



# Common pitfalls and errors when using Linux in Safety Applications

Igor Stoppa

<istoppa@nvidia.com>



LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026





# A Note on This Deck

- This deck may seem to discourage the use of Linux for safety applications.
- **NVIDIA's intent is the opposite.**
- But there are **many problems to solve** and obstacles to overcome.
- First, **gaps and feasibility** must be identified and thoroughly understood.
- We are not trying to prove it is all already available.
- This is a **shared, reviewable knowledge base**, open to all.



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# A Note on the Wording

Concepts presented in the talk refer to the Safety standard ISO 26262.

To keep this talk accessible:

- Terms are **simplified** and do not always match ISO 26262 wording exactly.
- The underlying **concepts are retained** and stay faithful to the standard.
- Access to the standard is not free of cost, so the presentation will be standalone.
- In definitions where exact wording matters, the standard is **quoted directly**.

**Simpler words, same concepts.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Agenda

**Summary**

**Functional Safety**

**Safety Concepts**

**Linux & Safety**

**Taking Stock**

**Conclusions and a Question**

**(Cheat Sheet)**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026





# Summary

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# In a nutshell

Linux can serve safety, but only with **justified, targeted effort**.

## Linux WRT Safety

Not built for safety

- Features **don't add up** to a safety case.
- Monolithic: **anything can interfere**.

## What It Takes

Justified and verifiable

- Know the **kernel** and every dependency.
- Prove completeness with **mechanisms**.

## Safety Is Targeted

Driven by the use case

- Mechanism **cost**.
- Focus on use cases, **not every driver**.
- The hardening must **survive intact**.

**Identify and close focused gaps, exhaustively and permanently.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Functional Safety

---

LINUX  
PLUMBERS  
CONFERENCE  
PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Functional Safety

ISO 26262-1:2018, clause 3.67:

*"absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems"*

Defined terms:

- **E/E systems:** electrical and/or electronic systems, comprising both HW & SW.
- **Malfunctioning behaviour:** failure or unintended behaviour of an item with respect to its design intent.
- **Unreasonable risk:** risk judged unacceptable in a certain context according to valid societal moral concepts. **Target: acceptably low risk, not zero risk.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026





# The Conundrum

The definition relies on **subjective** terms:

- **Unreasonable:** who draws the line?
- **Acceptable:** to whom, in which context?
- **“Valid societal moral concepts”:** vary by country, culture and era.

Projects routinely **tailor** the standard:

**Criteria are interpreted and applied per scenario  
and implementation.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# The Tension Within the Standard

The standard is a **live**, evolving set of guidelines:

- **Broader scope:** now spans simple microcontrollers to complex systems running server-class OSes.
- **Negotiated:** by regional delegations with differing ideas, methods and goals.
- **Commercial interests:** some positions may reflect industry practice; challenging them can meet resistance.
- **Mutual surveillance:** no party can steer the outcome alone.

**The price is compromise: some statements end up less than unambiguous.**



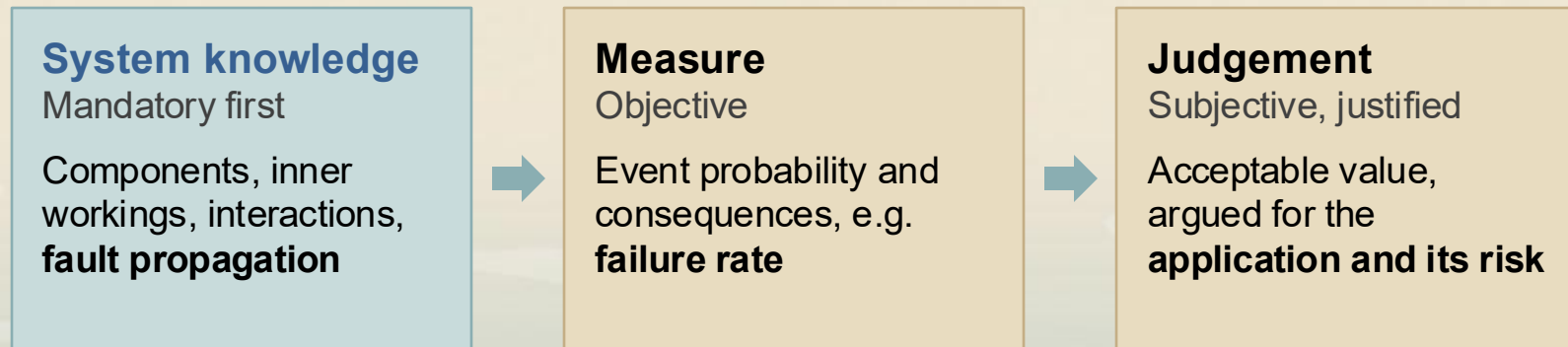
LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# The Completeness Problem

“Unreasonable risk” is subjective. **Verifiable justification** keeps it in check:



**Know the system completely and in detail,  
before judging what is reasonable.**





# Taming Subjectivity

**Assessors** validate how companies interpret and apply the standard to specific safety problems:

- **Goal:** some uniformity across companies and projects.
- **In practice:** assessors' own interpretations vary.
- **Certification:** whether a process suffices for a certificate of conformity varies by assessing body.
- **Linux:** its complexity divides assessors on how far microcontroller practices remain valid on microprocessors.

**Independent Assessment (3rd party) narrows interpretation, but does not eliminate it.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Separating Fact From Interpretation

How an approach meets functional safety needs is rarely **public**:

- **Assessment:** the assessor issues a certificate and a report.
- **Private:** the report and any evidence stay confidential.
- **Published:** only code under legal obligation (e.g. GPLv2).
- **Result:** comparing approaches is often practically impossible.

**The facts: safety concepts mapped onto Linux.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

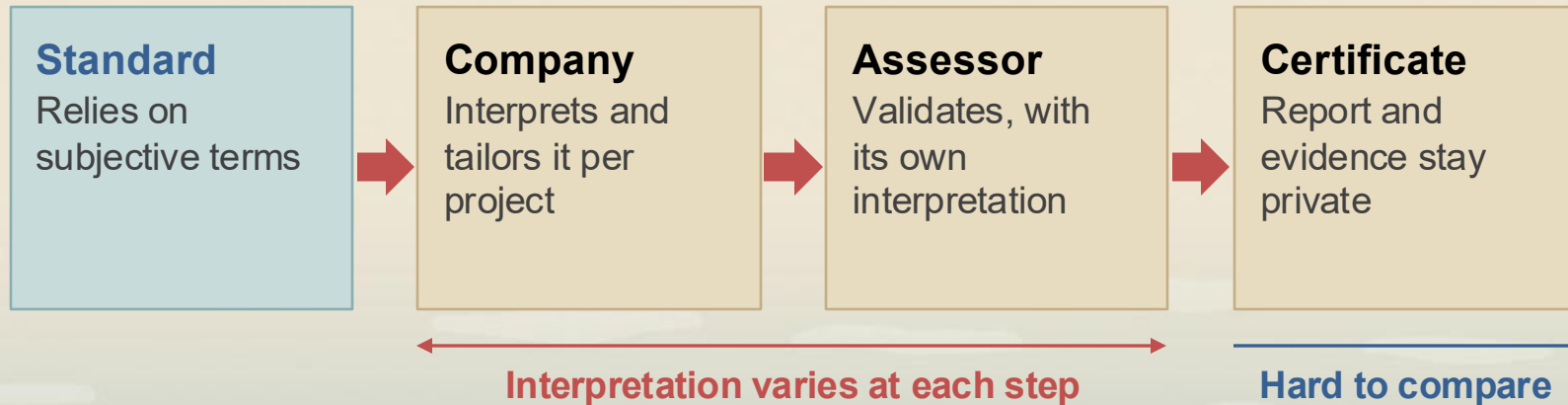


NVIDIA





# From Standard to Certificate





# Safety Concepts

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Safety Concepts 101

Safety accepts that failures may happen. It requires they are **handled as specified**:

- **Avoid**: keep faults out through design and process.
- **Detect**: notice the failures that still occur, reliably and in time (Fault Detection Time Interval).
- **Control**: move the system to a safe state, reliably and in time, and keep it there (Fault Reaction Time Interval).
- **Mitigate**: limit the harm, reliably and in time, if the hazard occurs anyway.

**Every safety goal aims to prevent harm.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

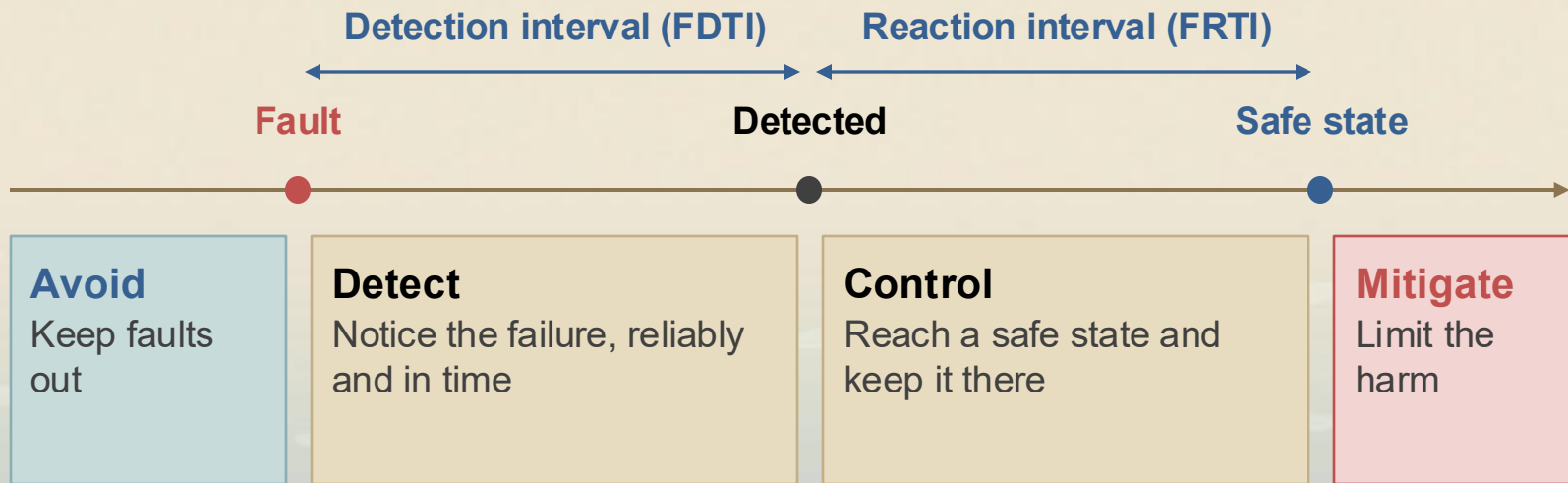


NVIDIA





# Detect and React in Time





# Failure as Disruption of Integrity

A system keeps its integrity in two dimensions:

- **Spatial:** its state changes only through intended operations (includes communication).  
*e.g. no data/message is overwritten by accident.*
- **Temporal:** events follow the expected order and timing for its inputs.  
*e.g. no task runs too late, too early or out of order.*

**A failure is any disruption of either.  
Interference is a failure cascading into another element  
and violating a safety goal (safety requirement).**





# Two Dimensions of Integrity

## Spatial

State changes only through intended operations

Faulty element



Overwritten  
data

e.g. a message overwritten by accident

## Temporal

Events follow the expected order and timing

Expected

1

2

3

Actual

1

3

2

e.g. a task runs too late or out of order





# Freedom From Interference (FFI)

**Freedom from interference:** safely coping with cascading failures by either **avoiding**, **detecting**, **controlling** or **mitigating** them.

- For each safety-relevant component: **how can it receive interference?**
- This is **objective**: it can be studied, proven, demonstrated, even empirically.
- Its twin: **common cause failures**, one root cause hitting several elements at once (e.g. a shared lock or allocator).

**Any safety measure should be verifiable.**

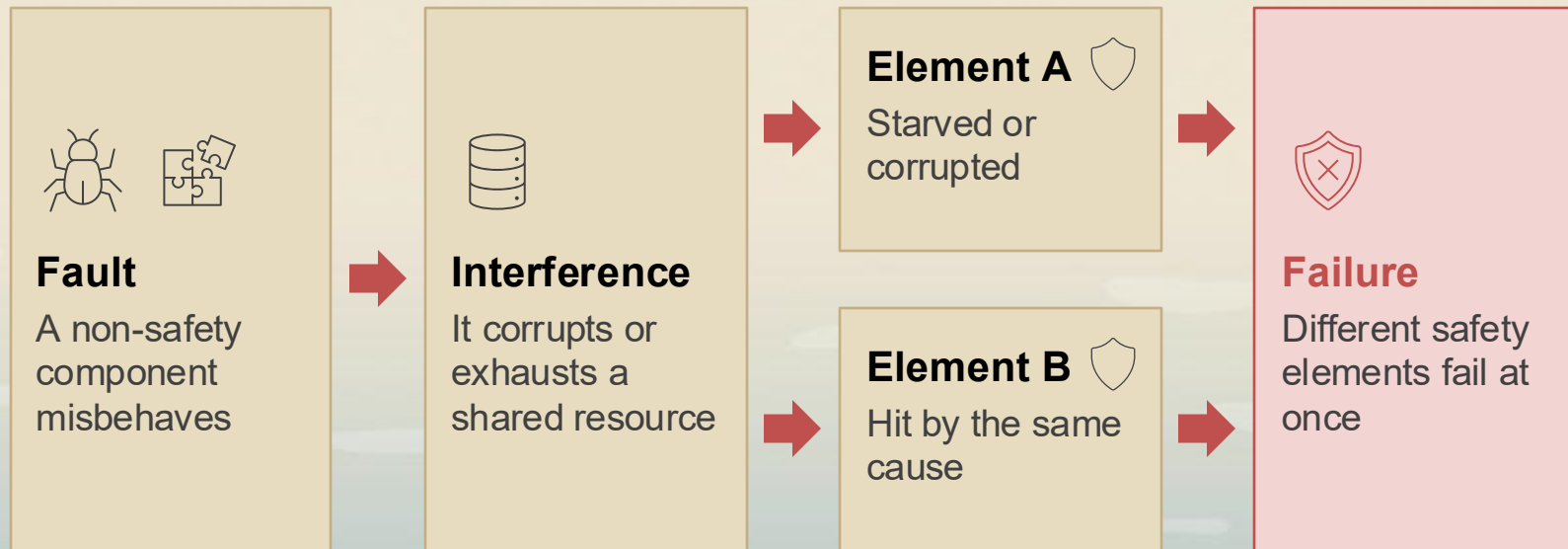


LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Cascading Failures



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

 NVIDIA





# Safety Claim

It states that an element **meets its safety requirements**, at a given level of safety.

## Evidence

Tests, analyses, reviews



## Argument

Why the evidence shows the requirement is met



## Claim

“Element X meets requirement Y at Safety Level Z”

**Assumptions and scope:** hardware, kernel version and config, use case. The claim holds only within them.

Claims make up the safety case the assessor evaluates.



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Integrity of a Safety Claim

A safety claim holds only while everything it rests on **stays valid**.

## Safety claim

“Element X meets requirement Y”



## Evidence

Tests, analyses,  
reviews

## Assumptions

Kernel version,  
config, hardware,  
use case

## Managing its integrity

A claim is not proven once: it must be kept true.

- Any change to code, config, toolchain or use case can **invalidate** it.
- **Trace** each claim to the exact artefacts it relies on.
- **Assess every change** for impact, then re-verify what it touches.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Verifiability of Safety Measures

How objectively can we verify that a safety measure is adequate, and **complete**?

## Process-based measures

Useful to avoid systematic failures.

Verified **indirectly**:

reviews, audits, assessment

- The process can be **observed and audited**.
- How well it catches deviations can, at best, be **estimated**.

## Safety mechanisms

Useful to detect and control systematic failures.

Verified **both indirectly and directly**: reviews, audits, simulation, **fault injection**

- Mechanisms can be **exercised** on the running system.
- Their response to **each targeted interference** can be tested.
- May affect performance.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Quality vs Safety

## Quality

Standard management of features/defects

- **Meet the specification, even implicit.**
- Reduce defects across **typical use**.
- Evidence: process and test results.

## Safety

ISO 26262 requirements

- **Absence of unreasonable risk.**
- Address **hazardous failures, even rare ones**.
- Evidence: a **safety case** covering the **safety requirements**.

**Requirements make safety stricter and more focused.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Quality vs Availability

**Availability** (ISO 26262-1)

*“Capability of a product to be in a state to execute the function required under given conditions, at a certain time or in a given period, supposing the required external resources are available.”*

**MTBF** (mean time between failures, IEC 60050-192)

*Expected operating time between consecutive failures of a repairable system.*

**Both measure continuity of service, not safety.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Safety Needs Focused Quality

## Quality

- A product with poor overall quality can still be safe, **provided that components with safety requirements meet them.**

## Availability

- Availability: the component **keeps providing** its safety function.
- Safety: it is enough that any discontinuation is **managed safely.**

**A product can be both safe and of very poor overall quality, as long as the safety of the relevant components is preserved.**





# Linux & Safety

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Then and Now

|          | Dedicated ECUs                                                                        | Linux-based platforms                                                    |
|----------|---------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Hardware | Multiple Simple <b>microcontroller</b>                                                | Complex hardware, close to <b>server-grade</b> architectures             |
| Software | Custom, <b>purpose-built</b> , often written from scratch for one or a few tasks      | Linux, effectively a <b>general-purpose</b> operating system             |
| Safety   | The process can be <b>fully enforced</b> , including adherence to safety requirements | Safety must be <b>retrofitted</b> onto existing general-purpose software |

ECU-oriented practices don't automatically apply  
to more complex systems



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Monolithic: A Runaway Safety Scope

In a monolithic kernel, **everything can interfere with everything**.  
There is no naturally isolated subset of components, for safety to focus on.

**The dilemma:** safety calls for hardening all interdependent components, yet taken all together, they are **unmanageable**.

## Route 1: claim it's all safe

...and prove it,  
**despite all the observations made.**

## Route 2: carve out a safe subset

... but the handling of the carve-out must  
be **both provable and complete.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# One Address Space, No Barriers

All kernel code runs at **one privilege level, in one address space**.

## Inside the kernel

Any code can reach any writable data

- No **hardware-enforced barrier** between components.
- e.g. a driver use-after-free can **overwrite page tables**.

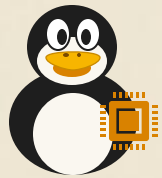
## Towards user space

The linear map exposes process memory

- The kernel can **write to any process's memory**.
- A kernel bug can indirectly let **one process corrupt another**.

**Isolation between processes does not cover the kernel.**





# Inside One Address Space

**Kernel: one address space, no barriers**

**Faulty driver**

**Other subsystems**

overwrites

**Page tables**

**Linear map**

**User space**

**Process A**

Isolated from each other

**Process B**





# Pure Process vs Mechanisms

|               | Pure process                                                                                              | Mechanism-based                                                               |
|---------------|-----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| Applicability | Attempt to retrofit to what exists                                                                        | Can adjust as needed                                                          |
| Gaps          | No direct way to close gaps                                                                               | Add mechanisms for each gap                                                   |
| Scope         | Must cover the <b>entire kernel</b>                                                                       | Process still applies.<br><b>But only for the mechanisms</b>                  |
| Verification  | Not possible to identify if there are errors, through the use of <b>negative testing</b> .                | Both positive verification and <b>negative testing</b> (e.g. fault injection) |
| Fit           | Suits small, purpose-built ECU code with an adjustable process;<br><b>hard to apply to upstream Linux</b> | Process targets the mechanisms:<br><b>small, ad hoc, under control</b>        |

**Mechanisms must prove to be both effective and complete.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

 NVIDIA





# Catching Interference

Interference can start **anywhere** in the kernel.

## At the source

Too many sources to cover

- Mechanisms not designed for safety are both complex and subjects to bugs
- Generic mechanism not designed to catch **complex systematic corruption**.
- Hardware is **not a catch-all**: ECC fixes bit flips, not wrong writes.

## At the receiver

Where all sources converge

- Interference can be **modelled and detected** where it lands.
- The check must be **isolated** from the faults it detects.

**Catch interference where it lands, not where it starts.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Failures in the Linux Kernel

A **thorough understanding of the kernel architecture** is needed to know:

- which components support **which safety goal** (safety requirement);
- what types of interference a component can **produce**;
- what types of interference a component can **receive**;
- whether the kernel functionality **fits the usage scenario**.

**Failures: not just bugs,  
also design–usage gaps (assumptions of use).**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Taking Stock

Action is needed, in **process** and **practice**.

But what **counts** for safety?

And is it **effective and adequate**?

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Misconceptions

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Frequent Misconceptions

**No single “Linux”:** every version, build and hardware mix differs.

**Kernel process:** helps, but isn't aligned with safety standards.

**Functional Correctness:** strong evaluations help, but don't imply safety.

**Functional Availability:** safety often doesn't require it.

**Stress testing:** useful, but can't cover all plausible scenarios.

**Monitoring:** dependencies, observability and common cause failure.

**Upstreaming:** current practices can't prevent regressions in mechanisms.



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# One Kernel, Many Variants

*“Linux runs everywhere, and well, so it must be safe.”*

Field experience doesn't transfer: each installation **differs** in

- **Kernel:** specific versions, very likely with their own patches.
- **Workloads:** specific and non-uniform (e.g. cloud farms), hard to generalise.
- **Operating conditions:** differ from those in safety-related use, including the failures they expose.
- **Crash data:** may not show whether a crash was a safety issue.

**Linux runs everywhere, and well, so it must be of high quality.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Kernel Process vs Safety Process

## Kernel development process

Affects the whole kernel

- Focus on **overall quality**, performance and other parameters.
- Safety is **not among its objectives**.

## Safety process

Only for safety-relevant components

- Focus exclusively on components with **safety requirements**.
- It goes **above and beyond quality**, but only for affected components.

**Focus and practices are not naturally aligned.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Functional Correctness vs Safety

## Kernel development

All components treated equally

- Huge range of **configurations, targets, and use cases.**
- What is secondary for some users is **key for others.** Everything is important.

## Safety

Specific applications and paths

- Isolates and **prioritises certain paths.**
- Those paths **carry the weight** of the safety requirements.
- On those paths, **unhandled errors are not allowed.**

**Safety applies stricter criteria to selected paths.**





# Functional Availability vs Safety

## System availability

Needs redundant, possibly diverse HW

- HW diversity reduces **common cause failures**.
- Each unit has its own, **diverse SW**.
- SW alone **can't provide it**.

## Safety

Needs failures to be handled

- Handling must be **safe** and **reliable**.

**Safety requires reliable failure handling, not only SW availability.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# The limits of (stress) testing

Testing faces the same **completeness problem** as a pure process approach:

## Testing for correctness

A completeness-of-observability problem

- Which execution paths were covered, and which **were not**?
- **Code coverage** shows branches ran, not the **call paths** that reached them.
- Same code, other paths: **different results** (e.g. wrong order of actions).

## Generic stressors

e.g. stress-ng: adequacy must be proven

- Do they push **kernel internals** to their limits?
- e.g. are **CPU caches drained**? Are **core migrations** exhaustive?
- Is the profile aligned with the **specific kernel + HW**?

**Testing must still prove completeness.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

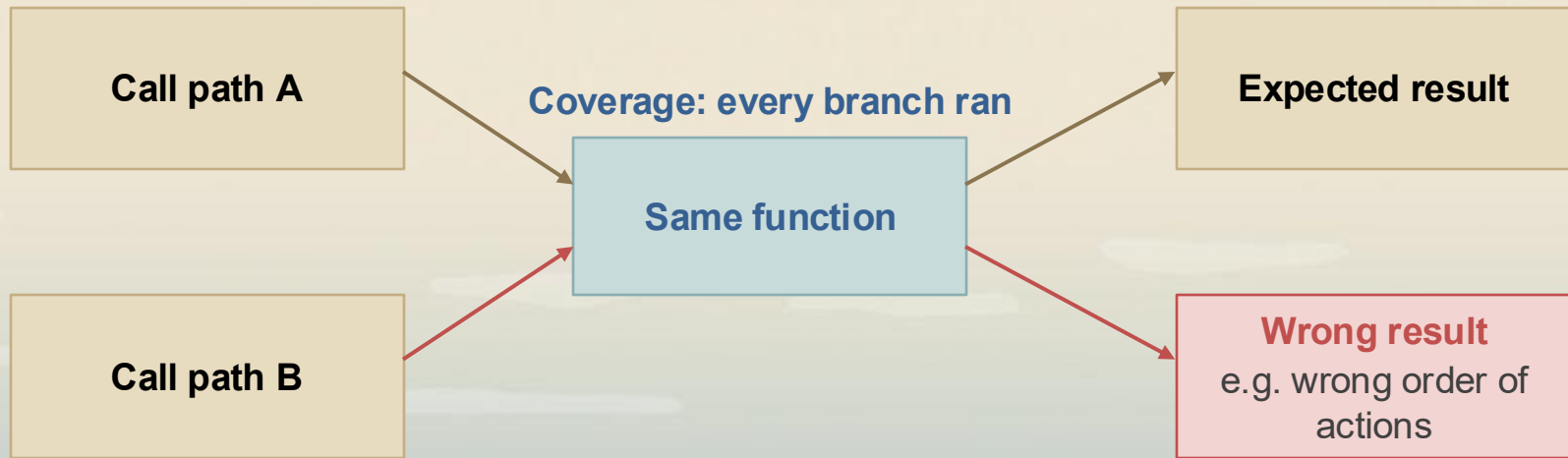


NVIDIA





# Covered, yet can lead to errors.





# Testing for Safety

Tests support a safety claim only if they are **targeted and repeatable**.

## Targeted

Driven by requirements and analysis

- **Positive tests** derived from the safety requirements.
- **Negative tests** focused by analysis of plausible interference.

## Repeatable

The same test, the same conditions

- Randomisation (e.g. **structure layout**) makes each build differ.
- **Preemption** can be necessary but it can expose latent races.

**A result that can't be reproduced can't be argued.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Internal Monitoring vs Safety

## Monitoring

Inside the kernel, or relying on it

- Faults **propagate** across the kernel.
- Faults **propagate** from kernel to user-space.
- Monitor shares the **same faults** it should detect.
- **The more the dependencies**, the weaker the isolation.

## Safety

Needs an independent monitor

- Monitoring mechanisms must be **isolated** from the faults they detect. (Required by the safety standard.)
- In the monolithic kernel, the monitor meta-data is exposed to the same interference.

**A monitor that depends on the kernel can fail with it.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# External Monitoring vs Safety

## Monitoring

Outside the kernel

- Monitor is **protected** from kernel interference.
- Kernel faults cannot reach the monitor's own state.

## Observability

Needs kernel internals exposed

- Symbols and events to monitor live **inside** the kernel.
- They must be **exposed** to the external monitor.
- Too many to expose: **unfeasible**.
- Overhead of exposing them, especially high-frequency ones: **impractical**.

An external monitor is isolated, but it hardly can see enough.



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026

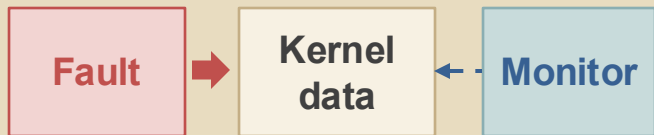




# Monitor: Exposure vs Visibility

## Internal monitor

### Kernel

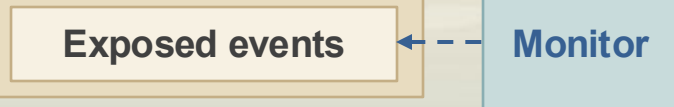


Shares the faults it should detect.

## External monitor

### Kernel

Most internals stay hidden



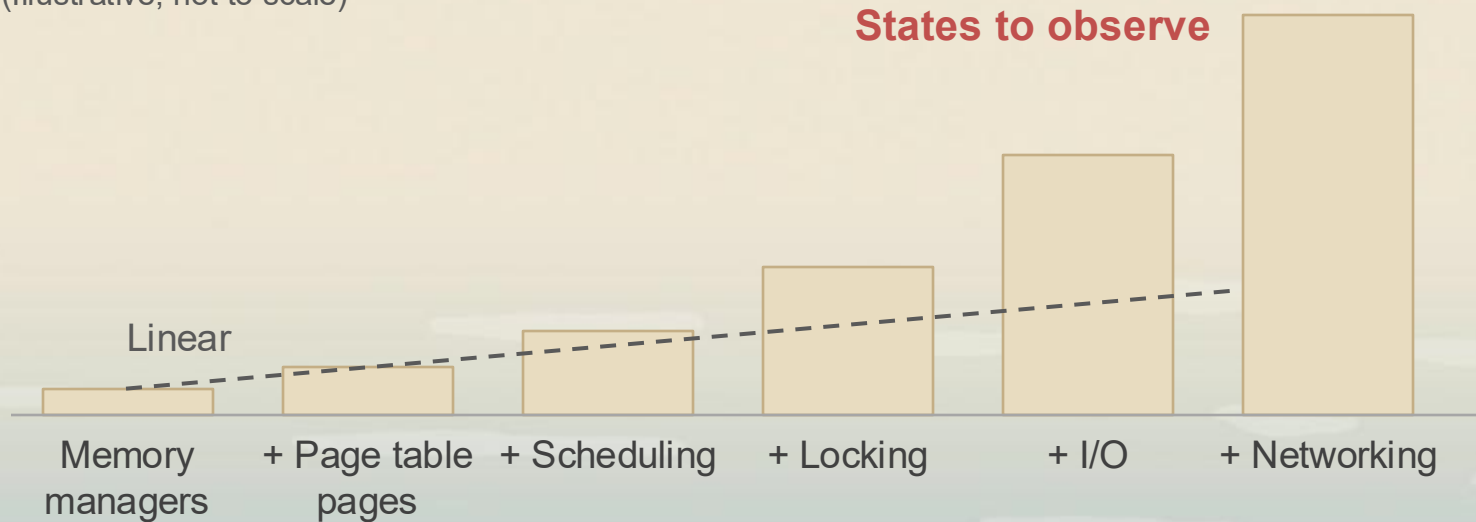
Isolated, but most likely it sees little.





# Observed States Grow Non-Linearly

(Illustrative, not to scale)



LINUX  
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026





# Upstreaming vs Safety

## Safety mechanisms

Must stay isolated from the kernel

- Their shape must keep them **not exposed** to the rest of the kernel.
- This isolation must be **maintained** for as long as the code lives.

## Upstream development

Code is reworked over time

- Any later change can introduce an **unwanted dependency**.
- The mechanism then shares faults with the kernel: **common cause failure**.
- Without a **safeguard**, nothing stops this degradation.

No upstream safeguard preventing regressions in mechanisms



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Safety Regressions

## When merged

Kernel code

Mechanism

Kept isolated: kernel faults do not reach it.



## After later rework

Kernel code



Mechanism

A new dependency shares the kernel's faults: common cause failure.





# Configuring Linux

---

LINUX  
PLUMBERS  
CONFERENCE  
PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Misconceptions on Configuration

**Config flags:** useful, but alone don't make a kernel safe.

**PREEMPT\_RT:** useful, but alone doesn't make a kernel safe.

**Security features:** partial overlap with safety, but different goal.

**Resources Management:** partial overlap with safety, but different goal.

**Like the kernel process, its configuration knobs  
alone are not sufficient for safety.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026

 NVIDIA





# The Limits of Config Flags

Kconfig exposes only what **its authors envisioned**, rarely for safety.

## Disabling features

Useful, but not enough

- Removes **unneeded features**.
- What remains is **unknown** from Kconfig alone.
- Without inspecting the code, it is usually **not safe per se**.

## Explicit settings

Don't rely on defaults

- Defaults reflect **the author's idea** of a valid choice.
- Set **all relevant options** explicitly.

**Configuration shapes the kernel, not its safety.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Limits of relying on PREEMPT\_RT

PREEMPT\_RT reduces worst-case latency **once a task becomes runnable**.

## Purpose

Exactly what it was designed for

- Reduce worst-case **scheduling latency**
- Helps **precisely** where latency matters.
- Preemption can expose latent races by changing interleavings, requiring more thorough testing and analysis.
- **Not a safety mechanism**: it doesn't detect or control faults.

## Safety needs more than that

Latency is only one aspect

- Correct **sequences** of actions.
- Sometimes a **minimum time** before acting.
- PREEMPT\_RT **metadata** is exposed to interference: what if corruption **alters the event timing**?

**Enabling PREEMPT\_RT alone is not enough.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

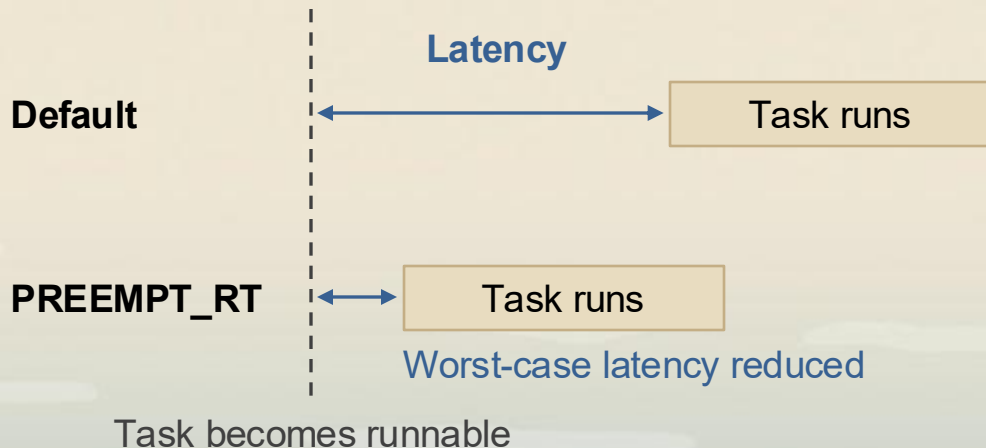


NVIDIA





# What PREEMPT\_RT Changes



## Not addressed

- Correct order of actions
- Minimum time before acting
- Faults in its own metadata





# Security vs Safety

|            | Security                                                       | Safety                                                                     |
|------------|----------------------------------------------------------------|----------------------------------------------------------------------------|
| Cause      | An intelligent <b>attacker</b> , usually a multi-step sequence | <b>Failures</b> , no actor; independent fault chains mostly out of scope   |
| Entry      | <b>Layered defences</b> : outer shells must be breached first  | Flaws may be <b>already inside</b> , with direct access to sensitive areas |
| Targets    | Focus on <b>high-value targets</b>                             | <b>Anything</b> can be affected                                            |
| Mechanisms | Their faults matter only if they <b>can be attacked</b>        | Must share <b>no common cause</b> with the interference they guard against |

Safety can't rely on outer perimeters: faults start inside.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

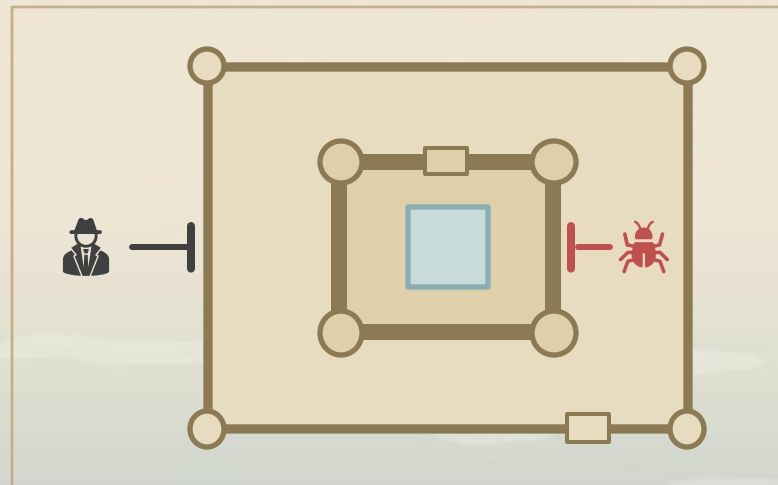
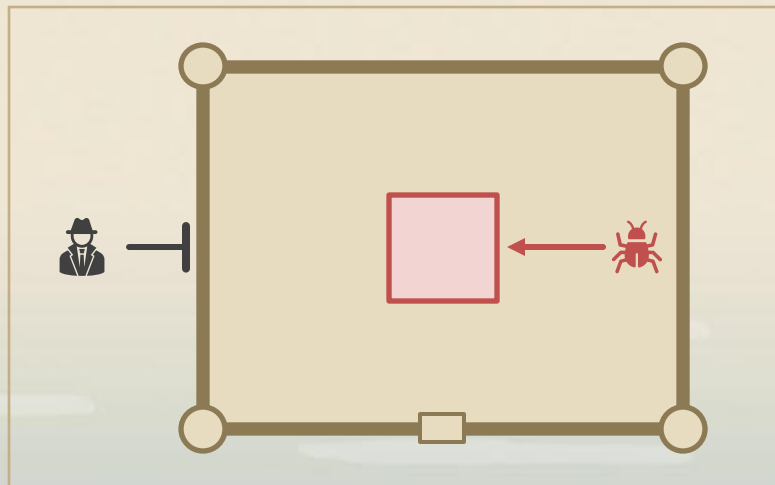


NVIDIA





# Safety needs more than Security

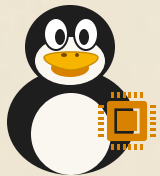


**The Safety Relevant content must be protected**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Resource Management vs Safety

In spirit, resource management resembles safety: it **contains impact**, but ...

## cgroups

Limits what a group of processes uses

- **Throttles** or caps CPU, memory and I/O.
- Accounting is **partial**: not all kernel-internal allocations are charged.

## Kernel allocation

Lazy, and allowed to overcommit

- Memory **overcommit**: granted is not backed.
- Reads of untouched memory map the **zero page** – **no real resource is obtained**.
- File data is read **on access** (plus readahead).
- Assigned **just in time**: failures and delays may hit at a **critical moment**.

**A granted resource isn't always an available one.**





# Granted Is Not Backed

## Request

Allocation  
granted, but not  
backed by  
memory



## Read

Untouched  
memory maps  
the zero page



## First write

Real memory  
assigned just in  
time



## Failure or delay

Hits at a critical  
moment

Time



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Reusing Kernel Components

Most kernel components were devised for **other purposes**.

## By design

Each re-uses whatever it finds useful

- Maximize reuse of **other components**.
- Result: a **tight, often intricate graph** of dependencies.

## For safety

Tempting to reuse, but not trivial

- Every dependency must be **argued** too.
- Defending one component **pulls in practically everything else**.

**There is no such thing as a free lunch.**



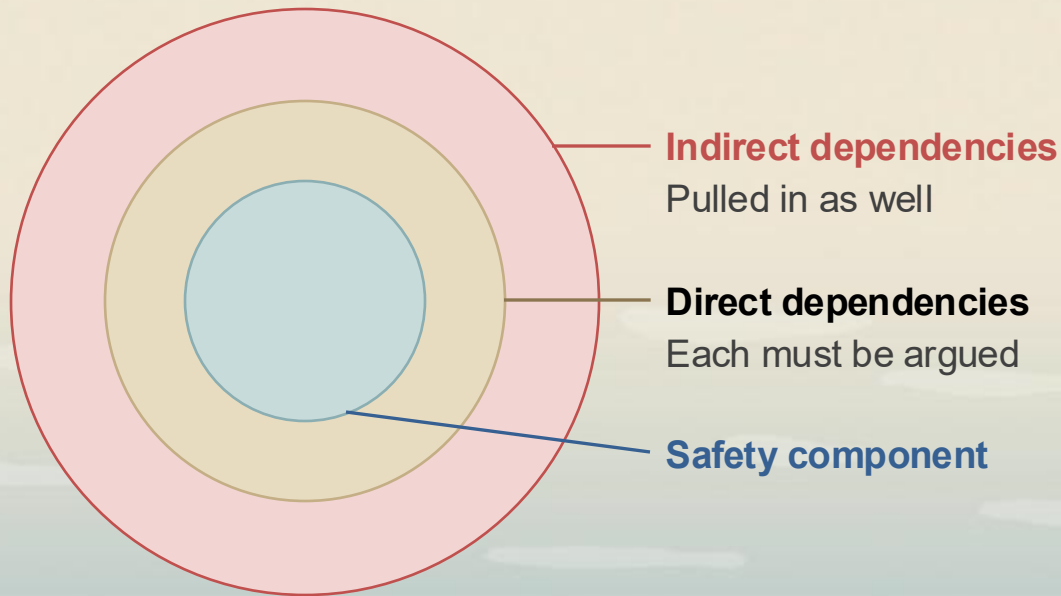
LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# One Component Pulls In Many

Defending one component pulls in practically everything else.



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Conclusions

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Where Linux Stands

Linux was not built to a safety standard, and **it shows**.

## Not safety arguments

Useful, but none is sufficient alone

- Quality, correctness, availability, **field experience**.
- Kernel process, **config flags**, **PREEMPT\_RT**, **cgroups**, security.

## Monolithic by design

Everything can interfere with everything

- One address space: **no barriers** inside the kernel or towards processes.
- Reuse pulls **practically every component** into the safety scope.

**Useful features do not add up to a safety case.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# What It Takes

A safety claim must be **justified, verifiable and complete**.

## Know the system

Before judging what is reasonable

- Understand the **kernel architecture** in detail.
- Argue **every dependency** of each safety-relevant component.

## Prove completeness

Mechanisms, not process alone

- Catch interference **where it lands**, with isolated monitors.
- Tests must be **targeted and repeatable**.

**Identify the gaps first, then close them together.**



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Safety Is Targeted

Safety mechanisms are **expensive** and **focused on use cases**.

## Expensive

Safety does not come for free

- Every mechanism takes **effort**.
- Expecting **all drivers** to become safe **without effort** is not reasonable.

## Focused

Driven by the use case

- Mechanisms target **specific use cases**.
- Making every driver safe is **usually not necessary**.

**Make safe what the use case needs, not every driver.**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# The Question

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# How to move forward?

**Given the need for preserving Qualifiability,  
how to fit Functional Safety in the Linux kernel?**



LINUX  
PLUMBERS CONFERENCE PRAGUE, CZECHIA / OCT. 5-7, 2026





# Cheat sheet

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026





# Linux as a starting point

1. Linux was not developed to a functional safety standard: it has no safety requirements and no safety case, and its process targets overall quality, not safety.
2. There is no single “Linux”: versions, patches, workloads and operating conditions differ, so field experience does not transfer.
3. Quality, functional correctness and availability are not safety: on safety-relevant paths, every failure must be handled safely and reliably.
4. Failures are not only bugs but also design–usage gaps: the kernel architecture must be understood well enough to check it fits the assumptions of use.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

 NVIDIA





# Interference within the kernel

5. In a monolithic kernel everything can interfere with everything: there is no naturally isolated subset for safety to focus on.
6. All kernel code shares one privilege level and one address space: no hardware-enforced barrier protects writable data (e.g. a driver use-after-free can overwrite page tables).
7. Interference is temporal as well as spatial: delays, wrong ordering and missed deadlines, not only corrupted data.
8. Shared components (e.g. locks, allocators) can cause common cause failures: one fault hits several safety-relevant elements at once.





# Reaching user space

9. Through the linear map, the kernel can write to the memory of any user-space process.
10. Any process can interfere with a safety-relevant process indirectly, through the kernel (e.g. a memory management bug corrupting another process's memory).
11. In a mixed-criticality system, all unqualified code is a potential source of interference, and exposure grows with how often it runs: device drivers (especially those unrelated to safety), syscalls, memory allocation and release, storage and network I/O, LSM hooks, cgroup checks, and the non-safety workloads that trigger them.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Dependencies

12. Interference can cascade through components qualified at unit level: unit qualification does not cover what the rest of the kernel does to them.
13. Reusing a kernel component brings in its dependencies: each must be argued too, and defending one pulls in practically everything else.
14. A monitor or watchdog that a safety claim relies on must be isolated from the faults it detects, and have the same or a higher level of safety integrity.
15. An external monitor is shielded from kernel interference, but it only sees what the kernel exposes: exposing every relevant symbol or event is unfeasible by number, or impractical by overhead.
16. A safety mechanism must stay isolated from the rest of the kernel over time: upstream rework can add an unwanted dependency and bring in common cause failure, unless a safeguard prevents it.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Limits of existing features

17. Configuration shapes the kernel, not its safety: Kconfig exposes what its authors envisioned, and defaults reflect their choices.
18. PREEMPT\_RT reduces worst-case scheduling latency, but is not a safety mechanism: it neither detects nor controls faults.
19. Security features overlap only partly with safety: they defend a perimeter, while safety-relevant faults start inside.
20. cgroups, containers and SELinux mitigate only simple forms of interference, and are unqualified code themselves (e.g. cgroups do not charge all kernel-internal allocations).
21. Kernel allocation is lazy and may overcommit: memory granted is not necessarily backed, and when it is really assigned is hard to predict.



LINUX

PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



NVIDIA





# Catching interference

- 22.** Safety mechanisms must be shown to be both effective and complete; process-based measures can only be verified indirectly.
- 23.** No generic mechanism detects or prevents complex systematic corruption of arbitrary writable memory.
- 24.** Hardware features are not a catch-all (e.g. ECC memory corrects bit flips, but does not stop software from writing wrong data).
- 25.** With many possible sources, interference can only be reliably modelled and detected in the receiving context.
- 26.** Detecting interference does not, by itself, control or manage availability.





# Testing

- 26. Systematic interference is so complex that estimating the probability of a failure is unrealistic.
- 27. Stress testing alone cannot support safety claims, except perhaps in very simple cases: code coverage shows which branches ran, not the call paths that reached them.
- 28. Safety claims involving the kernel need positive, requirements-based tests and focused, analysis-led negative tests.
- 29. Preemption changes interleavings and can expose latent races, so PREEMPT\_RT calls for more thorough testing and analysis.
- 30. Randomisation-based security features (e.g. structure layout randomisation) reduce the repeatability of testing.





# Thank You!

---

LINUX  
PLUMBERS  
CONFERENCE

PRAGUE, CZECHIA  
OCT. 5-7, 2026

