ▶ What is seL4?

- ▶ A **formally verified**, high-performance operating system kernel.
- ▶ Designed for **security- and safety-critical systems**, embedded, and cyber-physical applications.

## ▶ Core Features

- ▶ Minimal **microkernel architecture**.
- ▶ Strong **isolation** and **fine-grained access control** through capabilities.
- ▶ Robust support for **real-time** and **mixed-criticality systems**.

## ▶ Key Features

- ▶ Minimal **Trusted Computing Base (TCB)** for reduced attack surface.
- ▶ Strong **isolation** through capability-based access control.
- ▶ **Formally verified** with machine-checked proofs
  - ▶ Ensures **functional correctness** and **security enforcement**.
  - ▶ Guarantees **confidentiality, integrity, and availability**.
- ▶ Verification spans from **abstract model** to **binary code**.

## Introduction to seL4 Microkernel (cont'd)

- ▶ **Dual Role**
  - ▶ Operating system **microkernel**.
  - ▶ **Hypervisor** for secure virtual machines.
- ▶ **Real-Time and Mixed-Criticality Support**
  - ▶ Complete analysis of **worst-case execution time**.
  - ▶ Suitable for **hard real-time systems**.

# Applications and Benefits of seL4

## ▶ Applications

- ▶ Autonomous vehicles, defense systems, and **embedded devices**.
- ▶ IoT devices requiring security and safety.
- ▶ Cyber retrofitting of legacy systems.

## ▶ Key Advantages

- ▶ Industry benchmark for **reliability** and **performance**.
- ▶ Enables integration of existing software into **secure environments**.
- ▶ Combines **robust security** with **practical adaptability**.

## seL4 is a Microkernel, not an OS

- ▶ **Fundamental Difference** Unlike monolithic kernels (e.g., Linux), seL4
  - ▶ Minimizes code in **privileged mode**.
  - ▶ Reduces the **Trusted Computing Base (TCB)** and **attack surface**.

# Monolithic v. microkernel

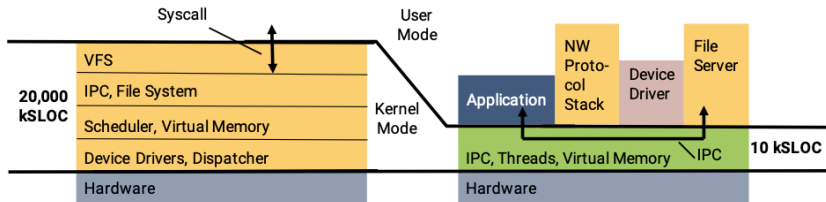


Figure 1: Operating system structure: Monolithic v. microkernel

## Monolithic vs. Microkernel

### ▶ **Monolithic Kernels**

- ▶ Integrate all essential services (e.g., drivers, file systems) into the kernel.
- ▶ Tens of millions of lines of code = higher vulnerability.

### ▶ **Microkernels**

- ▶ Provide only minimal functionality for managing hardware and isolating processes.
- ▶ Delegate OS services to **user-space programs**.

### ▶ **Benefits of seL4**

- ▶ Strong **isolation**.
- ▶ Fine-grained access control using **capabilities**.
- ▶ Lightweight, low-level API for high efficiency.
- ▶ Modular design ensures **fault isolation**, **security**, and **resilience**.

# Hypervisor Capabilities of seL4

## ► Virtual Machine Support

- Secure execution of full OSES (e.g., Linux) alongside **native applications**.
- Enables **seamless integration** of native and virtualized components.

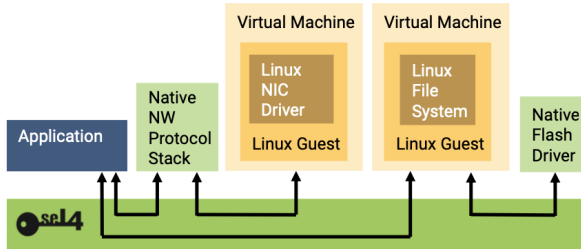


Figure 2: Virtualization to integrate native services with Linux



## ▶ Key Features

### ▶ Fine-grained capability-based access control

- ▶ Ensures strict isolation between VMs, native services, and applications.
- ▶ Prevents faults or compromises in one domain from affecting others.
- ▶ Secure interaction via **well-defined communication channels**.

## ▶ Incremental Modernization

- ▶ Run **legacy systems** in virtual machines.
- ▶ Operate new components **natively** for enhanced security and performance.
- ▶ Achieves modernization without a full system overhaul.

# seL4 for Real-Time Systems

- ▶ **Precise Timing Guarantees**
  - ▶ Priority-based scheduling for **predictable execution**.
  - ▶ Developers can control thread priorities to meet **strict deadlines**.
- ▶ **Key Features for Real-Time**
  - ▶ **Bounded worst-case execution times** for all kernel operations.
  - ▶ **Minimal and predictable interrupt latencies**, even under heavy workloads.
- ▶ **Mixed-Criticality Systems (MCS)**
  - ▶ Secure coexistence of components with varying safety and timing needs.
  - ▶ Strong isolation prevents interference between components.
- ▶ **Capability-Based Resource Management**
  - ▶ Precise, secure allocation of **time resources** alongside memory and I/O.
- ▶ **Ideal Applications**
  - ▶ Avionics, autonomous vehicles, and time-critical **embedded systems**.

# Verification of seL4

## ► **Formal Verification**

- First OS kernel with **machine-checked verification** of functional correctness.
- Ensures implementation aligns with high-level specifications.
- Guarantees absence of
  - Buffer overflows.
  - Null-pointer dereferences.
  - Code injection vulnerabilities.

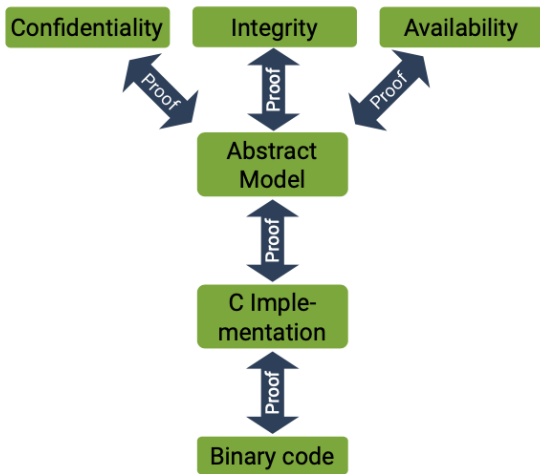


Figure 3: seL4 proof chain

▶ **Translation Validation**

- ▶ Extends verification to the compiled binary.
- ▶ Proves that the binary matches the verified source code, even with optimizations.
- ▶ Ensures **high assurance** in the deployed form.

▶ **Security Properties**

- ▶ Proven enforcement of **confidentiality, integrity, and availability**.
- ▶ Strict access controls and effective isolation of components.

▶ **Assumptions for Verification**

- ▶ Correctness of hardware.
- ▶ Accuracy of specifications.
- ▶ Reliability of the theorem prover.

▶ **Impact**

- ▶ Bridges formal reasoning and real-world execution.
- ▶ Sets a **new benchmark** for security and reliability in critical systems.

# Functional Correctness of seL4

## ► Definition

- Rigorous proof that seL4's C implementation - Is free from defects. - Adheres to a formal specification expressed in **higher-order logic (HOL)**.
- Guarantees the kernel behaves strictly as defined by its **abstract model**.

## ► Verification Process

- Uses **Isabelle/HOL** theorem prover.
- Translates C code into mathematical logic for formal verification.
- Restricts C usage to a well-defined subset with unambiguous semantics.
- Ensures the implementation remains **provably correct**.

# Translation Validation in seL4

## Why Translation Validation?

- ▶ Bug-free C implementation  $\leftrightarrow$  Guaranteed reliability
  - ▶ **Compilers** are complex systems that may
    - ▶ Introduce defects during code translation.
    - ▶ Contain bugs or malicious code (e.g., Trojan backdoors described by Ken Thompson).



# What is Translation Validation?

- ▶ Process of verifying the compiled binary against the **formally verified C code**.
- ▶ Ensures the binary faithfully represents the verified source code.

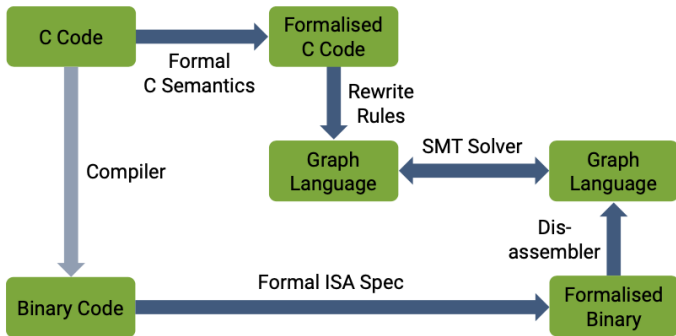


Figure 4: Translation validation proof chain

## ▶ **How it Works**

1. Formalization of the processor's ISA (instruction set architecture).
2. Disassembly of the binary and transformation into a graph-based intermediate representation.
3. Transformation of C code into the same intermediate representation.
4. Use of SMT solvers and rewrite rules to prove equivalence.

## ▶ **Key Outcome**

- ▶ Validates that the compiler's output matches the **abstract specification**.
- ▶ Ensures seL4's high-assurance guarantees extend to the executable binary.

## ▶ **Impact**

- ▶ Bridges the gap between the **formal model** and **real-world deployment**.
- ▶ Protects against risks introduced by compiler behavior.

# Security Properties of seL4

- ▶ **Core Guarantees**

- ▶ **Confidentiality**

- ▶ Ensures no unauthorized entity can read or infer data.
    - ▶ Enforced through strict access control mechanisms.

- ▶ **Integrity**

- ▶ Prevents unauthorized modifications to data.

- ▶ **Availability**

- ▶ Protects against denial of authorized resource access.

- ▶ **Formal Proofs**

- ▶ Demonstrate the kernel's ability to secure critical systems.
  - ▶ Mitigate common attack vectors.

- ▶ **Limitations**

- ▶ Current model does **not yet cover timing-related security issues**.
  - ▶ Timing channels remain an area of active research.

- ▶ **Enhancements for Real-Time Systems**

- ▶ **Mixed-Criticality Systems (MCS)** model
    - ▶ Extends integrity and availability guarantees to include **timeliness**.
    - ▶ Ensures security in real-time environments.

# Proof Assumptions in seL4 Verification

## ► **Explicit Assumptions in Formal Reasoning**

- Every assumption is **explicitly defined** and documented.
- Prevents risks of **overlooking or misinterpreting** critical dependencies.
- Enhances clarity and confidence in system correctness.

## ► **Key Assumptions**

### 1. **Hardware behaves as expected**

- Kernel guarantees depend on reliable hardware.
- Faulty or malicious components invalidate kernel assurances.

### 2. **Specification matches expectations**

- Formal specification must align with intended behavior.
- A gap may exist between **mathematical reasoning** and **real-world interpretation**.

### 3. **Theorem prover is correct**

- Tools like Isabelle/HOL have a **small, well-tested core**.
- The risk of bugs in the prover affecting proofs is extremely low.

# CAmkES Component Framework

## ► What is CAmkES?

- A framework for designing systems on **seL4** as collections of **isolated components**.
- Components interact through **defined communication channels**.
- Formal **Architecture Description Language (ADL)** ensures secure system interactions.

## Main Abstractions in CAmkES

### 1. Components

- ▶ Represented as **square boxes**.
- ▶ Self-contained units of **code and data**.
- ▶ Encapsulated by **seL4**, functioning as independent programs.

### 2. Interfaces

- ▶ Define how components interact
  - ▶ **Importing** Invoke another component's interface.
  - ▶ **Exporting** Allow other components to invoke their interface.
  - ▶ Symmetric **shared-memory interfaces** for direct data sharing.

### 3. Connectors

- ▶ Link **importing** and **exporting interfaces** for communication.
- ▶ **One-to-one** by design; additional components enable **broadcast** or **multicast**.

## Automated Translation and Setup

- ▶ **ADL** (architecture description language) CapDL\*\* (capability distribution language)
  - ▶ ADL specifies architecture; CapDL defines **seL4 objects** and **access rights**.
  - ▶ Ensures faithful implementation of the described architecture.
- ▶ **Generated Code**
  - ▶ **Startup code** Initializes seL4 objects and allocates capabilities.
  - ▶ **Glue code** Simplifies communication between components via function calls.



## Key Benefits

- ▶ Simplifies **design**, **verification**, and **implementation** of secure systems.
- ▶ Enables **sandboxed components** with precise communication channels.
- ▶ Maintains **security and reliability** through formal specifications and automated tools.

# Capabilities in seL4

## ► What are Capabilities?

- Object references similar to **pointers** but include **access rights**.
- Immutable and uniquely reference specific objects.
- Encapsulate the rights needed to operate on objects.



Figure 5: A capability is a key that conveys specific rights to a particular object

## Example

- ▶ An operation may be to call a function in a component.
- ▶ The object reference embedded in the capability then points to an interface to that object
- ▶ It conveys the right to invoke that function, a particular method on the component object.
- ▶ The capability may or may not at the same time convey the right to pass another capability along as a function argument
  - ▶ (delegating to the component the right to use the object referenced by the capability argument).

# Key Features of Capabilities

## 1. Fine-Grained Access Control

- ▶ Invoking a capability is the only way to operate on system objects.
- ▶ Ensures strict adherence to the **principle of least privilege**.

## 2. Delegation

- ▶ Capabilities can be passed to delegate access securely.

## 3. Kernel Protection

- ▶ Unlike traditional ACLs, capabilities are
  - ▶ Managed by the **kernel**.
  - ▶ Immune to vulnerabilities like the **confused deputy problem**.

## Types of Objects in seL4 referenced capabilities

1. **Endpoints** are used to perform protected function calls;
2. **Reply Objects** represent a return path from a protected procedure call;
3. **Address Spaces** provide the sandboxes around components (thin wrappers abstracting hardware page tables);
4. **Cnodes** store capabilities representing a component's access rights;
5. **Thread Control Blocks** represent threads of execution;
6. **Scheduling Contexts** represent the right to access a certain fraction of execution time on a core;
7. **Notifications** are synchronisation objects (similar to semaphores);
8. **Frames** represent physical memory that can be mapped into address spaces;
9. **Interrupt objects** provide access to interrupt handling; and
10. **Untyped** unused (free) physical memory that can be converted ("retyped") into any of the other types.

## Benefits of Capabilities

- ▶ **Security**

- ▶ Restrict access to the **minimum rights** required for tasks.
- ▶ Avoid common vulnerabilities of traditional systems.

- ▶ **Comprehensive Control**

- ▶ Offers precise, object-oriented access control.

- ▶ **Reliability**

- ▶ Robust enforcement of system policies ensures high assurance.

# Fine-Grained Access Control in seL4

- ▶ **Capabilities vs. ACLs** (access control list)
  - ▶ **Capabilities**
    - ▶ Object-oriented access control.
    - ▶ Aligns with the **Principle of Least Privilege (POLA)**.
    - ▶ Grants access only to explicitly authorized resources.
  - ▶ **ACLs** (Access-Control Lists)
    - ▶ Subject-oriented (based on user or group IDs).
    - ▶ Coarse-grained permissions limit precise security enforcement.

# Access control in Linux

- ▶ File has an a set of access-mode bits.
  - ▶ Some of those bits determine what operations its owner can perform on the file
  - ▶ Others represent operations by each member of the file's "group"
  - ▶ A third gives rights to everyone else

```
-rw-r--r--@    1 dgcole  staff    181 Nov 12 14:48 .zshrc
drwx-----@    4 dgcole  staff    128 Aug 27 13:26 Applications/
drwxr-xr-x@   31 dgcole  staff    992 Jan 22 17:57 Desktop/
```

- ▶ This is a **subject-oriented** theme
  - ▶ a property of the subject/process attempting access
  - ▶ all subjects with the same value of the property (ID or group) have the same rights



## Capability control

- ▶ Capabilities provide object-oriented access control
- ▶ The kernel will allow an operation if and only if the subject requesting the operation presents a capability that allows it to perform the operation
- ▶ E.g., an untrusted app can only access files for which it has been given a capability
- ▶ Alice invokes a program
  - ▶ she handing it a capability to the one files the program is allowed to read,
  - ▶ plus a capability to a file where the program can write its output,
  - ▶ the program is unable to access anything else — proper least privilege.

## Confinement of Untrusted Programs

### ▶ **Traditional Systems (e.g., Linux)**

- ▶ Restricting access requires cumbersome workarounds like
  - ▶ chroot jails
  - ▶ Containers

### ▶ **Capabilities in seL4**

- ▶ Precisely grant access to specific resources
  - ▶ Example: Reading/writing specific files.
- ▶ Ensure untrusted programs cannot interact with unauthorized resources.

## Key Benefits of Capabilities

### 1. **Precision**

- ▶ Allows application-specific access control.

### 2. **True Least Privilege**

- ▶ Confines programs to minimal required permissions.

### 3. **Simplicity and Security**

- ▶ Avoids the complexity and vulnerabilities of ACL-based systems.

# Solutions for Delegation and Interposition

## Interposition

- ▶ Capabilities enable transparent mediation
  - ▶ Capabilities are opaque references.
  - ▶ Example: A capability given to a user points to a security monitor instead of the resource.
- ▶ Applications of interposition
  - ▶ Enforcing security policies.
  - ▶ Packet filtering.
  - ▶ Debugging and resource virtualization.

## Interpose access

A consequence of the fact that they are opaque object references. I

- ▶ Alice is given a capability to an object, she has no way of knowing what that object really is, all she can do is invoke methods on the object.
- ▶ The system designer may pretend that the capability given to Alice refers to a file, when in fact it refers to a communication channel to a security monitor, which in turns holds the actual file capability.
- ▶ The monitor can examine Alice's requested operations
  - ▶ If valid,
    - ▶ then performs them on the file on her behalf
    - ▶ else ignoring invalid ones
- ▶ The monitor effectively virtualises the file.

# Delegation with Capabilities

## ▶ **Efficient Privilege Delegation**

- ▶ Users can “**mint**” new capabilities with specific permissions (e.g., **read-only access**).
- ▶ Delegate capabilities securely to others.
- ▶ **Revocation**
  - ▶ Capabilities can be revoked at any time, enhancing control.

## ▶ **Autonomous Resource Management**

- ▶ Subsystems can independently manage their resources
  - ▶ Maintain isolation and security.
  - ▶ Avoid reliance on centralized control.

# Delegation

Capabilities support safe and efficient delegation of privilege.

- ▶ If Alice wants to give Bob access to one of her objects,
  - ▶ then she can create (“mint” in seL4 speak) a new capability to the object and hand it to Bob.
- ▶ Bob then can use that capability to operate on the object without referring back to Alice.
- ▶ The new capability can have diminished rights
- ▶ Alice can use this to give Bob only read-only access to the file
- ▶ Alice can revoke Bob’s access at any time by destroying the derived capability she handed to Bob.

# Delegation is powerful and cannot easily and safely be done in ACL systems

A typical use case: setting up sub-systems that manage resources autonomously

- ▶ When the system starts up, the initial process holds authority to all resources in the system
- ▶ This initial resource manager can then partition the system, by creating new processes (secondary resource managers) and handing them privilege to disjoint subsets of the system resources
- ▶ The subsystems can then autonomously, without referring back to the original manager, control their subset of resources, while unable to interfere with each other
- ▶ Only if they want to change the original resource allocation do they need to involve the original manager.



## Benefits of Capabilities for Delegation and Interposition

### 1. **Flexibility**

- ▶ Tailor capabilities for specific tasks or permissions.

### 2. **Transparency**

- ▶ Mediate access without revealing the resource.

### 3. **Control**

- ▶ Simplify revocation and enhance autonomous management.

### 4. **Enhanced Security**

- ▶ Achieve goals difficult with traditional access-control systems.

# The Confused Deputy Problem

- ▶ **Definition**

- ▶ A security vulnerability where a program (**deputy**) is tricked into misusing its authority, leading to unintended actions.

## Classic Example The Compiler Incident

A C compiler.

- ▶ It takes a C source file and produces an object-code output file, the file names are passed as arguments
- ▶ A user, Alice, must have execute permission on the compiler

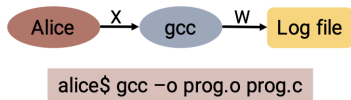


Figure 6: The compiler as a confused deputy

- ▶ Assume the compiler also creates an entry in a system-wide log file for auditing purposes
- ▶ The log file is not accessible to normal users, so the compiler must execute with elevated privilege in order to write to the log file

- ▶ If Alice is malicious,
  - ▶ then she can specify the password file as the output file when invoking the compiler.
  - ▶ The compiler, unless it is written very carefully to avoid any potential abuse, will just open the output file (password file) and overwrite it with the compiled object code
  - ▶ It doesn't take a lot of skill for Alice to write a program which compiles such that the newly generated password file will give her privileges she should not have

The fundamental problem: ACL-based systems use ambient authority for determining access rights.

- ▶ When the compiler opens its output to write, the OS looks at the compiler's subject ID to determine whether it has access to the object
- ▶ It is up to the compiler to determine whether the operation is valid or not, making the compiler part of the system's TCB
- ▶ It has to be fully trusted to do the right thing under all circumstances

## Root Cause

- ▶ **Authority Misuse**
  - ▶ The compiler uses its own permissions rather than the user's, leading to unintended access.

## Mitigation Capability-Based Security

### ▶ **Solution**

- ▶ Use capabilities that combine object designation with access rights.
- ▶ Ensures programs operate only within explicitly granted permissions, preventing such vulnerabilities.

## How Capabilities Solve This Problem

- ▶ **Coupling Denomination and Authority**

- ▶ A capability references an object and encapsulates its **access rights**.
- ▶ Operations require the user to provide explicit capabilities.

- ▶ **Example**

- ▶ A compiler in a capability-based system
  - ▶ Operates only within the authority granted explicitly by the user.
  - ▶ Cannot access or modify resources beyond its designated scope.

## Benefits of Capabilities in Avoiding Confusion

### 1. **Elimination of Ambient Authority**

- ▶ Prevents unauthorized operations.

### 2. **Secure Delegation**

- ▶ The user explicitly controls access rights.

### 3. **True Principle of Least Privilege**

- ▶ Programs act only on resources for which they have explicit capabilities.

### 4. **Essential for Secure Environments**

- ▶ Ensures reliable and unambiguous access control.



# Hard Real-Time Support in seL4

- ▶ **Priority-Based Scheduling**
  - ▶ Simple, deterministic **priority-based policy**.
  - ▶ No autonomous priority adjustments—full control remains with the user.
- ▶ **Low Interrupt Latencies**
  - ▶ **Bounded latencies** achieved by
    - ▶ Disabling interrupts during kernel mode.
    - ▶ Simplified design with no need for complex concurrency control.
  - ▶ Efficient **short system calls** eliminate the need for preemptible kernels.

## Handling Long-Running Operations

### ► **Incremental Consistency**

- Operations (e.g., capability revocation) are broken into **smaller sub-operations**.
- If interrupted
  1. Current operation is **aborted**.
  2. Interrupt is processed.
  3. Operation **resumes** where it stopped.

## Worst-Case Execution Time (WCET) Analysis

- ▶ **Key Features**

- ▶ **Sound and complete WCET analysis** provides provable upper bounds for
  - ▶ System call latencies.
  - ▶ Interrupt handling.

- ▶ **RISC-V Opportunity**

- ▶ Open-source RISC-V processors enable reapplying WCET analysis.
- ▶ Reinforces seL4's strength in **real-time** and **safety-critical systems**.

## Advantages for Real-Time Systems

1. Deterministic and efficient performance.
2. Responsive even during complex operations.
3. Guarantees critical for **safety-critical environments**.

# Mixed-Criticality Systems (MCS)

## Definition

Systems with components of varying **criticality levels**

- ▶ **Criticality** Severity of consequences in case of a failure.
- ▶ Example Avionics classify failures from “no effect” to “catastrophic.”

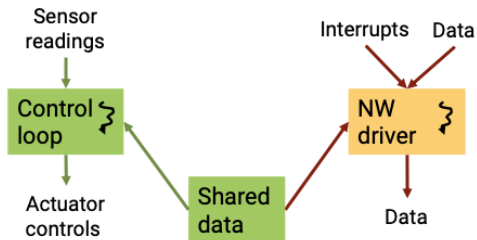


Figure 7: A simplified example of mixed-criticality

## Key Characteristics of MCS

### 1. Strong Isolation

- ▶ Ensures lower-criticality failures cannot affect higher-criticality components.

### 2. Consolidation Goals

- ▶ Reduces **space, weight, and power (SWaP)** by consolidating functionality.
- ▶ Mimics the principle of isolating trusted and untrusted components with added safety requirements.

### 3. Real-Time Challenges

- ▶ Safety demands **timeliness** and real-time deadline adherence.
- ▶ Both **functional correctness** and timing are critical.

## Traditional MCS with Time and Space Partitioning (TSP)

### ▶ **Strict Isolation**

- ▶ Fixed memory areas and dedicated time slices for each component.
- ▶ Guarantees **spatial and temporal isolation**.

### ▶ **Resource Efficiency Issues**

- ▶ **Worst-Case Execution Time (WCET)** allocations
  - ▶ Time slices sized for worst-case scenarios.
  - ▶ Leads to underutilized processor resources.
- ▶ Slack time is wasted, as it cannot be reallocated.

### ▶ **Interrupt Latency Challenges**

- ▶ Strict time slicing delays handling of **external events**.
- ▶ Example Autonomous vehicle
  - ▶ Control loop runs every 5 ms.
  - ▶ Interrupts delayed by time slices, impacting responsiveness.

## Trade-offs

1. **TSP** ensures strong isolation but resembles inefficiencies of air-gapped systems.
2. **Efficient MCS** requires balancing isolation with resource utilization and real-time responsiveness.



# Mixed-Criticality Systems (MCS) in seL4

## Core Challenge

Achieve **strong resource isolation** without the rigidity of strict Time and Space Partitioning (TSP).

## Scheduling-Context Capabilities in seL4

### ▶ **Key Features**

- ▶ Regulate processor access by
  - ▶ **Time budget** How much CPU time a component can use.
  - ▶ **Time period** How often the budget can be used.
- ▶ Prevents components from monopolizing CPU while ensuring **responsiveness**.

### ▶ **Advantages over Traditional Time Slices**

- ▶ More **granular control** of CPU allocation.
- ▶ Enables dynamic resource utilization with strict isolation.

## Example Critical vs. Non-Critical Components

### ▶ Scenario

- ▶ Critical control loop Requires **guaranteed CPU availability**.
- ▶ Non-critical driver Needs **high responsiveness** but must not interfere.

### ▶ Configuration

- ▶ Critical Controller
  - ▶ Budget **3 ms**.
  - ▶ Period **5 ms**.
  - ▶ Guarantees **60% CPU availability**.
- ▶ High-Priority Driver
  - ▶ Smaller budget and shorter period.
  - ▶ High responsiveness without exceeding **30% CPU time**.

### ▶ Result

- ▶ Ensures **critical deadlines** are met, regardless of non-critical behavior.
- ▶ Fulfills MCS requirements with flexibility and isolation.

## Why seL4 for MCS?

- ▶ **Advanced Time Capabilities**

- ▶ Granular CPU control ensures **isolation** and **real-time guarantees**.
- ▶ State-of-the-art solution for **safety-critical environments**.

**Security is No Excuse for Poor Performance**

# Deployment and Incremental Cyber Retrofit

## Planning Deployment with seL4

### 1. Identify and Protect Critical Assets

- ▶ Structure assets as **modular, seL4-protected CAmkES components**.

### 2. Verification for Highest Assurance

- ▶ Use the verified kernel for your platform when possible.
- ▶ Even unverified versions provide stronger guarantees than most OSes.

### 3. User-Level Infrastructure

- ▶ Evaluate if existing components meet your needs.
- ▶ Collaborate with the community or specialized providers for missing infrastructure.

### 4. Contribute Back

- ▶ Share useful components under an **open-source license** to foster collaboration.

### ▶ Virtualization for Legacy Components

- ▶ Legacy systems often cannot be ported due to
  - ▶ Size or complex dependencies.
  - ▶ Minimal security benefits from running natively.
- ▶ Use seL4's **virtualization capabilities** as a baseline.

## Example DARPA HACMS and Boeing ULB

- ▶ Initial Setup: Linux system placed in a **VM** on seL4.
- ▶ Incremental Transformation
  - ▶ Isolated untrusted components (e.g., camera software, GPS) into
    - ▶ Separate VMs.
    - ▶ Native CAMkES components
  - ▶ Critical modules moved to **secure native implementations**.

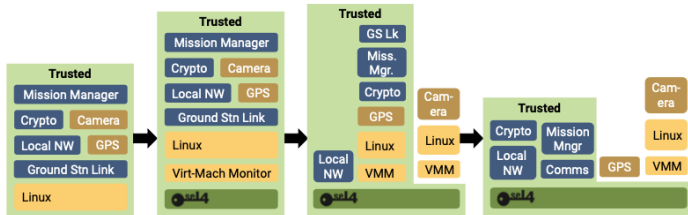


Figure 8: Incremental cyber-retrofit of the Boeing ULB mission computer

- ▶ Result: Even if Linux is compromised, the rest remains secure.

## Key Benefits

- ▶ Gradual system modernization with **minimal disruption**.
- ▶ Enhanced **security and resilience** against attacks.
- ▶ Efficient reuse of legacy systems while isolating critical components.