# LFSCS WG - Linux Virtual Address Space Safety

Alessandro Carminati
Red Hat

ELISA
Enabling Linux in Safety Applications

Aerospace · Automotive · Linux Features
Medical Devices · OS Engineering Process
Safety Architecture · Space Grade Linux · Systems · Tools

# Agenda

- Introduction & Scope
- VMA Architecture: Lifecycle & Features
- The Linear Mapping Threat
- Safety Features: Defense & Detection
- Defining the Path to Functional Safety
- Epilogue

# Introduction & Scope

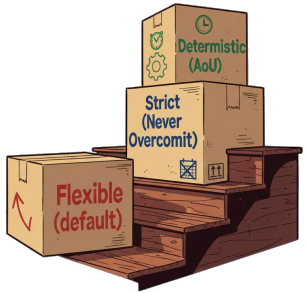

- **Introduction & Scope**
- VMA Architecture: Lifecycle & Features
- The Linear Mapping Threat
- Safety Features: Defense & Detection
- Defining the Path to Functional Safety
- Epilogue

**Linux: A Control Center Built for Efficiency… Not for Safety**

# Linux VMA Safety

## An Architectural Roadmap for Functional Safety

- **Functional Safety Requires Deterministic Isolation.** The VMA is the key mechanism for isolation and resource management in a Linux Mixed-Criticality environment.
- **Current Architecture Compromises Isolation.** The default Linux VMA design prioritizes flexibility over the determinism required for safety.
- **Defining the "Safe VMA" Architectural Roadmap.** We must identify and address:
  - Fundamental Assumptions and Risks in the VMA lifecycle.
  - Corner Cases in allocation and memory pressure handling.
  - Necessary changes to the "Software Around the VMA.



# The VMA's Critical Role

## Isolation and Resource Management in Mixed-Criticality Systems

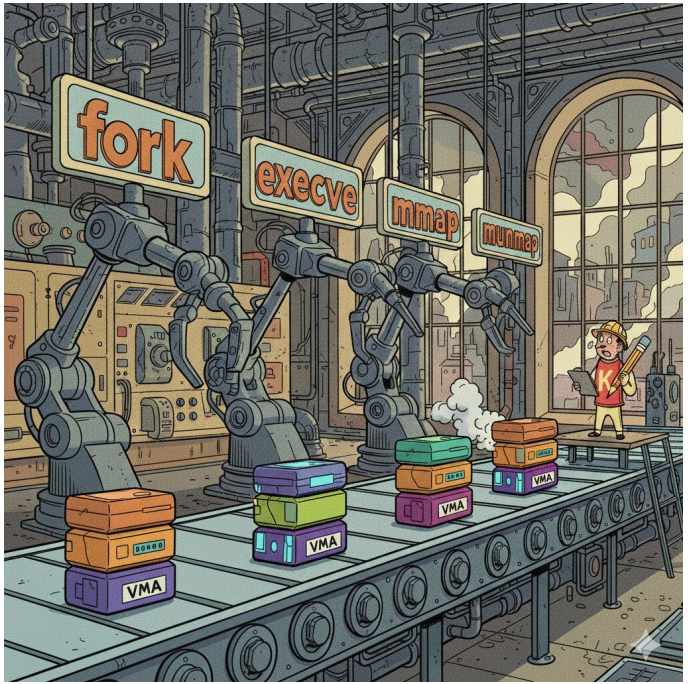
- **Heuristic Overcommit:** The Linux default, prioritizes flexibility.
- **Never Overcommit:** Safe systems prefer this policy, which restricts allocations to a defined limit to avoid unexpected failures.
- **An Architectural idea is "Allocation on Usage (AoU)":** Safer processes pre-allocate all necessary memory.

# Scope and Goal

## Defining the "Software Around the VMA" and the Functional Safety Mandate

- Strict temporal and spatial isolation.
- Deterministic resource access.
- Preventing unintended access across process boundaries.
- Guaranteeing VMA integrity against races.

# VMA Architecture: Lifecycle & Features



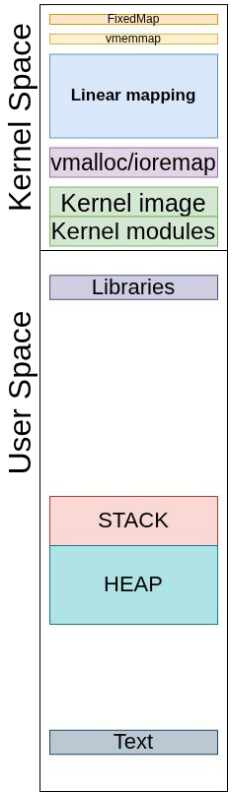

- Introduction & Scope
- **VMA Architecture: Lifecycle & Features**
- The Linear Mapping Threat
- Safety Features: Defense & Detection
- Defining the Path to Functional Safety
- Epilogue

**Where Processes Get Their Memory… One Box at a Time.**

# The VMA's Full Scope

Governing User Space **AND** Kernel Space

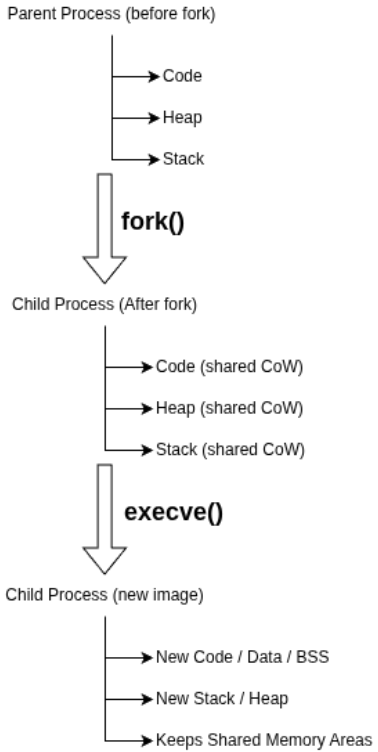
- **User VMAs:** Private, per-process address spaces.
- **Kernel VMA:** Global, shared by all processes.
- **Main kernel memory zones:**
  - **Linear Map**: the direct 1:1 mapping of physical RAM,
  - **vmalloc Area:** non-contiguous mappings for dynamic kernel allocations,
  - **Vmemmap:** metadata mapping for struct page descriptors, and
  - **Fixmap:** reserved static mappings for special kernel addresses.



# Processes VMA Lifecycle I

## Initialization (fork, execve)

- **fork():** Clone `mm_struct` & VMAs — shared pages via CoW.
- **execve():** Tear down old VMAs (except shared); rebuild text, data, heap, stack.
- **Safety note:** Shared memory can persist intentionally; defaults like `O_CLOEXEC` limit accidental carryover.
- **init process:** First process built by kernel; later rebuilt via `execve()`.
- **Kernel VMA:** Always inherited, shared by all processes.



# Processes VMA Lifecycle II

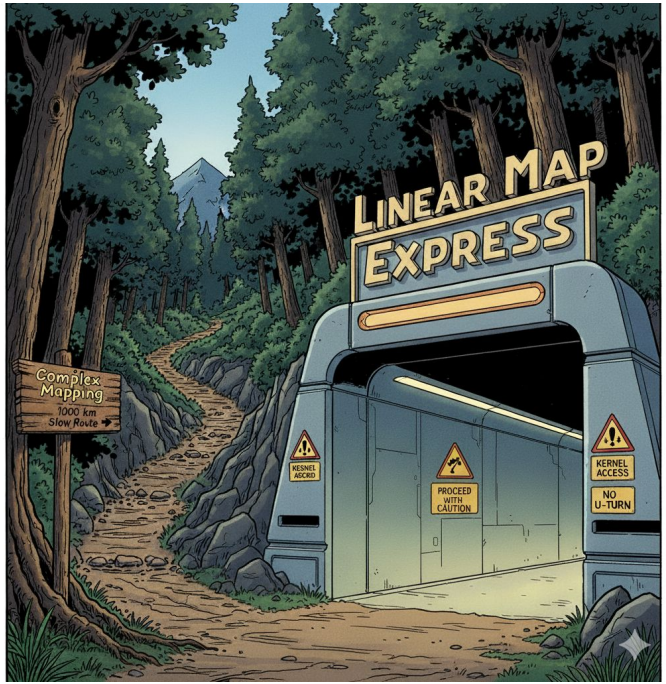
## Dynamic Allocation & Runtime Mechanisms

- **Syscalls:** `mmap()`, `munmap()`, `brk()/sbrk()`, `mprotect()`, `mremap()`
- **mlock()/mlockall():** Prevent swap-out, improve temporal determinism.
- **Risks:** Non-deterministic allocation, races, and layout shifts under load.
- **Safety Practices:**
  - Pre-allocate & lock critical pages.
  - Check all syscall results.
  - Understand mapping & overcommit behavior.

# VMA Instrumentation and Debugging Tools

- `/proc/<pid>/maps`, `/proc/<pid>/smaps`: show VMA layout and usage.
- **KASAN:** Shadow-memory detector for invalid access. Excellent for testing, but heavy, partial, and nondeterministic.
- **KFENCE:** Lightweight guard-page monitor for production, but samples only a tiny fraction of allocations.
- **Hardware Tagging (ARM64):** reduces overhead, future path for in-field safety diagnostics.

# The Linear Mapping Threat




- Introduction & Scope
- VMA Architecture: Lifecycle & Features
- **The Linear Mapping Threat**
- Safety Features: Defense & Detection
- Defining the Path to Functional Safety
- Epilogue

**A Straight Tunnel Through Complexity… but at What Safety Cost?**

# Historical Context

## The 32-bit Lowmem/Highmem Partitioning

- **4 GB total VA space:** typically split 3 GB user / 1 GB kernel (`CONFIG_VMSPLIT_*`).
- **Lowmem:** Permanently mapped kernel region.
- **Highmem:** Unmapped RAM, accessed only via temporary mappings.
- **Accidental Safety:** Kernel couldn’t touch all RAM at once… Natural containment.

# The Linear Mapping Threat I

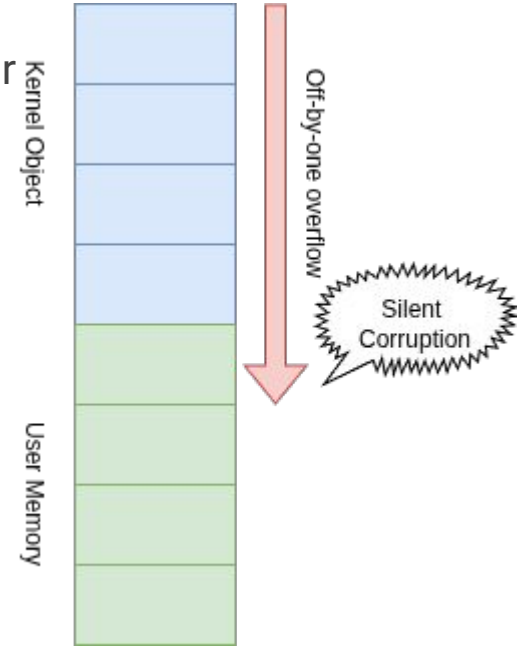
## The Accidental Unification of Memory

- **64-bit kernels dropped Lowmem/Highmem juggling:** one continuous Linear Map.
- Every physical page gets a fixed virtual twin (fast & simple).
- **Kernel can now address all memory:** including user pages.
- Convenience removed the natural isolation barrier.

# The Linear Mapping Threat II

## Adjacent Security Failure

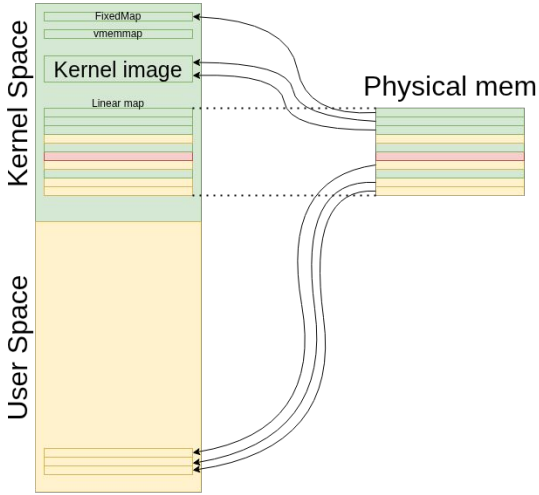
- Kernel allocators (kmalloc, slab, per-CPU data) all use the Linear Map.
- Kernel and user pages live in one physical continuum, no hardware fence.
- A small overflow in kernel space can corrupt neighboring user pages.
- The risk isn’t malice… it’s proximity.



# The Safety Conflict

## Isolation Broken

- **Unified View:** Every physical page has a twin in the kernel's virtual map: no true boundary.
- **The Implicit Trust:** Safety depends entirely on the kernel never making a memory error.
- **Reality Check:** One stray write in kernel space can corrupt user data, no guardrail.
- **Safety Gap:** Functional safety demands provable separation — the linear map removes it.



# Reconsidering Isolation

## Highmem as a Conceptual Barrier

- **Old Idea, New Role:** 32-bit Highmem once split memory by necessity; today, it could define safety boundaries by choice.
- **Selective Reach:** Mark safety-critical processes so their pages stay outside the kernel's linear map, reachable only via explicit mappings.
- **Controlled Access:** Legacy interfaces like `/dev/mem` must respect these no-go zones.
- **Goal:** Limit what the kernel can touch… not to weaken Linux, but to contain its reach.

# Potential Software Mitigation:

## Proof-of-Concept Isolation: `secretmemfd()`

- **Goal:** Show that user-space memory can exist outside the kernel's linear map.
- **Mechanism:** `secretmemfd()` allocates pages unmapped from the kernel view — even privileged code cannot access them.
- **Value:** Demonstrates that physical-level isolation is technically possible in today's Linux.
- **Limitations:**
  - `copy_to_user()` / `copy_from_user()` fail on these regions.
  - Only the owning process can safely access its data.
  - Designed for security, not safety… not transparent to existing software.

*(Note: behavior depends on architecture; ARM64 may limit full page unmapping.)*

# Safety Features: Defense & Detection


- Introduction & Scope
- VMA Architecture: Lifecycle & Features
- The Linear Mapping Threat
- **Safety Features: Defense & Detection**
- Defining the Path to Functional Safety
- Epilogue

**A fortress under siege… brave defenders, but no peace of mind.**

# Defense Baseline I: Kernel Hardening

## List Poisoning and Randomized Freelist Management

- **List Poisoning:** Checks list integrity and poisons freed pointers to catch early use-after-free or double deletions.
  - Great for early bug detection, but offers no containment.
- **SLAB Hardening:** Shuffles SLAB allocations to stop predictable heap layouts.
  - Boosts security; adds non-determinism: not ideal for safety.
- **KASAN:** Uses shadow memory to detect invalid accesses and use-after-free.
  - Powerful diagnostic tool, too heavy for production safety use.

# Defense Baseline II: Memory Control

## OOM Killer and Overcommit Policies (A Safety Compromise?)

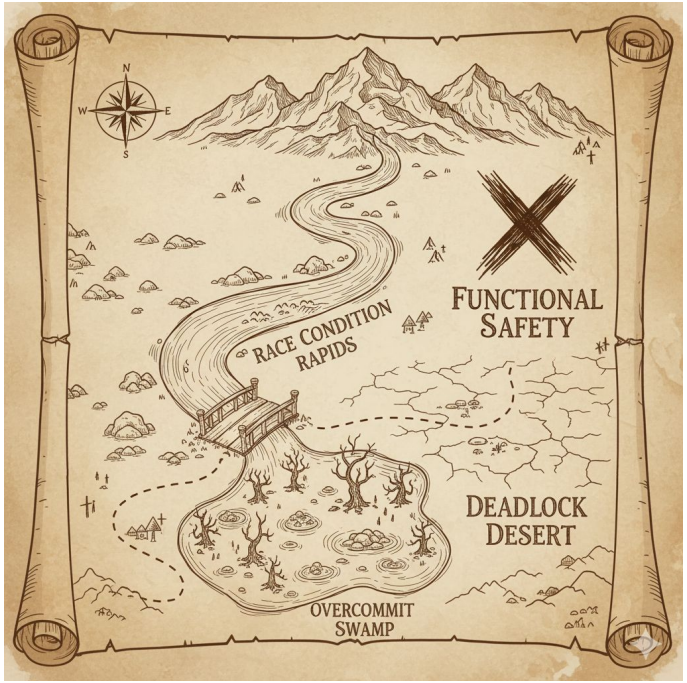
- **OOM Killer:** When memory runs out, it frees space by killing a process based on heuristic scoring.
  - Keeps the system alive, not predictable.
- **Overcommit Policies:** (`vm.overcommit_memory` = 0/1/2) decide how much virtual memory to promise.
  - Default favors efficiency over guaranteed success.
- **Memory Pressure Handling (PSI, DAMON):** Monitor stalls and reclaim memory under stress.
  - Reactive, heuristic: help performance, not determinism.
- **Safety Gap:** These mechanisms ensure survival, not bounded behavior.
  - Good for uptime, weak for safety.

# Detection: VMA Cache Integrity

## Identifying and Preventing Fast-Path Corruption

- **VMA Cache Integrity Risks:** Concurrency and timing bugs can corrupt cached VMA entries.
- **Real-world Failures:**
  - **CVE-2018-17182:** Sequence-number overflow led to stale VMA cache entries and crashes.
  - **CVE-2016-5195 (Dirty COW):** Race in copy-on-write logic broke page-protection rules.
- **Safety Lesson:**
  - These aren't attacks: they're accidents of timing.
  - They prove that fast-path optimizations can silently break memory isolation.

# Defining the Path to Functional Safety




- Introduction & Scope
- VMA Architecture: Lifecycle & Features
- The Linear Mapping Threat
- Safety Features: Defense & Detection
- **Defining the Path to Functional Safety**
- Epilogue

**‘X’ Never Marks the Spot… Except When It Does.**

# The Architectural Gap

## Why Existing Defenses are Insufficient Against Linear Mapping Threats

- **Existing Defenses:** Built to stop exploitation after a fault: they react to bad behavior.
- **The Linear Map Threat:** A design flaw, not an exploit: it makes faults inherently possible.
- **The Gap:** Without spatial isolation or deterministic allocation, Functional Safety can’t be guaranteed: even when no attacker is present.

# Functional Safety Requirements for VMA

## What “safe” memory management must guarantee

- **Spatial and Temporal Isolation:** Each VMA must stay within its bounds for its entire lifetime: no neighbor overlap, no kernel overreach, no cross-process bleed.
- **Deterministic Resource Access:** Memory availability for safety-critical tasks must be predictable. Enforced through Never Overcommit or Allocation-on-Usage (AoU) policies.
- **Concurrency Integrity:** VMAs must remain consistent under parallel activity: no race-driven corruption like Dirty COW or stale cache reuse.
- **Transparent Verification:** The system must be able to prove these properties at runtime or via traceability: safety cannot rely on trust alone.

# Mitigation Avenues Under Review

## Architectural Proposals

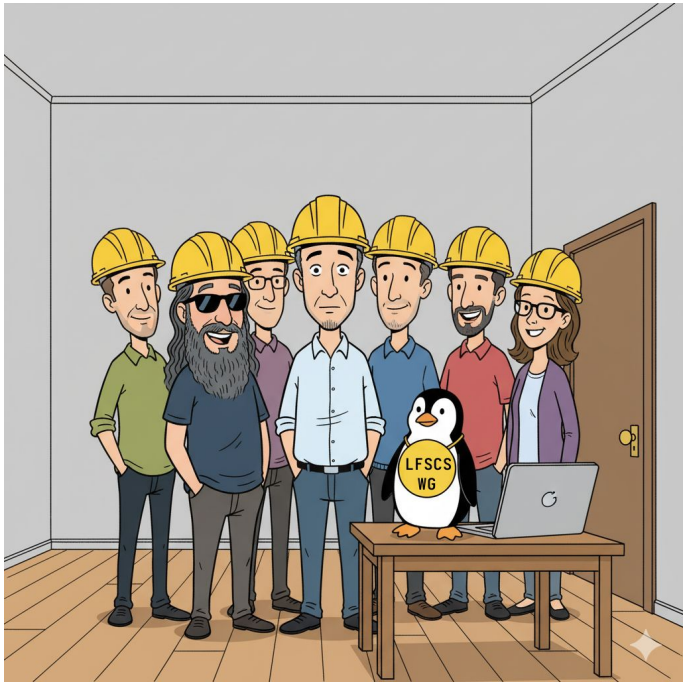
- **Layered Isolation (Revisiting Highmem):** Introduce deliberately unmapped regions to re-establish hard separation between trusted and untrusted memory.
- **Controlled Access Pools:** Extend the `secretmemfd()` principle system-wide: defining kernel-invisible “safe” VMAs by design, not by opt-in.
- **Allocation on Usage (AoU):** Commit memory only when first touched, adding temporal determinism while reducing overcommit risk.

# Next Step

## Quantifying Risk & Feasibility: Setting the Stage for Design

- **Understand Allocation Behavior:** Examine SLUB's per-CPU and global caches to see how real allocation paths align, or conflict, with isolation goals.
- **Validate Docs vs. Code:** Check whether what's written still matches what's running. Identify outdated assumptions and real-world divergences.
- **Build the Roadmap:** Combine all findings: VMA lifecycle, overcommit, linear map, allocator behavior, into a structured plan for measurable risk reduction.

# Epilogue




- Introduction & Scope
- VMA Architecture: Lifecycle & Features
- The Linear Mapping Threat
- Safety Features: Defense & Detection
- Defining the Path to Functional Safety
- **Epilogue**

**No one finds safety alone — it's always a team expedition.**

Sponsor Message

# From Insight to Action

## Key Takeaways & Roadmap

### Technical Lessons

- The Linear Map trades safety for speed.
- Security tools help, but don’t ensure determinism.
- Safety needs true isolation and predictable allocation.

### Next Steps for the Working Group

- Define the Safe VMA concept.
- Prototype isolation mechanisms.
- Align findings with ELISA Architecture SIG.

ELISA
Enabling Linux in
Safety Applications

# Thanks