Virtualization-assisted Security

A Resilient Security Foundation for the Linux Kernel

FOSDEM 2025

Sergej Proskurin | BlueRock Security

February 1-2, 2025

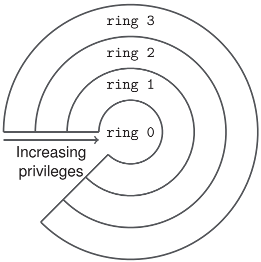
Motivation & Background

Status Quo

Problem: The Kernel Self-Protection Paradox

The Linux kernel is responsible for:

- Protecting and isolating applications in user space
- Protecting **itself** from unauthorized accesses (e.g., kernel modules, exploits, `BPF` programs, etc.)



Who protects the Linux kernel from malicious entities with same privileges?

Virtualization-assisted Security (VAS)

Rethinking Linux Kernel Security

Idea: Design the Linux kernel with Virtualization-assisted Security in mind

- Alleviate the strict separation between the Linux kernel and a VMM
  - Empower Linux with new capabilities supported by the system's virtualization extensions
  - Virtualization extensions become inherent OS building blocks for defense purposes
- Equip Linux subsystems with security primitives offered by the VMM
  - The VMM becomes a resilient security support layer offering holistic security services
  - Define security policies to protect critical kernel code and data
- Strengthen the Linux kernel's defense against malicious activities
  - Enhances overall security without replacing OS responsibilities
  - Detects and prevents unauthorized activities, despite the presence of kernel vulnerabilities

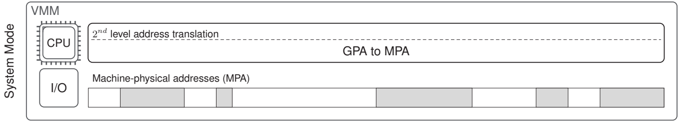
Virtualization-assisted Security (VAS)

BlueRock Security Architecture

The Security Support Layer

- Based on the NOVA μhypervisor
- Provides a hypercall interface to supply VMs with VAS capabilities
- Supports 64-bit Intel & Armv8-A

→ The conceptual architecture is hypervisor-agnostic (Similarly applicable to Linux KVM)



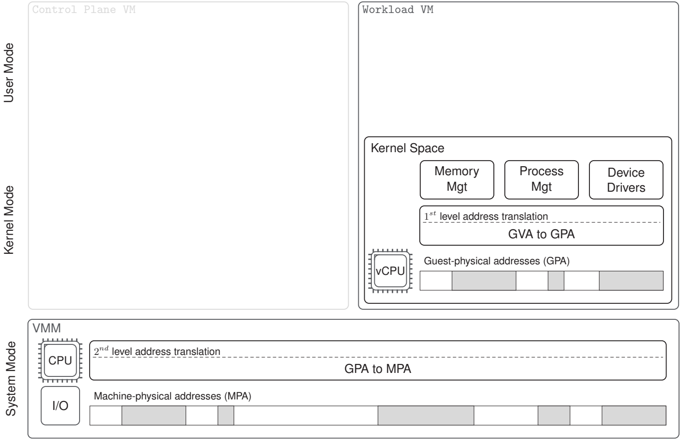
Virtualization-assisted Security (VAS)

BlueRock Security Architecture

The Workload VM

- Enlightened, VAS-aware general-purpose Linux kernel
- Actively collaborates with the security support layer
- Leverages VAS building blocks to enhance the security of subsystems

→ Can be a standalone VM

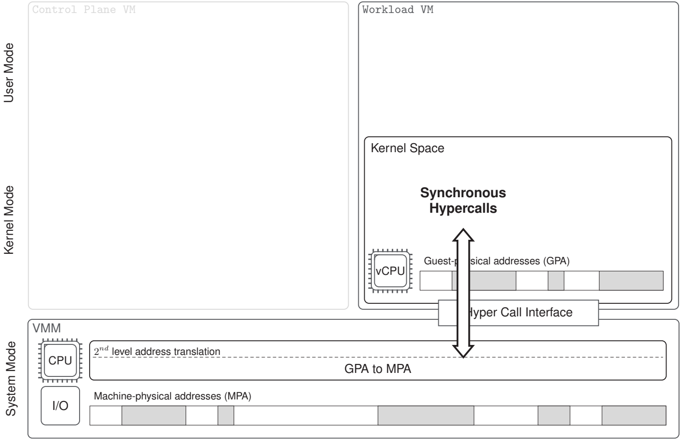


Virtualization-assisted Security (VAS)

BlueRock Security Architecture

Communication via hypercalls

- Policy initialization and initial state/context sharing
- Policy compliance verification



Virtualization-assisted Security (VAS)

BlueRock Security Architecture

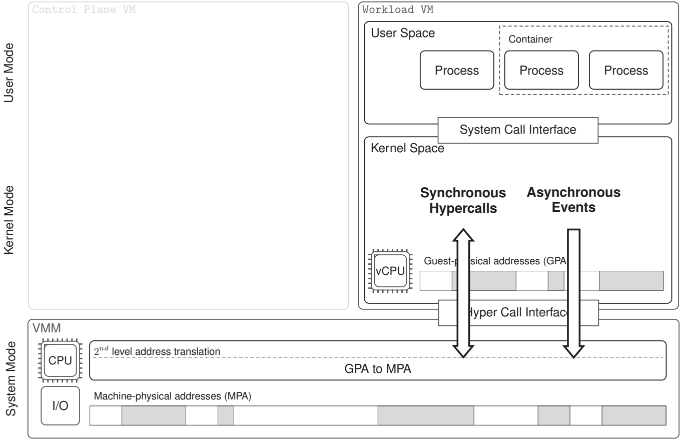
Communication via hypercalls

- Policy initialization and initial state/context sharing
- Policy compliance verification

Communication via `virtio`

- Optional event reporting (process lifecycle, container drift, etc.)
- Focus on user space processes and container events

→ Policy violations trigger fault injections into the Workload VM



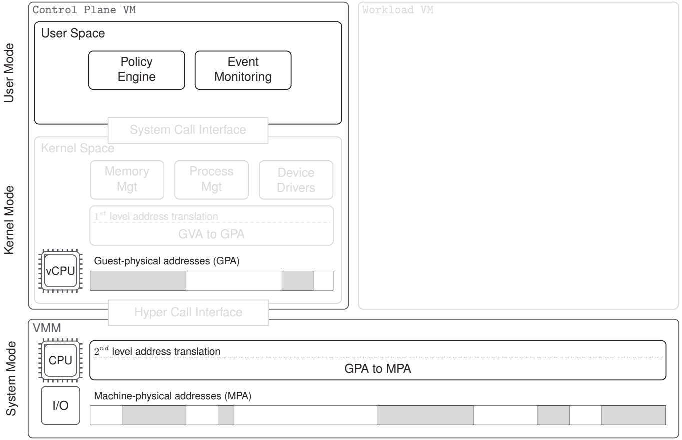
Virtualization-assisted Security (VAS)

BlueRock Security Architecture

The Control Plane VM (optional)

- Highly stripped-down, VAS-aware Linux kernel
- Decouples system monitoring and policy decision points
- Configures OPA-based policies
- Receives security events via `virtio` from the Workload VM and VMM

→ A compromised workload VM cannot easily evade monitoring



Virtualization-assisted Security Primitives

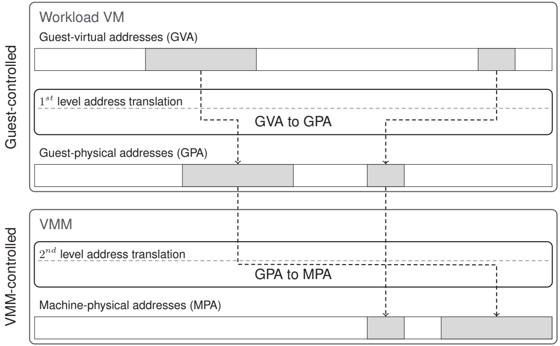
Overview

The security support layer implements VAS primitives

- Linux kernel integrity targetting `.text` and `.rodata`
  - Prevents unauthorized modification of the Linux kernel, modules, and `BPF` programs
  - Safeguards the `VDSO`, `idt_table`, `sys_call_table`, etc.
- Selective data structure and pointer integrity
  - Global data structures, including `core_pattern`, `modprobe_path`, etc.
  - Process credentials, privileged inodes, system-trusted keyrings, fops, etc.
- Further security features for the Linux kernel:
  - Control register value locking, SELinux policy protection, driver signature enforcement,
  - Read-only file protection, kernel patching mediation, etc.

Virtualization-assisted Security Primitives

Linux Kernel Integrity

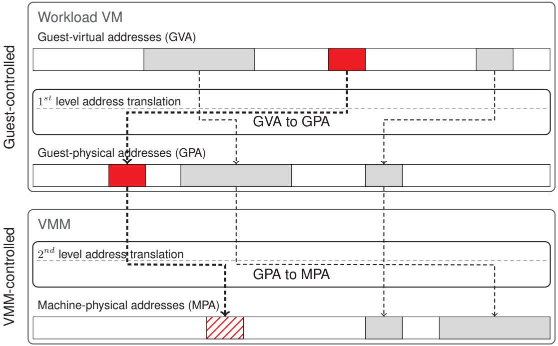


The Linux kernel controls how and which memory regions are to be protected

- The Workload VM uses hypercalls to register memory regions in the VMM
  - This applies to static kernel segments, as well as dynamically loaded code
- Combine the Linux kernel's `mm` with Second-Level Address Translation (SLAT)
  - Grant access permissions exclusively to registered memory regions

Virtualization-assisted Security Primitives

Linux Kernel Integrity

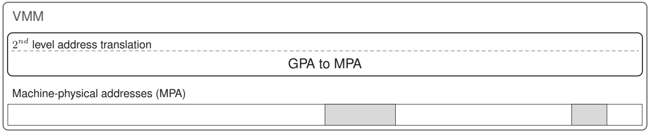


Linux kernel integrity allows to identify unauthorized supervisor executions

- Detect any supervisor execution of non-registered kernel memory
  - This is efficiently possible with hardware support (Intel `MBEC` / Arm `PXN`)
- Eliminate unauthorized code injections into the kernel

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Memory Model

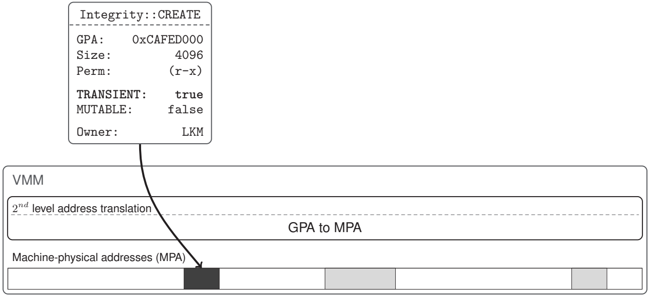


Each registered memory region has an associated type and flags

- Memory types: `CODE`, `CODE_PATCHABLE`, `DATA`, `DATA_READ_ONLY`, etc.
  - Hypervisor- and hardware-independent memory types
  - Translate into hardware-defined memory permissions
- Memory flags: `TRANSIENT` and `MUTABLE`

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Memory Model

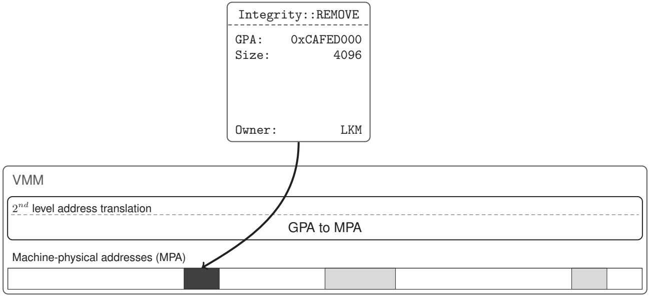


The `TRANSIENT` flag distinguishes between static and dynamic memory regions

- Transient memory regions are dynamic and can be removed
  - E.g., `.init.text` sections, kernel modules, and `BPF` programs
- Non-transient memory regions are static and cannot be unmapped
  - Static memory regions that do not change in benign contexts
  - E.g., `.text` and `.rodata`

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Memory Model

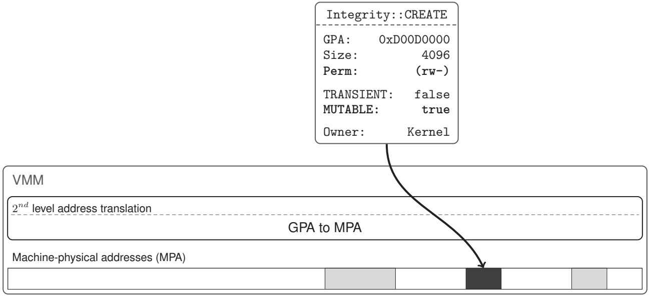


The `TRANSIENT` flag distinguishes between static and dynamic memory regions

- Transient memory regions are dynamic and can be removed
  - E.g., `.init.text` sections, kernel modules, and `BPF` programs
- Non-transient memory regions are static and cannot be unmapped
  - Static memory regions that do not change in benign contexts
  - E.g., `.text` and `.rodata`

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Memory Model

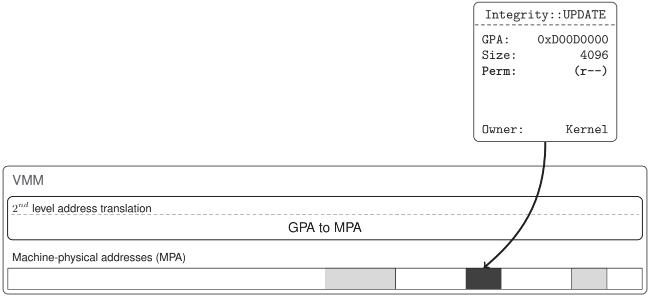


The `MUTABLE` flag allows memory regions to update their memory type

- Mutable memory regions allow only more-restrictive updates of their memory types
  - E.g. the `.data..ro_after_init` section changes its memory type: `DATA` to `DATA_READ_ONLY`
- Immutable memory regions lock-down their contents
  - Once the security permissions are applied, they cannot be undone
  - Highly-constrained environments can lock-down the entire memory map

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Memory Model



The `MUTABLE` flag allows memory regions to update their memory type

- Mutable memory regions allow only more-restrictive updates of their memory types
  - E.g. the `.data..ro_after_init` section changes its memory type: `DATA` to `DATA_READ_ONLY`
- Immutable memory regions lock-down their contents
  - Once the security permissions are applied, they cannot be undone
  - Highly-constrained environments can lock-down the entire memory map

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Challenges in Dynamic Environments

The Linux kernel is highly dynamic and heavily relies on run-time patching

- Alternative instructions, jump labels, static keys, static calls, tracepoints
  - Optimize performance by replacing instructions, avoiding indirect jumps, etc.
  - Enable kernel features by toggling rarely-used conditional code paths
  - Attach probes/functions to statically (or dynamically) defined hooks

Attackers can abuse the patching facility to take over the kernel

- Attackers can reuse patching-related code gadgets
- Attackers can compromise patching-related data structures to
  - Arbitrarily write to the kernel code segment, despite CFI
  - Disarm security monitors (in part without having to change the code segment)

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Challenges in Dynamic Environments

The Linux kernel is highly dynamic and heavily relies on run-time patching

- Alternative instructions, jump labels, static keys, static calls, tracepoints
  - Optimize performance by replacing instructions, avoiding indirect jumps, etc.
  - Enable kernel features by toggling rarely-used conditional code paths
  - Attach probes/functions to statically (or dynamically) defined hooks

Attackers can abuse the patching facility to take over the kernel

- Attackers can reuse patching-related code gadgets
- Attackers can compromise patching-related data structures to
  - Arbitrarily write to the kernel code segment, despite CFI
  - Disarm security monitors (in part without having to change the code segment)

→ **Challenge:** How to reliably distinguish **legitimate** from **malicious** changes?

Virtualization-assisted Security Primitives

Linux Kernel Integrity: Challenges in Dynamic Environments

To thwart attacks abusing the patching facility we must:

(i) Ensure that the patching facility is always called from a benign context

(ii) Maintain integrity of patching-related data structures

Issue: While we can address **(i)** with `CFI`, **(ii)** remains an open problem!

Virtualization-assisted Security Primitives

To thwart attacks abusing the patching facility we must:

(i) Ensure that the patching facility is always called from a benign context

(ii) Maintain integrity of patching-related data structures

Issue: While we can address **(i)** with `CFI`, **(ii)** remains an open problem!

Idea: Leverage Virtualization-assisted Security to achieve **(i)** and **(ii)**!

The Vault

Subsystem Isolation for the Linux Kernel

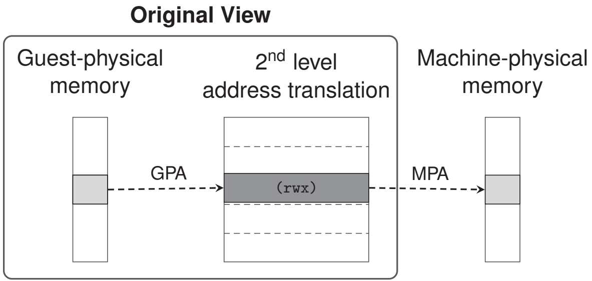
The Vault is a general-purpose security primitive to isolate subsystems

- Utilize hardware virtualization to define Vaults in kernel space
  - Encapsulate and isolate sensitive code and data in dedicated sections in the Vault
  - Empower Linux to shift entire subsystems into Vaults
  - Partition and isolate Vault-protected subsystems from each other / the kernel
- The Linux kernel must not directly access arbitrary memory inside the Vault
  - Unauthorized accesses trap into the security support layer
  - Govern Vault transitions through designated transit points
  - Maintain sensitive subsystem-related data exclusively inside the Vault

→ Attackers cannot divert control-flow to reuse code or alter sensitive data in the Vault

The Vault

The NOVA GST Spaces Subsystem



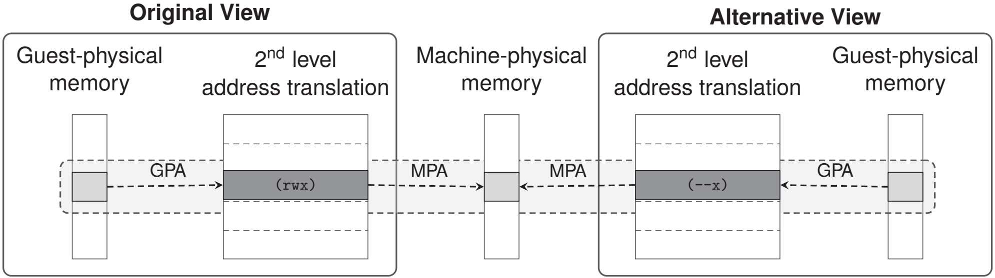
Typically, a VMM uses one set of second level address translation tables (SLAT)

- Defines the guest's global view on the physical memory

→ Changes in the global view are perceived by all vCPUs

The Vault

The NOVA GST Spaces Subsystem



Introducing the NOVA GST Spaces subsystem

- Maintains different views on the guest's physical memory
- Allocates and assigns different memory views to vCPUs

→ Switch views instead of relaxing permissions in a global view!

The `Vault`

Showing `NOVA GST Spaces` In Action

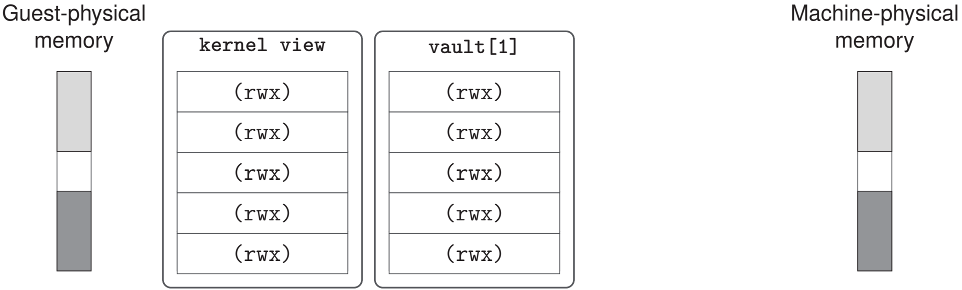
Leverage SLAT tables to configure multiple disjoint guest-physical memory views

- Only a single guest-physical memory view can be active at a given time

→ Propagate restrictive permissions of each `Vault` across all available memory views

The Vault

Showing NOVA GST Spaces In Action

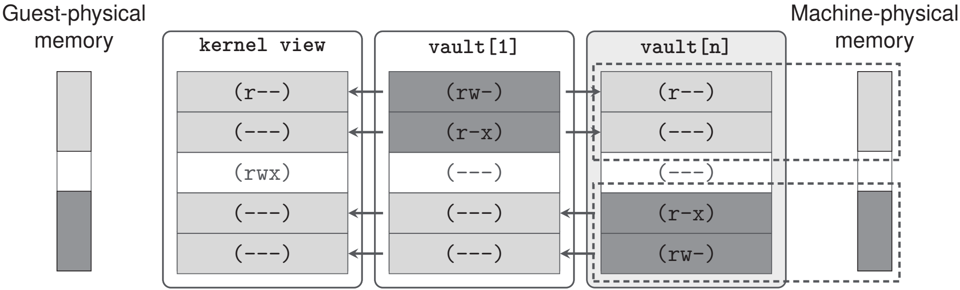


One Vault requires 2 memory views (restricted and relaxed view)

- The restricted kernel view unifies memory restrictions of all Vaults
  - Configured as the default view on all vCPUs

The Vault

Showing NOVA GST Spaces In Action

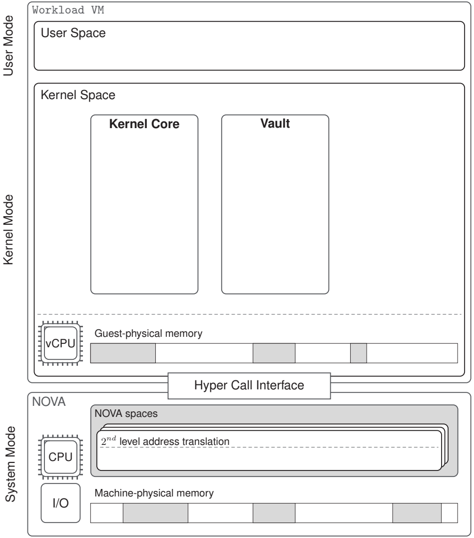


For n Vaults, we define $n+1$ views on the guest-physical memory

- Each $\{\texttt{vault}[i] \mid i \in \{1, \ldots, n\}\}$
  - Relaxes the permissions of sensitive memory in Vault i
  - Restricts access to memory regions belonging to the kernel and Vaults $\neq i$

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

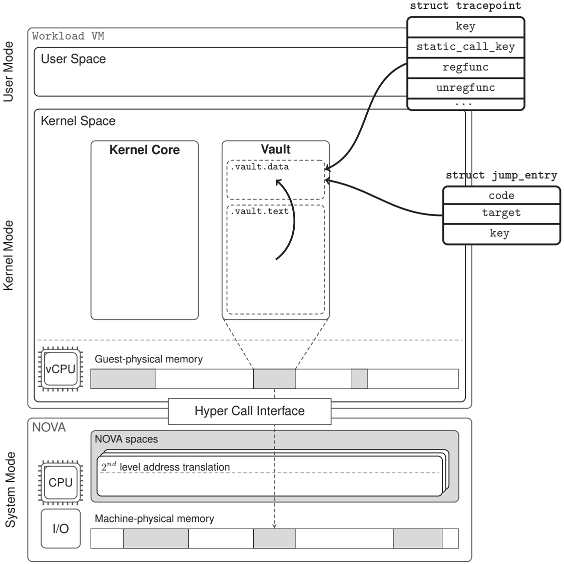


`Vault`'s API allows to partition the Linux kernel

- Move (*patching*|*tracing*)-related code and data into designated sections within the vault
- Define authorized `Vault` entry and exit points
- Communicate locations of the `Vault`'s sections and transition points to the VMM at boot time

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

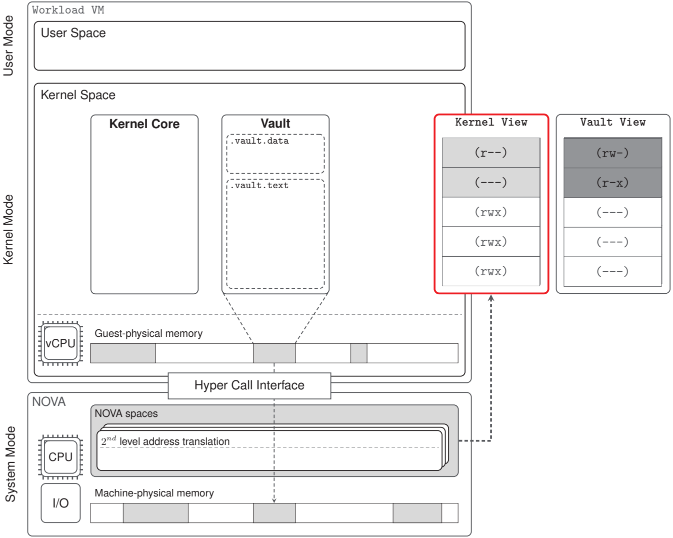


Key requirements for secure patching

1. Code outside the vault must not be able to reuse patching-related code gadgets
2. Only code within the `Vault` can access sensitive data structures
   - `struct alt_instr`, `struct jump_entry`,
   - `struct tracepoint`, etc.
3. Only code within the `Vault` is authorized to instruct the VMM to patch kernel code

Integrating Vaults into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access



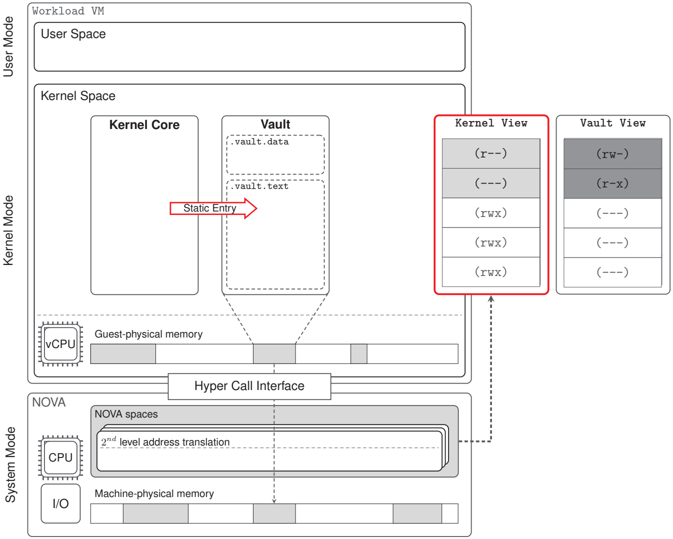
Enforce isolation via `NOVA GST Spaces`

- The kernel view restricts access to the `Vault`-protected code and data
- The `Vault` view defines permissions of the isolated sections inside the vault
- `NOVA Spaces` govern `Vault` transitions
  - Switching the memory view allows to enter/exit the `Vault`

→ Technology not bound to `NOVA`

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

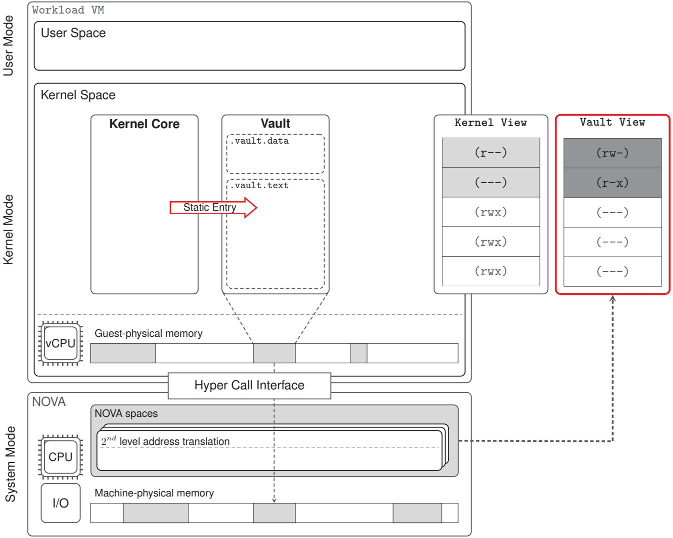


Vault entries at designated locations

- Authorized **`entry points`**
  - Define the `Vault`'s interface
  - Annotated function entries (future: leverage `objtool`)
- The `Vault` can be entered only by executing trusted **`entry points`**

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

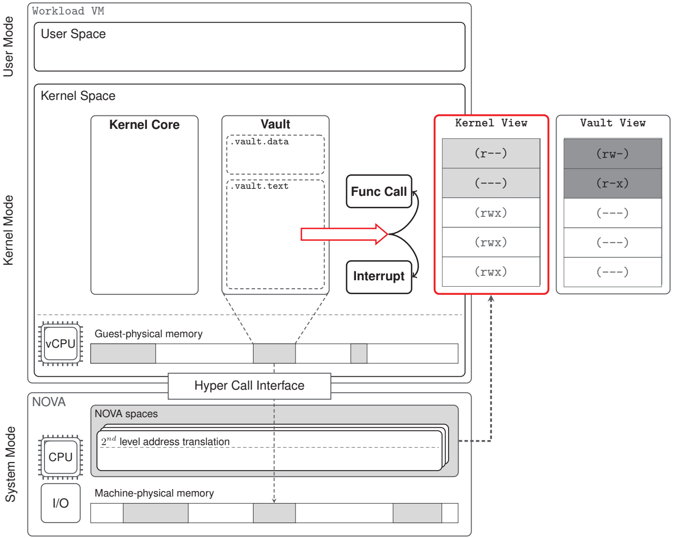


Vault entries at designated locations

- Authorized `entry points`
  - Define the `Vault`'s interface
  - Annotated function entries (future: leverage `objtool`)
- The `Vault` can be entered only by executing trusted `entry points`

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

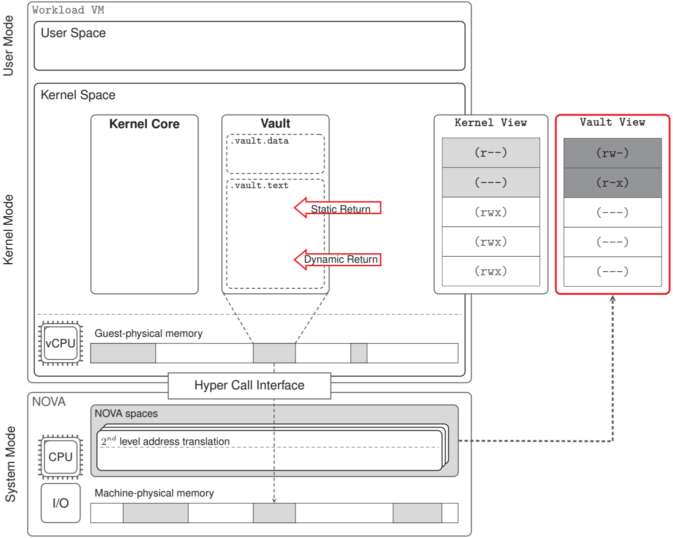


Temporary `Vault` exits and returns

- `Vault` exits due to external functions
  - Static `return points`: identified via `objtool`
  - Passed to the VMM during early boot
- `Vault` exits due to interrupts
  - Dynamic `return points`: extracted from the stack

Integrating `Vaults` into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

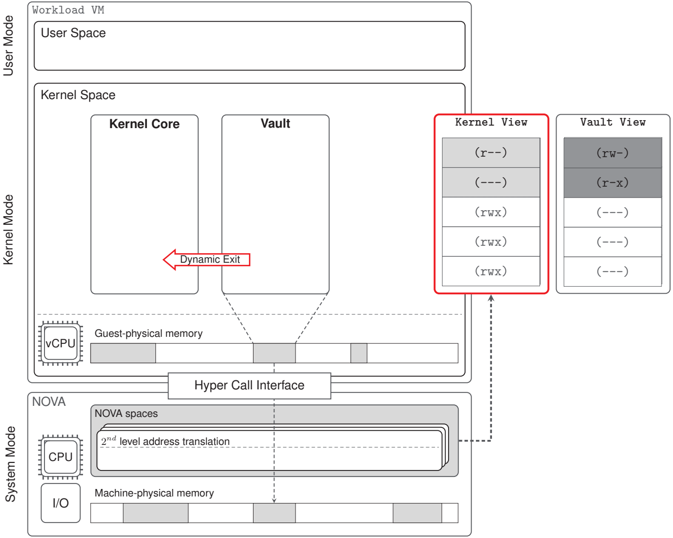


Temporary `Vault` exits and returns

- `Vault` exits due to external functions
  - Static `return points`: identified via `objtool`
  - Passed to the VMM during early boot
- `Vault` exits due to interrupts
  - Dynamic `return points`: extracted from the stack
- Authorized return conditions
  - The `Vault` was legitimately opened
  - The return address matches an authorized `return point`

Integrating Vaults into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access

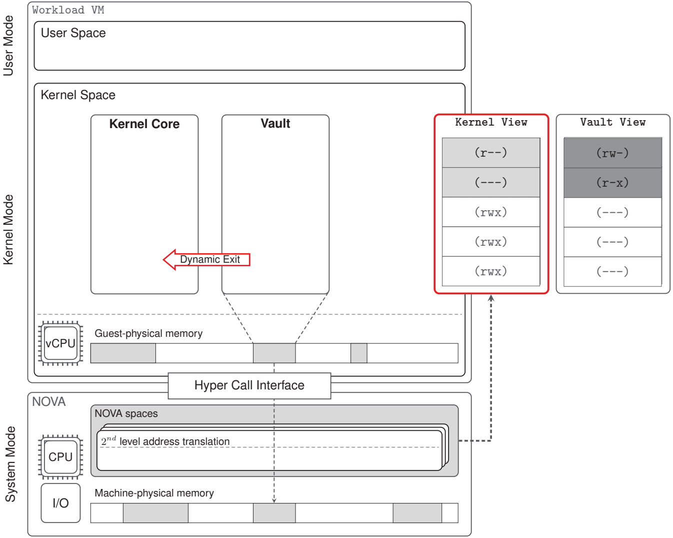


The end of the Vault's lifecycle

- The Vault closes when it reaches its exit point

Integrating Vaults into the Linux Kernel

Harden the Patching and Tracing Facility Against Unauthorized Access



The end of the Vault's lifecycle

- The Vault closes when it reaches its exit point

Wait, what about KPROBES?

The KPROBES & BPF Conundrum

Seeking Community Insights

BlueRock

KPROBES & BPF progs serve as foundation for tracing and security frameworks

- Allows placing hooks at (almost) any point in the kernel
- Enables comprehensive introspection of kernel behavior

→ Ideal for debugging, profiling, and generating security events

Problem: Dangerous in the wrong hands

- KPROBES are not bound by namespaces
  - Potential for leaking data among different execution contexts
- KPROBES-attached BPF programs are frequently deployed without being signed
  - Approaches for signing BPF bytecode have been presented
  - No definitive solution exists, just yet
    (remaining challenges include relocations (CO-RE), or compiled BPF bytecode)

The KPROBES & BPF Conundrum

Seeking Community Insights

BlueRock

Could VAS assist in limiting the attack surface of `KPROBES` & `BPF` progs?

- Move the `KPROBE` facility into a `Vault`?
  → Remove patching gadgets from the Linux kernel!
- Move the `BPF` JIT compiler into a `Vault`?
  → Thwart write attempts from other CPUs to pages holding the `BPF` program at compile-time
- Isolate `BPF` programs from sensitive contexts?
  → Disallow using `BPF` for exploit payloads

Feel free to reach out, we are open for feedback and collaboration!

Final Words | Call for Action

BlueRock

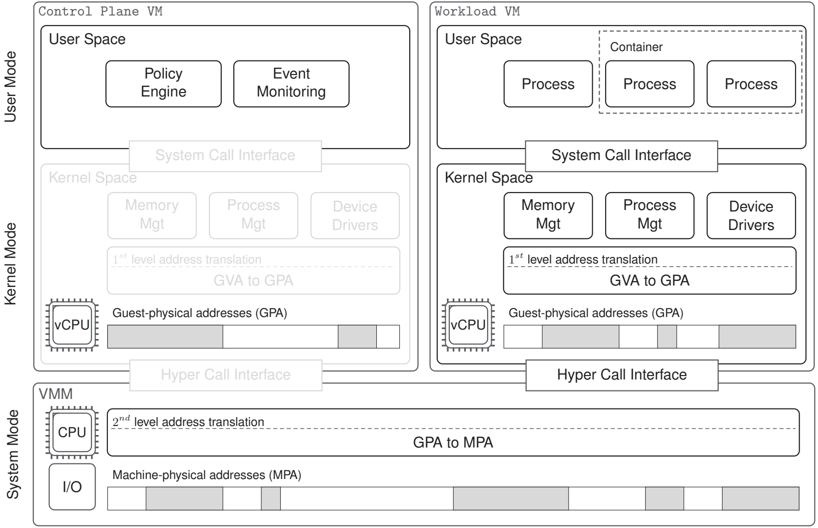
- Engage with the Linux community
  - BlueRock's VAS Linux kernel [1] and NOVA [2] are open source
  - Prepare patches to start getting parts of our code base into the Linux mainline
- Virtualization-assisted Security receives increasing attention from the industry
  - Microsoft (L)VBS, Samsung Knox RKP, Huawei Security Hypervisor, etc.
  - We are in need for a common, hypervisor-agnostic hypercall API for the Linux kernel
- Virtualization-assisted security has not been explored to its full extent
  - Protections around `KPROBES` & `BPF` programs
  - We are open for feedback!

[1] VAS Linux kernel: `https://github.com/bedrocksystems/linux-bhv-patches`
[2] NOVA μhypervisor: `https://github.com/udosteinberg/NOVA`

Thank You

sergej@bluerock.io

A1: VAS Architecture



A2: Data Structure and Context-bound Pointer Integrity

Building blocks to protect security-critical data structures

- Leverage the invariants of the data structure's (d) life-cycle to protect their integrity
- Bind d to its unique and immutable context (maintain the result in the security support layer)

$$h = SipHash(addr_d \,||\, addr_{context} \,||\, d)$$

- Verify the data structure's integrity at selected verification points
  (Utilize a custom VAS LSM to consult the security support layer)

BlueRock supports the following VAS capabilities:

- Process credential protection
- Privileged inode protection
- System-trusted keyring protection