東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

Achieving ASIL B Qualified Linux while minimizing expectations from upstream kernel

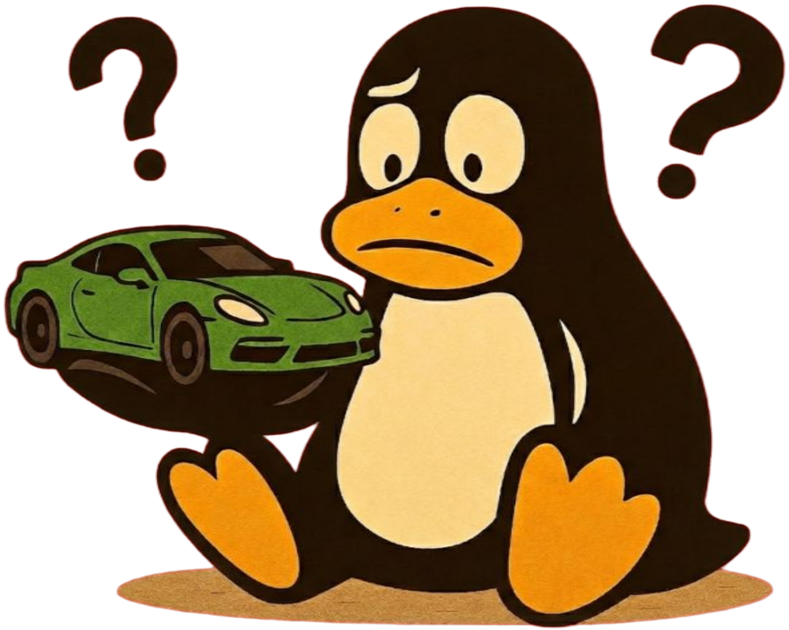
(Condensed Edition)

Igor Stoppa <istoppa@nvidia.com>

東京 2025
LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

NVIDIA.

The Question



“How to merge our code upstream, while preserving value for our users?”

The value:

Ease of Qualification of the Linux kernel to ASILB rating (According to ISO 26262)

Empirically verifiable safety cases
(with standard techniques)

Who are the users

Vehicles Manufacturers

Robotics Manufacturers

Industrial/Civilian safety providers

(e.g. wildfire monitoring)

Retaining ease of qualification

We add **safety mechanisms** to the kernel that:

1. become the **target** for safety requirements
2. allow most of the kernel to **stay unqualified**

The mechanisms are mostly self contained
<u>And they can be compiled OFF</u>

All of their kernel dependencies must be qualified

It become rapidly impossible, if not done carefully

Example of unfeasible refactoring

We implement a specialized Address Space Isolation

We should try to align with ongoing efforts, right?

At least at API level, right?

Wrong!

Why the refactoring is unfeasible

Allowing a function to **specify** (e.g. like with GFP)
the **safety level** of the memory it requests,
it means we should **safety qualify** said function.

And the functions should still determine if it is being invoked for safety purposes or not.

E.g.: allocating process pages.
It might be a safe process.
Or not.

What we do instead

threads are assigned **capabilities**

functions are **tagged** to request **capabilities**
from the thread running them

*The **safety level** of the memory returned to a function is the **convolution** of the **tag** it has received with the **capabilities** of the thread executing it*

How does the convolution work?

Capabilities of a thread (and their **state** of activation) are **stored** (with redundancy) in its **task struct**

states are used when a memory manager needs to **select** the memory to return

the memory returned is **vetted end-to-end** for correctness (linear/virtually linear & physical)

the memory manager remains unaware of safety

How does the vetting work?

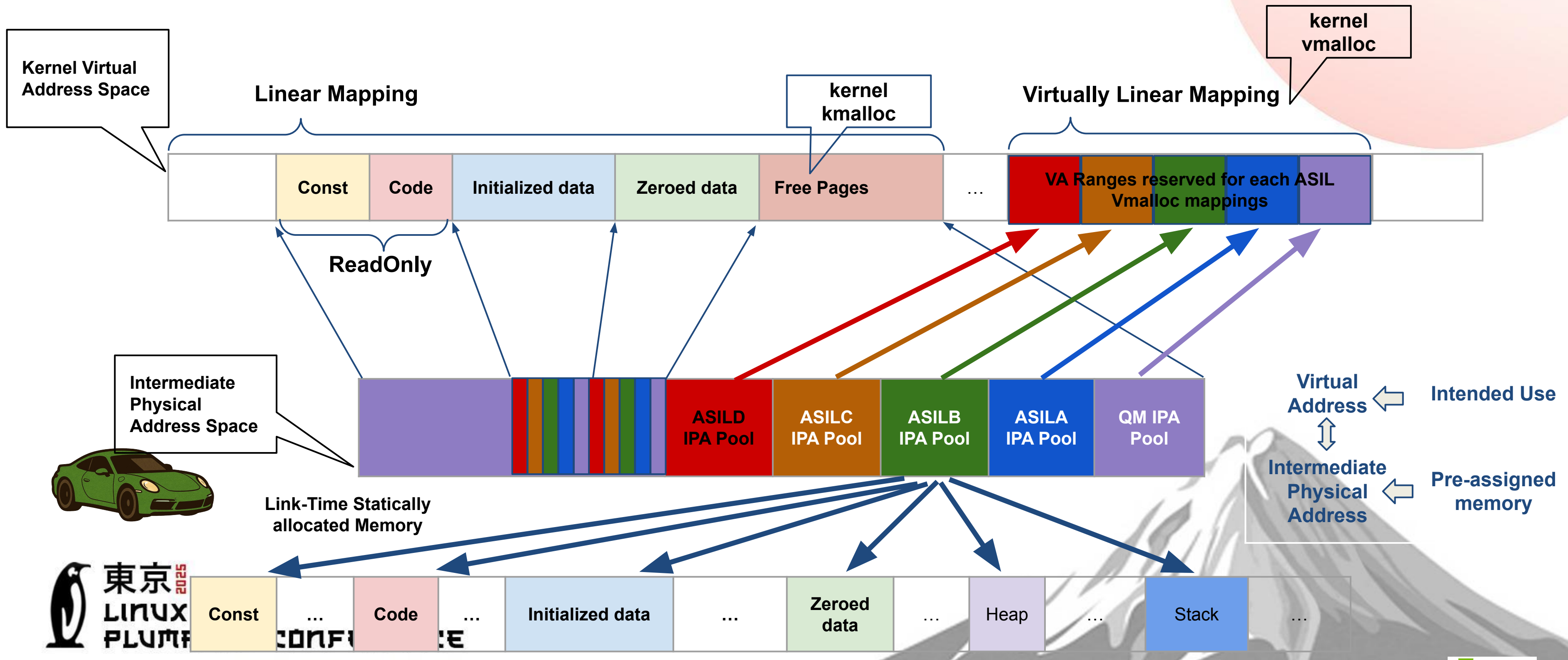
Resources are preventively subdivided into **pools**,
one pool for resource and for each **safety level**.

Homogeneous resources belong to the same range.

Memory example, page level allocations:
1st level: does the **adress** belong to the right **range**
2nd level: **pfns** are tracked through **redundant bitmaps**
3rd level: virtually linear addresses must not be in use (**no pte**)

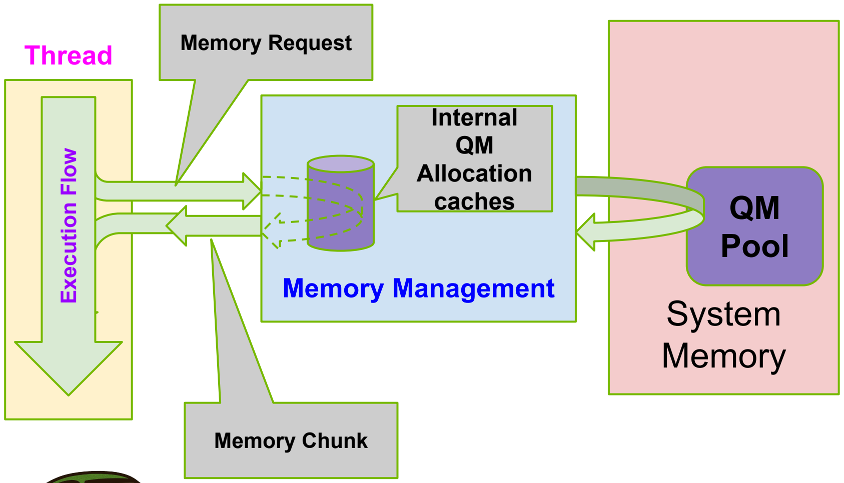
Multi-ctx partitioning of Address Spaces

Simplified View

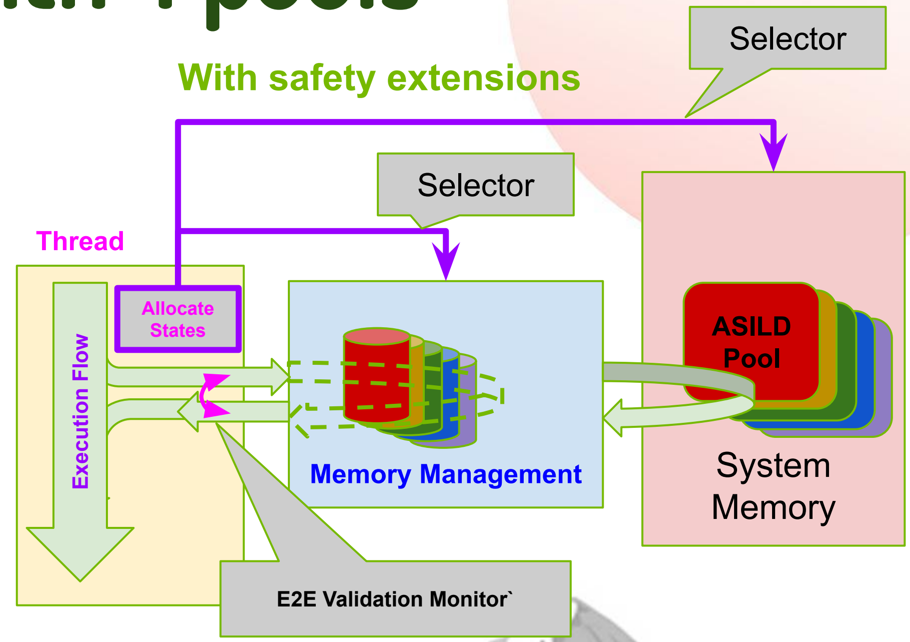


Example with 4 pools

With Vanilla Kernel



With safety extensions



Why we use these pools?

Pools are associated with **safety levels**.

A pool of a given safety level is **not writable** by threads from **lower safety levels.**

We create **mutiple** kernel **ad-hoc** memory **maps**

Example with 4 contexts

From \ To	QM Context	ASIL A Context	ASIL B Context	ASIL C Context	ASIL D Context
QM Context	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL A Context	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL B Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL C Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES
ASIL D Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES

What we do with the map?

Memory **maps** are used **independently** per **core**

The **map** on a **core** depends on the **thread.**

The **state** of the **thread** determines the **map**

The **state** derives from the **convolution**
between thread capability and
function required capability.

Example with 3 contexts, 9 cores

Core 1 User_i	Core 4 Thread_n	Core 7 Thread_x
Core 2 Thread_j	Core 5 User_o	Core 8 Thread_y
Core 3 User_k	Core 6 Thread_p	Core 9 Thread_z

User_i	QM
Thread_j	QM
User_k	ASIL B
Thread_n	ASIL A
User_o	ASIL A
Thread_p	ASIL A
Thread_x	ASIL B
Thread_y	QM
Thread_z	QM

Example with 3 contexts, 7 threads

User Space	P1 Physical Pages from ASIL A Pool	P2 Physical Pages from QM Pool	P3 Physical Pages from ASIL A Pool	P4 Physical Pages from ASIL B Pool			
Kernel Space	kernel mode ASIL A Memory Map	kernel mode QM Memory Map	kernel mode ASIL A Memory Map	kernel mode ASIL B Memory Map	kernel thread QM Memory Map	kernel thread ASIL B Memory Map	kernel thread ASIL A Memory Map

What is the purpose of all of this?

We can empyrically confirm that:

1. We got the memory from the right pool
2. The memory is write protected from lesser pools
3. The memory managers involved were NOT safe

"Ease of qualification and empyrical verification"

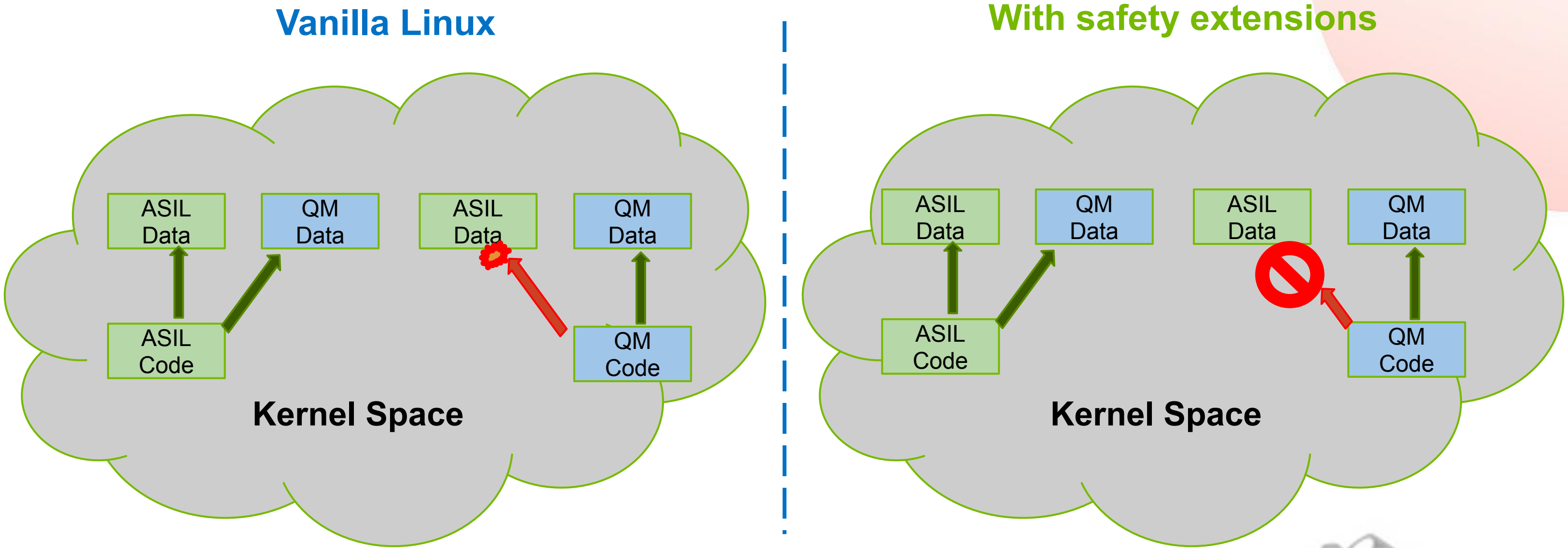
What is the impact on a mantainer?

Repeating: ALL of this can be compiled OFF

Changes to the code:

1. introduction of tags for certain functions
2. some data structures become arrays of their former self
3. some variables/fields must be declared/set/get through macros, for redundancy.
4. some components (e.g. interrupts) need pre/post hooks

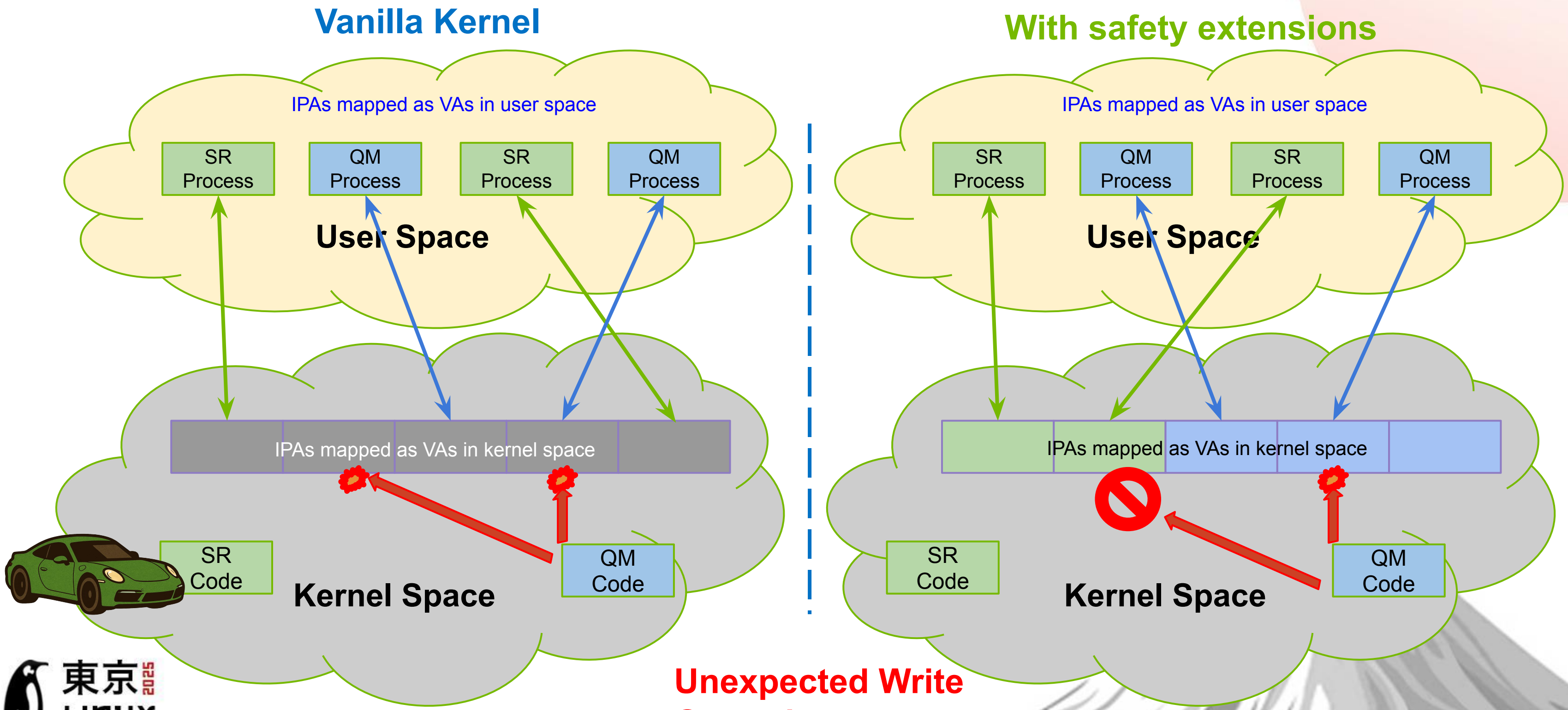
Example of in-Kernel FFI



Legitimate Write Operation

Unexpected Write Operation (Interference)

Example of Kernel-userspace FFI



Unexpected Write Operation (Interference)

What does this bring to upstream?

1. Highest levels of safety qualification possible
2. Not very invasive changes to the code.
3. No burden of changing kernel processes
4. No need for extensive documentation/description for the vast majority of kernel sources.

Answers/Comments/Questions?

The End

(But it's not over, keep scrolling for more)

東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

Achieving ASIL B Qualified Linux while minimizing expectations from upstream kernel

(Unabridged Edition)

Igor Stoppa <istoppa@nvidia.com>

東京 2025 Linux Plumbers Conference
TOKYO, JAPAN / DEC. 11-13, 2025

NVIDIA

Disclaimers

The following explanations will sometimes trade accurate use of Functional Safety terminology for a higher accessibility.

It will also focus on the specific case of the Linux Kernel.

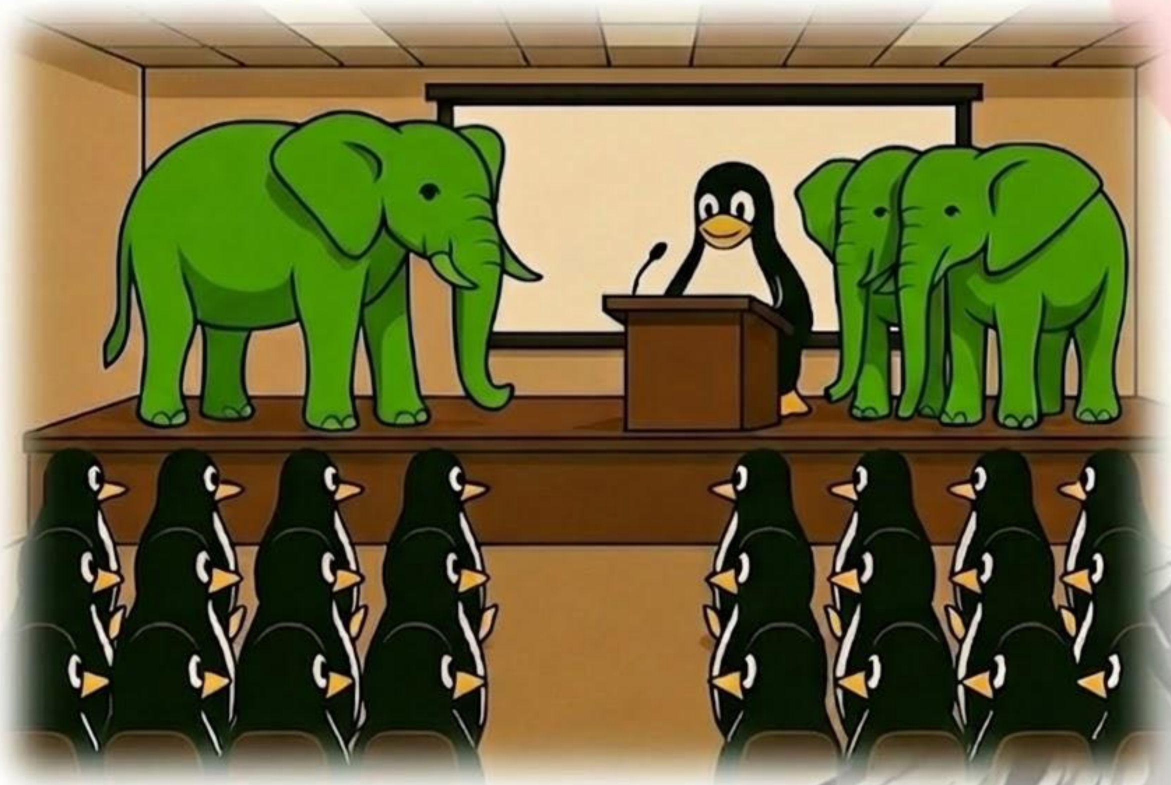
ARM64 as reference HW platform

Please hold back the nitpicking ;-)

Spoilers

- A long-ish Introduction
- Typical Puzzled Replies
- The search for Safe Linux
- A Verifiably-Safe Linux
- **The Question, Redux**

The elephant(s) in the room



Rumor #1: Custom HW is indispensable

Reality: ***ARMv8.2 or better*** needed for advanced TLB management (available since Q1 2018)

Rumor #2: Closed source SW required

Reality:

- Minimal dependency on:
    - Hypervisor (e.g. *protecting EL1 registers*)
    - UserSpace (e.g. *invoking ad-hoc syscalls*)
- ***Non-Kernel dependencies will be published***
  (e.g. using a reference FOSS hypervisor)

Rumor #3: Kernel version stuck in time

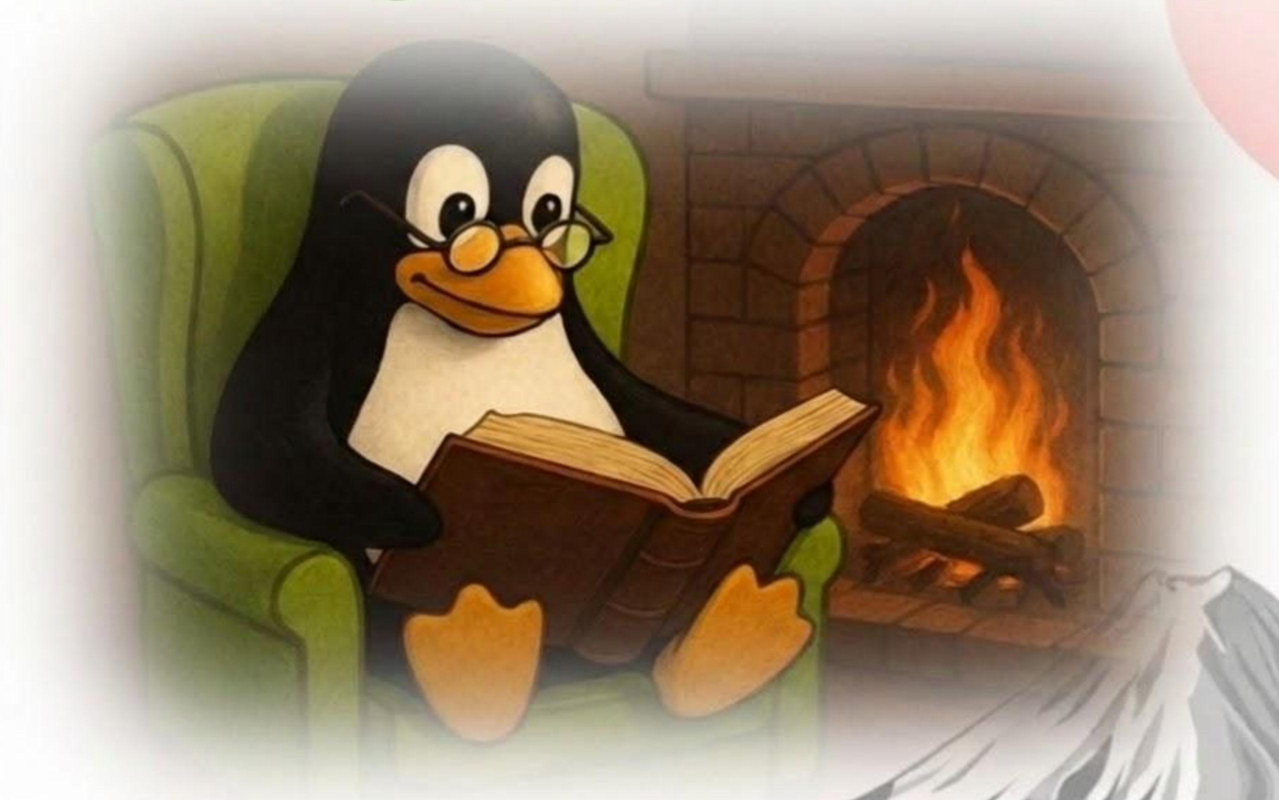
Reality: closely tracking upstream
Presently **running on *v6.18***
Continuously rebasing on ToT

Rumor #4: It's a Drop-The-Code & Run

Reality: We have a ***business*** interest.

Many other companies do,
they would step in,
if we run away.

A long-ish Introduction



The User Value

Ease of Qualification of the Linux kernel to ASILB rating, and beyond.
(According to ISO 26262)

Safety Levels

	Automotive Safety	Generic Safety
Highest, strictest	ASILD	SIL4
	ASILC	SIL3
	ASILB	SIL2
Lowest, relaxed	ASILA	SIL1
Quality Managed	QM	QM

NOTE: not a direct ASIL-to-SIL mapping

What it means, in practice

Mainly, it means to achieve:

- Absence and Control of Systematic Failures
- Freedom From Interference (FFI)

Absence and Control of Systematic Failures

In layman terms:

Avoiding and managing mistakes in:
design/implementation/
manufacturing/configration

FFI: what is “Interference”

Undesired influence on a SW component, preventing it from performing **promptly** its roles, safety included.

E.g. the brake doesn’t work promptly

What achieving FFI means

*"Prompt rejection or detection,
or avoidance by design,
of deviations from
safe operative conditions,
either in a component
or in a system."*

In layman terms

“Support of safety features cannot stop silently, for longer than a given time.”

Achieving FFI

Option 1: The Pure Process Path

“Argue that the processes used are safe.”
(e.g. FFI achieved by design and validation)

*Relatively easy and **apparently** inexpensive to attempt.*
Focuses more on the processes than the content.
Doesn’t generate high confidence.
Little kernel expertise required.
Excessively subjective.
Insufficient for higher safety levels

“Argue that the processes used are safe.”

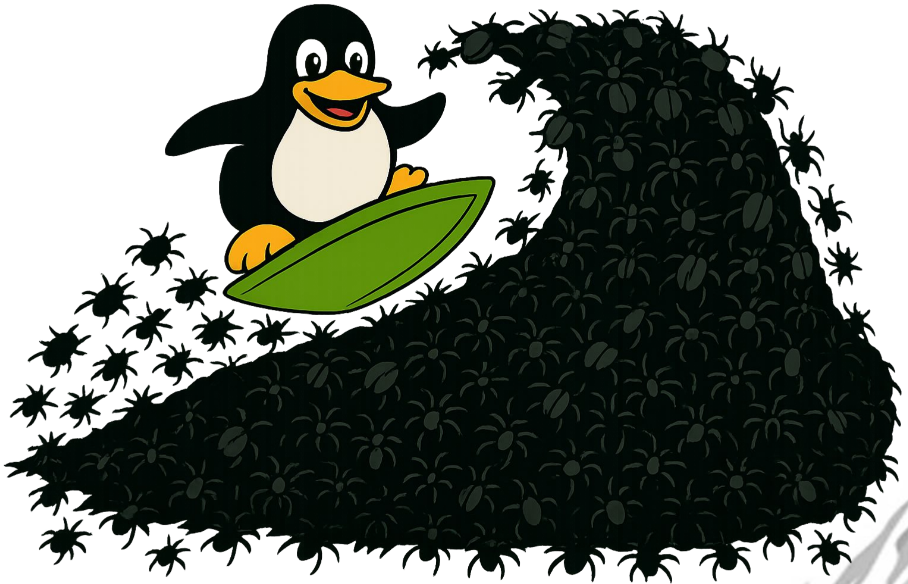
Achieving FFI

Option 2: The Hardening Path

“Implement Hardening, and validate it through (negative) testing”

*Difficult and **apparently** expensive to attempt.*
Brings very high confidence.
Extensive kernel expertise required.
Highly objective.
Enables higher safety levels

“Implement Hardening.”



The differences between the two paths

The Hardening Path requires
analysing all the code involved,
including ***every function*** that could interfere.

The Process Path is akin to
assuming that classes
of bugs are absent.

The problem with using Linux

The Linux Kernel is NOT safe.

Not in the way defined by safety standards.

Why Linux is not safe (per ISO 26262)

Process Perspective

It is not developed according to safety standards.

e.g. implementing without formalized requirements

Why Linux is not safe (per ISO 26262)

Functional Perspective

Linux is a monolithic kernel

No barriers to interference between components.
Possible "domino effect" (Cascaded Interference)

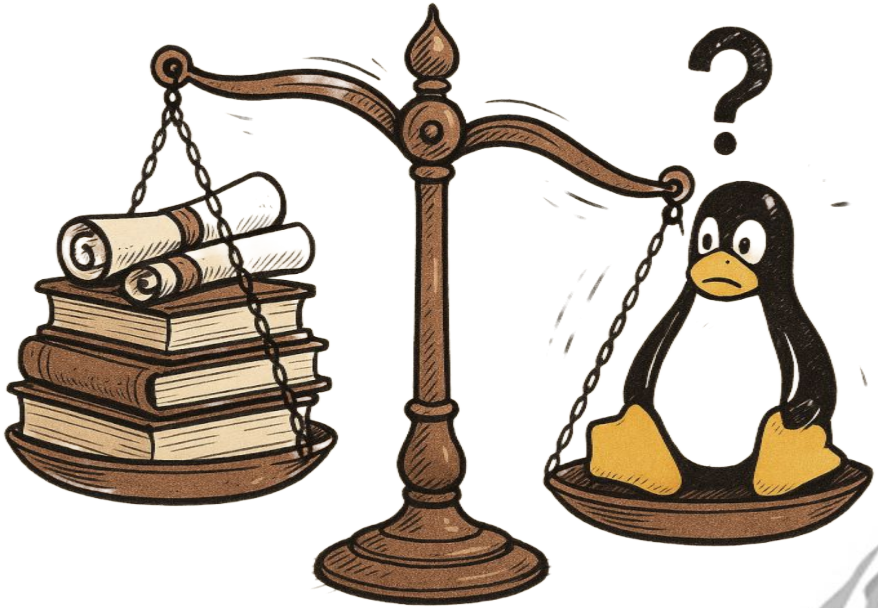
Making Linux **verifiably** safe

ISO26262 methods are impractical.

If applied mechanically.

The entire code base should be qualified.

“The entire code base should be qualified”



Making Linux **verifiably** safe

Our approach:

Independent, yet integrated, hardening mechanisms

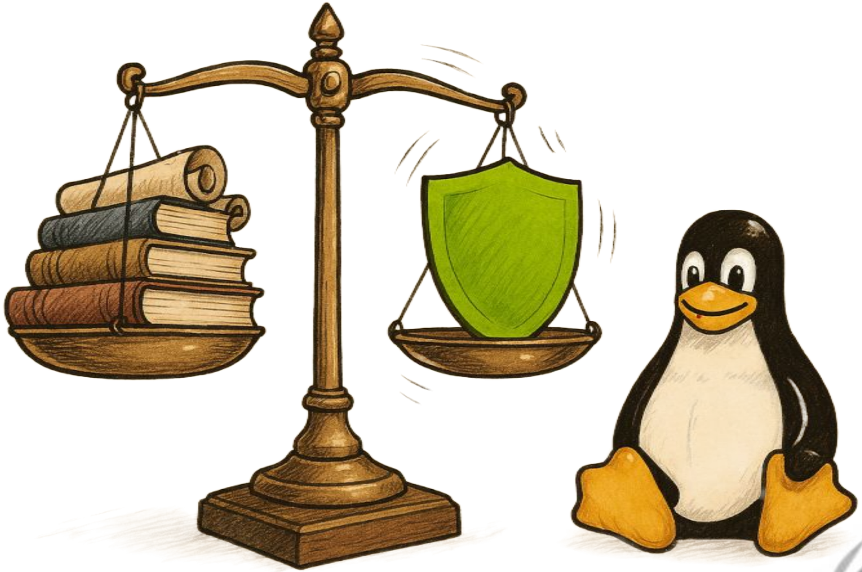
Benefits

Drastically reduced *need for qualification of parts of the Linux Kernel*

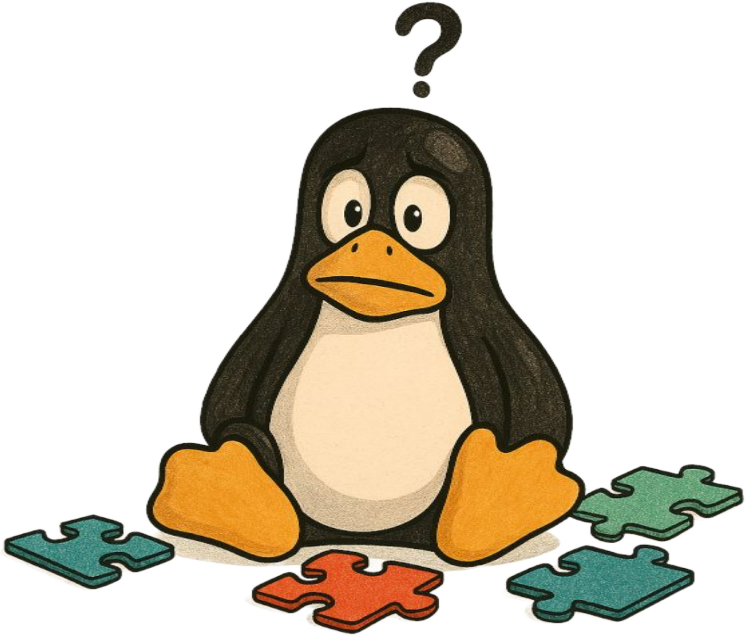
The upstream kernel can remain non-safe

Ease of tracking upstream

"Independent yet integrated hardening mechanisms"



Typical Puzzled Replies



Why insisting, despite it not being safe

Strong demand from manufacturers:

- It is open - no vendor lock-in
- Availability of developers
- Ease of development and integration

“But I think that …”

- “ … Linux must be already good enough …”
- “ … you should use …
  - Rust
  - A watchdog
  - An hypervisor
  - Realtime features
  - <*Insert Security Feature here*>
- … company XYZ has a solution for you …
- … you should instead use a specialised Operating System (we do it already)

Why those ideas are insufficient

see: Open Source Summit Europe 2025

Identifying Safety Weaknesses and Fault Propagation in the Linux Kernel

https://tinyurl.com/UnsafeKernel

Why not using some specialized OS

We *already* have a safe product that doesn't use Linux.

Yet, many customers *still* want a *SAFE Linux* product.

"Just file a bug report, it will be fixed"

High number of units and operating hours.

Even extremely improbable bugs **will** happen.

When they happen, **life & limb** are at risk.

Safety cannot be reactive.

“Why didn’t you discuss it sooner?”

The problem itself was not clear
It was not even clear which questions to ask

The Search for Safe Linux

Our efforts

4 years spent evaluating offerings

- We couldn't find a satisfying option.
- We even seeked for a bespoke solution.
- Nobody would satisfy our requirements.

When "Buy" is not an option,
"Make" is all that is left

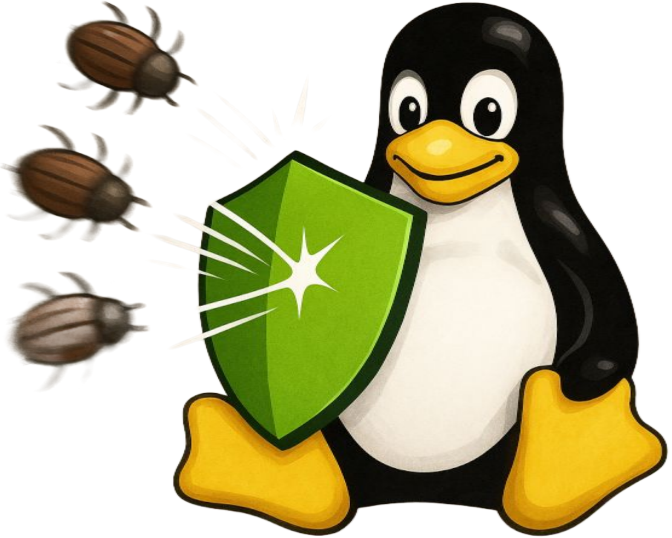
What we were looking for

- **Retention of performance parameters**
  - compared to ad-hoc safe OSes
- **Empirically Verifiable Safety Solution**
  - *simulate interference, assess response*
  - *confirm reaction/detection as expected*
- **Low exposure to upstream**
  - ***unmodified upstream processes***
  - *ease to rebase/backport patches*
  - *little re-certification efforts*
- **No vendor lock-in**

What we rejected

- **Unverifiable Approaches**
  - *unable to perform negative testing*
  - *arbitrarily defined “confidence”*
  - *based on chances and probability*
  - *relying on non-deterministic testing*
  - *discovering that “luckily” everything is already safe (but only from that vendor)*
- **Incomplete approaches**
  - *only some temporal (no spatial) protection*
  - *arbitrary selection of components*
- **Alien approaches**
  - hypervisor hijacking kernel functionality

A Verifiably-Safe Linux



What we want

- identify safety requirements (physical constraints)
- neither device drivers nor processes involved with safety requirements can fail silently
- use ad-hoc verifiable mechanisms, to provide safety
- restrict requirement to few simple kernel functions

The mostly-non-safe Linux Kernel is now the threat

Benefits of an unsafe kernel

- **The kernel can be updated frequently**
- ***<u>No need to change upstream processes</u>***
- **The safety mechanisms take upon themselves:**
  - **safety requirement for the whole system**
  - **the burden of qualification**

But this only transforms the problem …

Coping with an evolving unsafe kernel

“The kernel can be updated frequently”

Constraints on the safety mechanisms:

- the patchset must be **VERY non-invasive**
  to avoid lots of churn at every rebase
- design and implementation must be **VERY stable**
  to avoid the need for frequent re-certifications

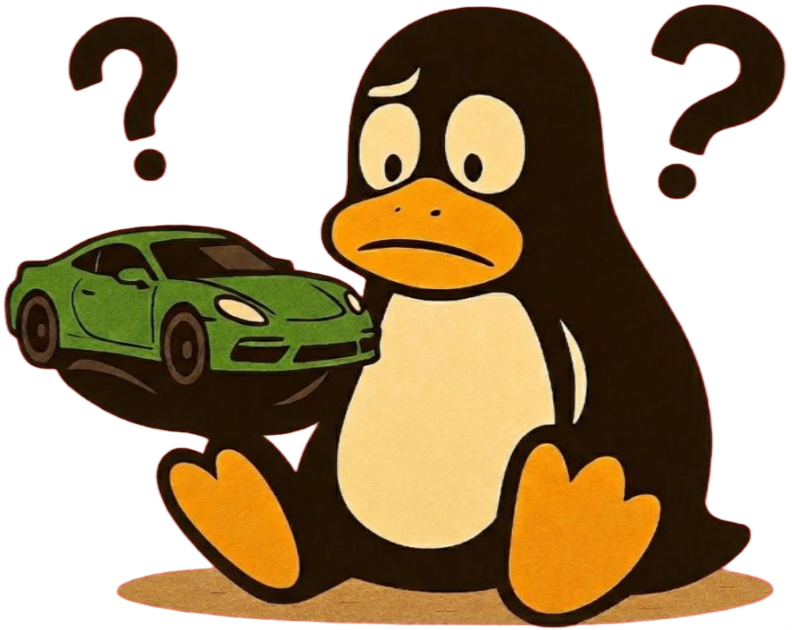
Chibi safety concept

- identify **memory/addresses** relevant to safety (e.g. kernel data & mmio, process pages, irq states)
- either **protect** it (MMU & SMMU) or use **redundancy**
- create **chains** of monitored activity (***multi level WD***) internal states: (irqs, tasks, processes, etc.)

Chibi Safety Mechanisms

- HW-based resources isolation between cores in EL1
- Safe association of **hierarchical "capabilities"** with:
  threads of execution, functions, resources
- HW-enforced resources allocation and access permission
  through ***convolution of capabilities***
  (can "subject" perform "action" on "target"?)
- *Context-Based Safe Interprocess Communication*
- Per-context SW-WD channel for:
  irqs, threads, processes, user threads

The Question Redux



"How to merge our code upstream, while preserving value for our users?"

Safety mechanisms must stay firmly decoupled from most of the kernel.

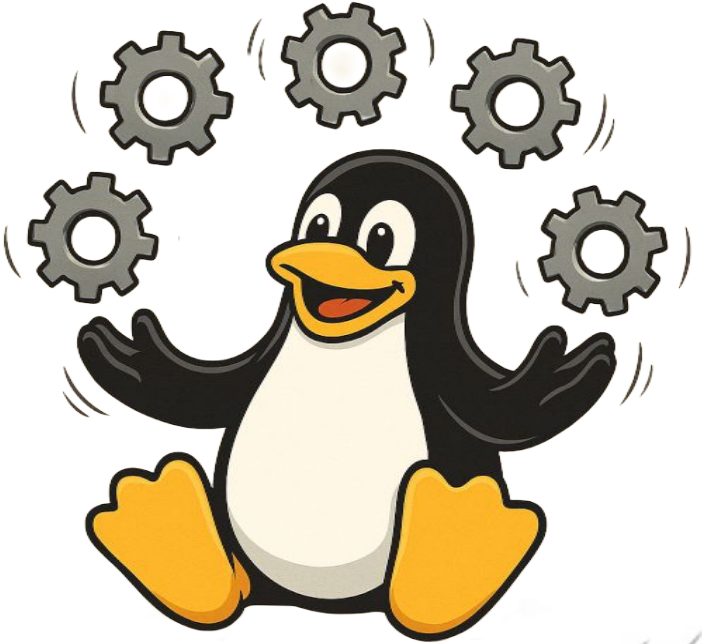
In practice it means:

- *allowing for function tagging (or someone proposing a valid alternative)*
- *introduce several hooks in key subsystems*
- *turn into arrays some core data structures (they go back to 1-element, with safety OFF)*
- *declare and access certain variables through macros, to implement safety through redundancy*

What's in it for the upstream community

- *some features can be useful for security and even for performance improvement (as long as they remain usable for safety)*
- *mounting demand for "safe linux":*
  - ***no need to change kernel processes***
  - ***no need for formal requirements***
- ***the safety extensions can reach safety levels otherwise unattainable***

Safety Mechanisms



What are these mechanisms in practice?

The following are some of the more basic mechanisms implemented.

The purpose of the description is to exemplify the concepts, rather than a full deep-dive.

Resources Access Control, HW perspective

Multiple <u>Contexts:</u>

Each context is comprised of:

- ***own kernel (EL1) memory map***
- ***pools of resources:***
    - linear mapping range
    - virtually linear mapping range
    - intermediate physical addresses range
    - mmio range (BAR, etc,)
    - irqs
    - ...

Resources Access Control, HW perspective

Per-context kernel memory maps enable:

- *per-context write-access control for the same address (including mmio resources)*
- *hierarchical access permissions*

In practice:

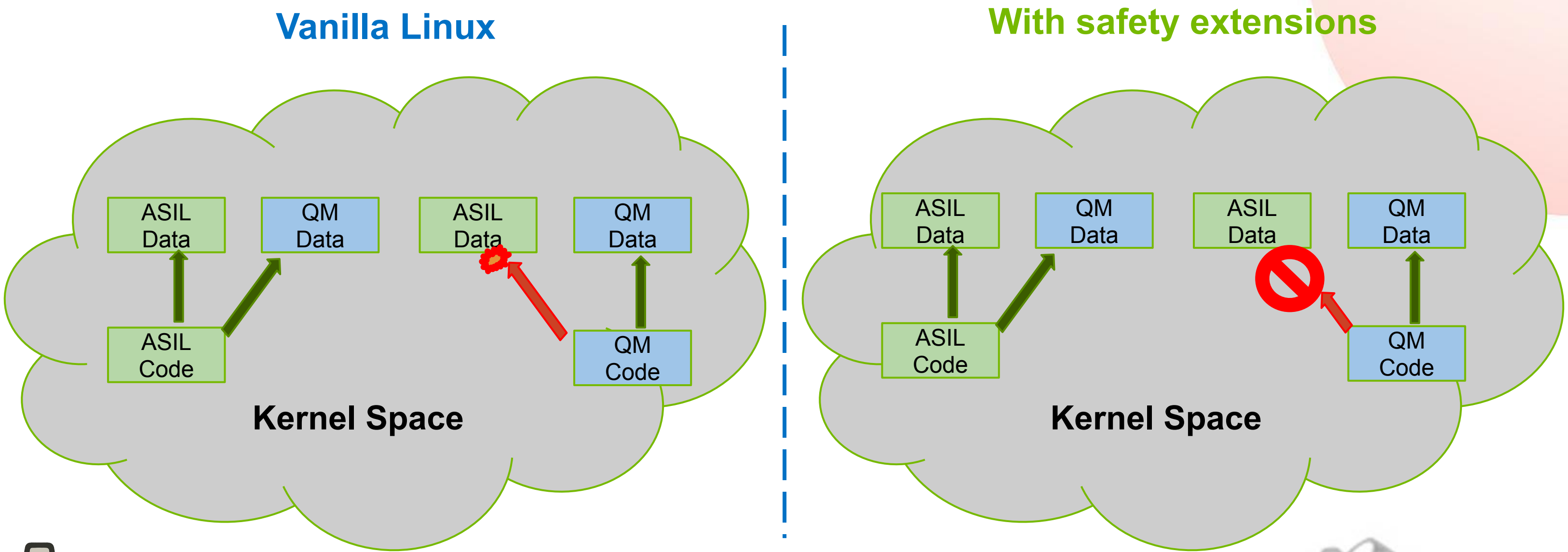
A less safe context can read from a safer one, but it cannot alter it

A safer context can both read from and write to a less safe one

Example with 4 contexts

From \ To	QM Context	ASIL A Context	ASIL B Context	ASIL C Context	ASIL D Context
QM Context	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL A Context	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL B Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES	Write: NO Read: YES
ASIL C Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: NO Read: YES
ASIL D Context	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES	Write: YES Read: YES

Example of in-Kernel FFI



Legitimate Write Operation

Unexpected Write Operation (Interference)

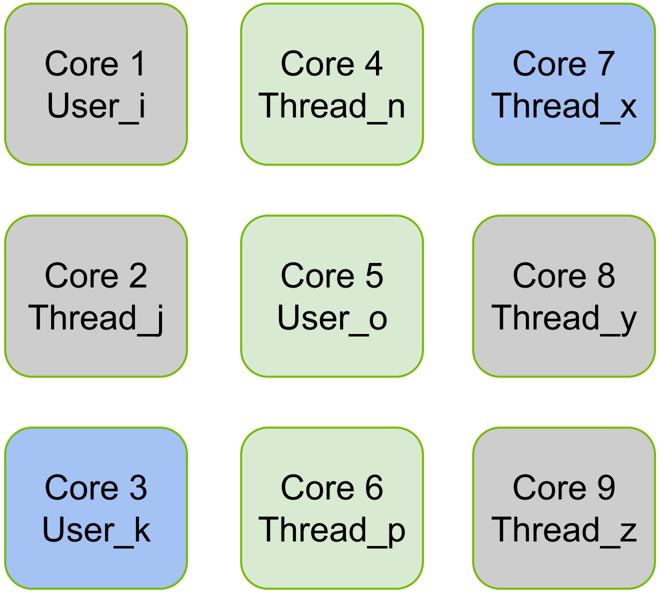
Resources Access Control, HW perspective

Independent per-core context activation:

Safe simultaneous execution of kernel threads with different safety ratings.

A less safe thread running on one core cannot alter resources that are writable by a safer thread running on another core.

Example with 3 contexts, 9 cores



User_i	QM
Thread_j	QM
User_k	ASIL B
Thread_n	ASIL A
User_o	ASIL A
Thread_p	ASIL A
Thread_x	ASIL B
Thread_y	QM
Thread_z	QM

Resources Access Control, HW perspective

Per-thread dynamic context activation:

A thread can transition between contexts depending on the safety rating of the code it is presently executing.

A thread can always perform read operations, in any context.
Higher contexts are needed only for write operations on safe(r) data

Resources Access Control, SW perspective

- ***different permissions*** for individual kernel threads
- allow only transitions to ***more restrictive contexts***
- only ***selected functions*** allowed to alter safe data
- altering safe data requires ***both*** a thread with sufficient permission, and a selected functions
- user space processes as ***extension*** of kernel threads
- syscalls, ioctls, interrupts are treated ***in the same way***

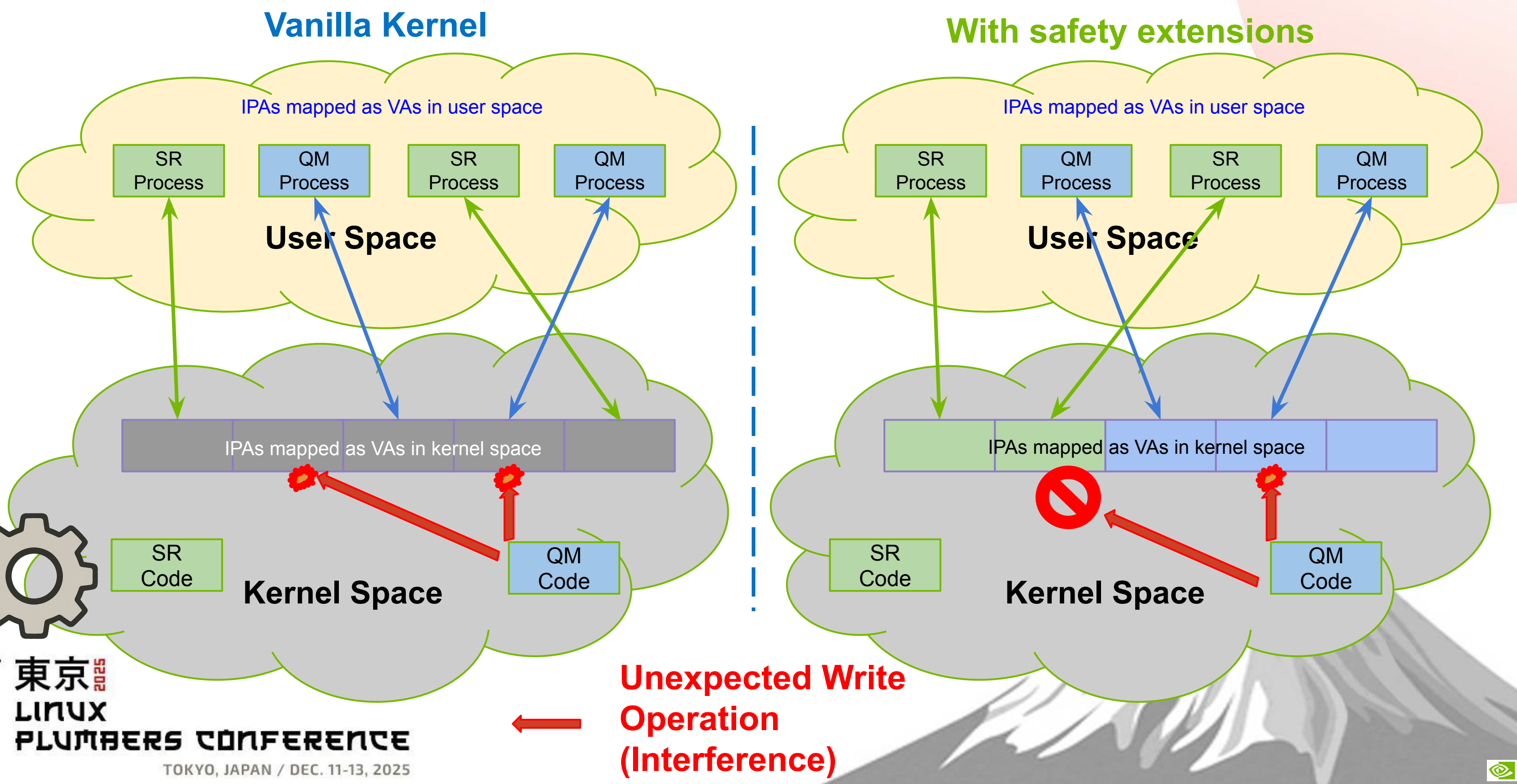
Resources Access Control, Memory Pools

- A thread allocating resources from a given pool ***must have matching capability***
- Kernel memory managers ***do not need to be safe***, the memory they return can always be vetted
  - Given a memory location, its pool can be ***inferred***
  - Ad-hoc mechanisms detect ***double allocations***
- Parts of EL1 Page Table pages supporting pools are ***coherently protected*** to compatible level
- So are EL0 page tables of protected processes

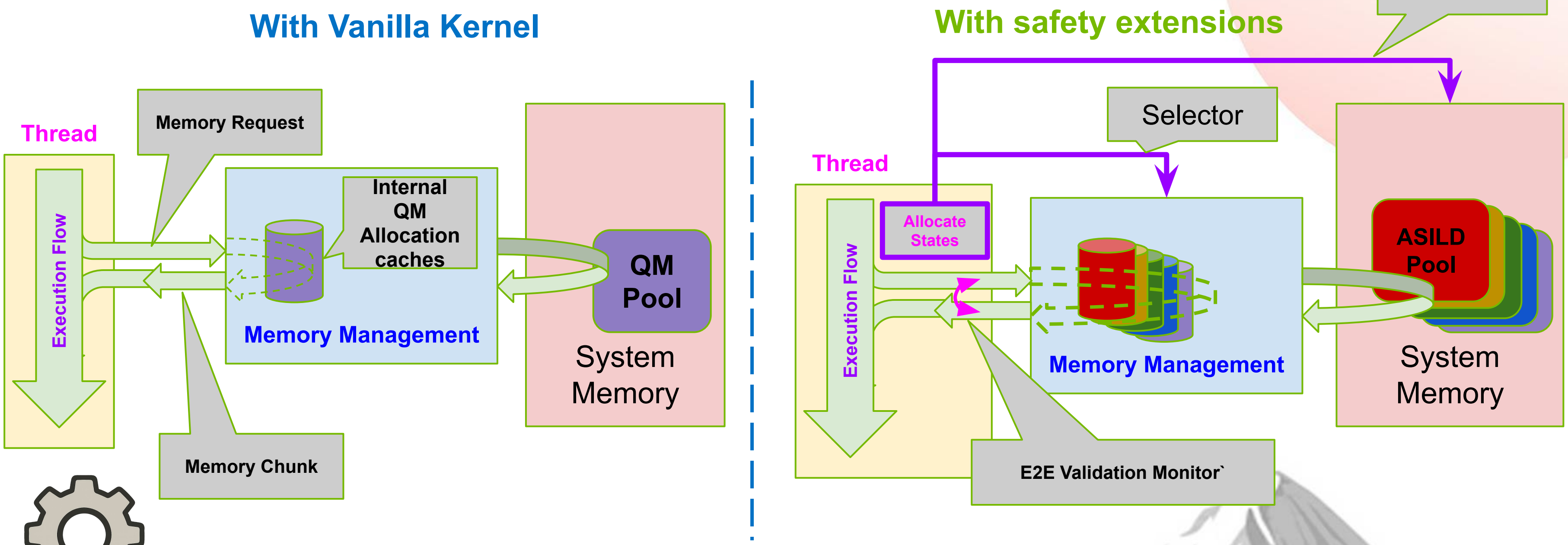
Example with 3 contexts, 7 threads

User Space	P1 Physical Pages from ASIL A Pool	P2 Physical Pages from QM Pool	P3 Physical Pages from ASIL A Pool	P4 Physical Pages from ASIL B Pool			
Kernel Space	kernel mode ASIL A Memory Map	kernel mode QM Memory Map	kernel mode ASIL A Memory Map	kernel mode ASIL B Memory Map	kernel thread QM Memory Map	kernel thread ASIL B Memory Map	kernel thread ASIL A Memory Map

Example of Kernel-userspace FFI



Example with 4 pools



What shown so far is just a fraction …

… but it is clear that it is doing things that are unusual in Linux.

And that's not all of it ...

***... it's also how they are implemented,
e.g. through use of function tags.***

***They control which state management code
is injected in specific functions,
by a compiler plugin.***

The major upsides

ALL of these mechanisms can be completely disable at build time.

The resulting kernel binary doesn't incur into any penalty.

BUT EVEN WHEN ACTIVE THEY DO NOT AFFECT REGULAR NON SAFE DRIVERS/PROCESSES

The End

(For Now)

東京 2025

LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025