Enabling Windows Credential Guard in KVM

NICOLAS SAENZ JULIENNE, ANEL ORAZGALIYEVA

Intro

We are Kernel/Hypervisor engineers at AWS.

Building the Nitro Hypervisor which powers most of EC2.

It comprises a mix of custom user-space components and Linux/KVM.

All the work presented here is the result of multiple people’s efforts. The project was bootstrapped and led by Evgeny Iakovlev. Later joined by Siddharth Chandrasekaran, Sebastian Ott, Alex Graf, Anel Orazgaliyeva and Nicolas Saenz Julienne.

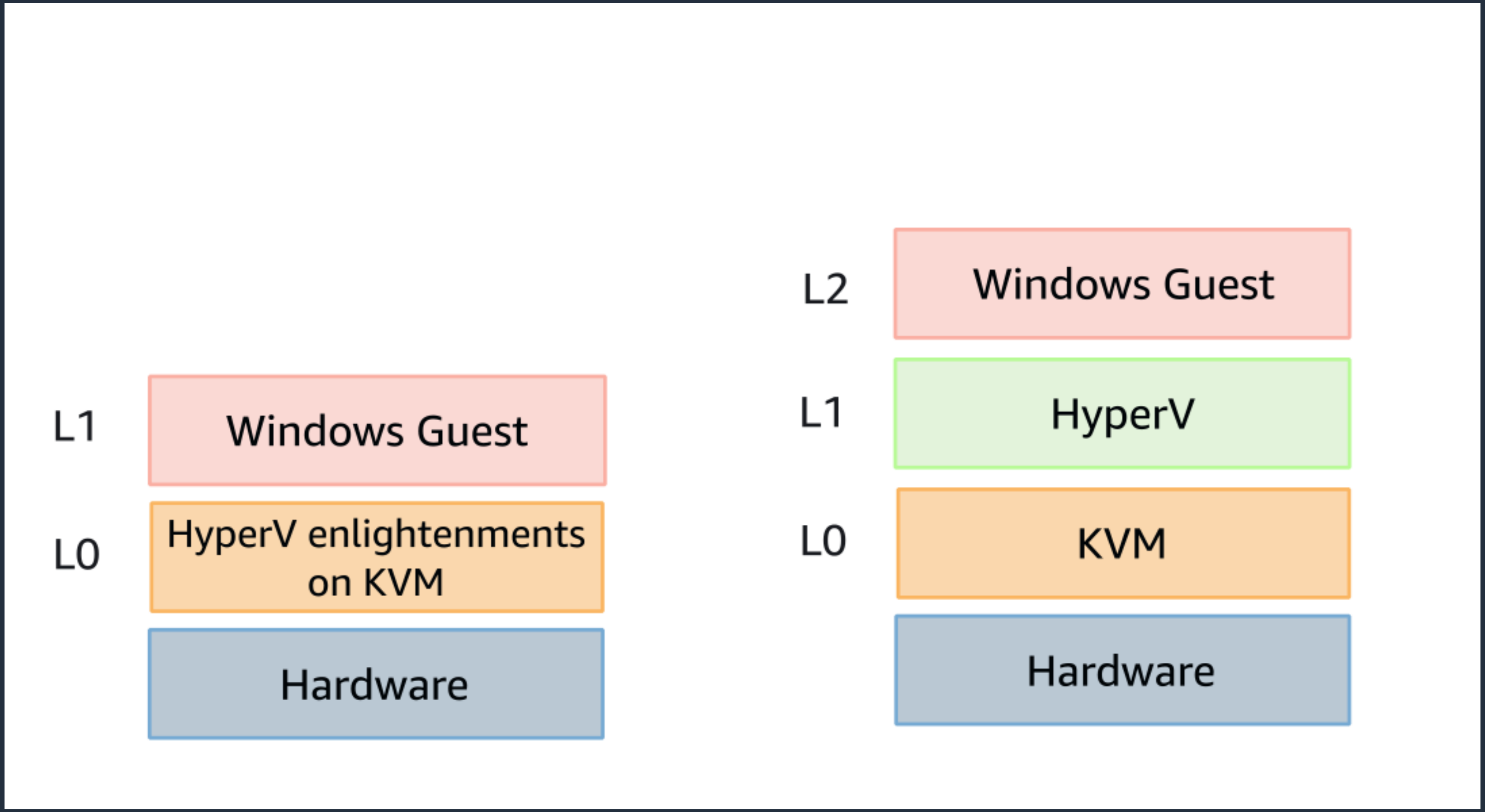
Agenda

Microsoft's Virtualization Based Security (VBS) technologies and their *raison d'être.*

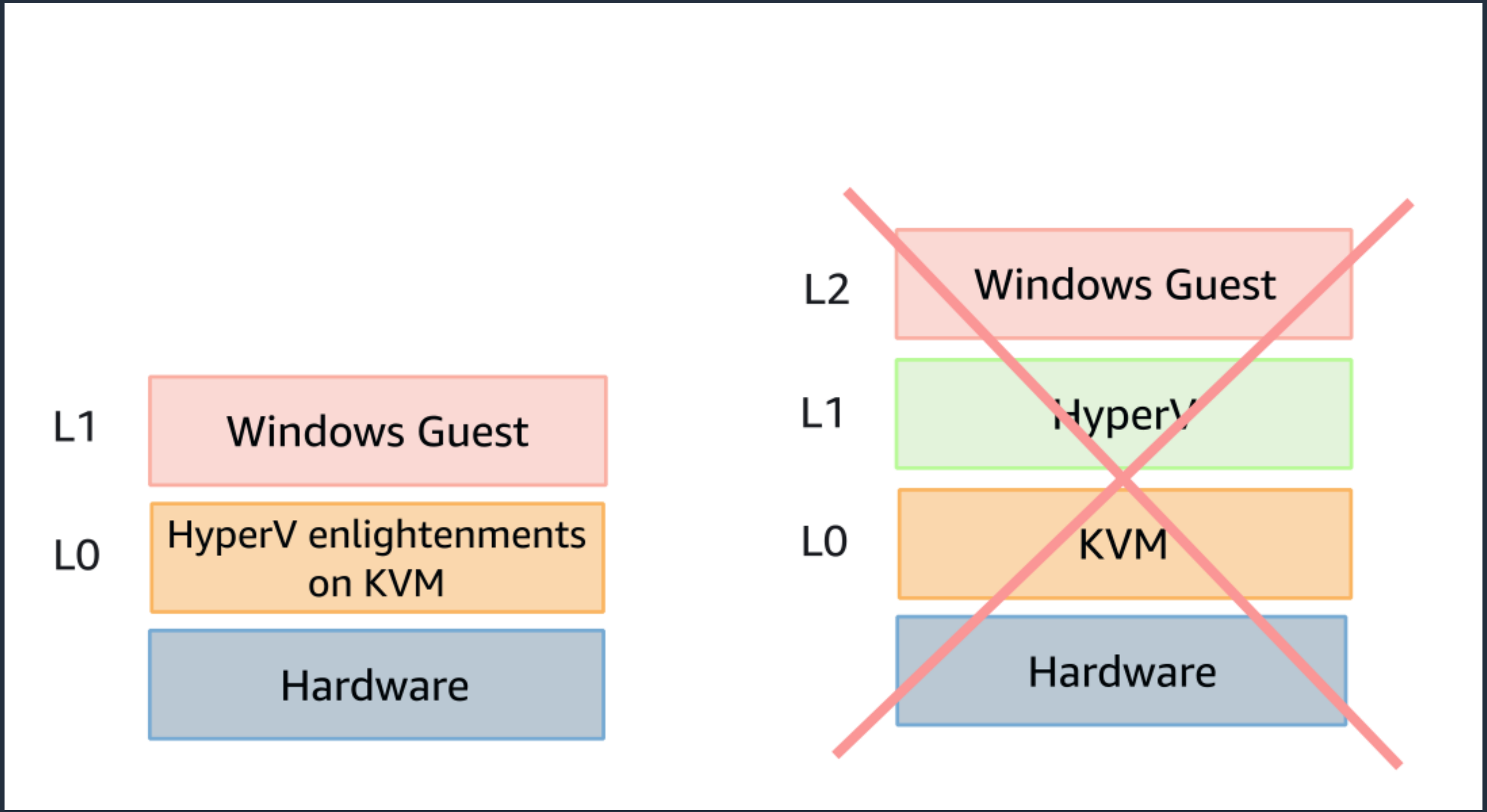
VSM overview including some implementation notes.

Upstreaming plan and challenges.

Disambiguation: Virtualizing Windows



Disambiguation: Virtualizing Windows



Disambiguation

Virtualization Based Security (VBS): *Technologies that rely on the hypervisor to create isolated virtual environments.*

Device Guard (DG): *Set of components that prevent untrusted code from running on Windows.*

Credential Guard (CG): *VBS technology that prevents attackers from stealing credentials by virtue of storing them in memory inaccessible to the kernel.*

Virtual Secure Mode (VSM): *Set of virtualization extensions that serve as the building block for CG and DG. Defined in Microsoft's TLFS (Hypervisor Top Level Functional Specification).*

Enabling Hyper-V’s Virtual Secure Mode in KVM

NICOLAS SAENZ JULIENNE, ANEL ORAZGALIYEVA

Platform requirements

Mandatory:

- Secure boot
- Modern CPU with virtualization extensions: Intel 8th generation, AMD Zen 2, Qualcomm Snapdragon 8cx and later.

Recommended:

- UEFI lock
- TPM

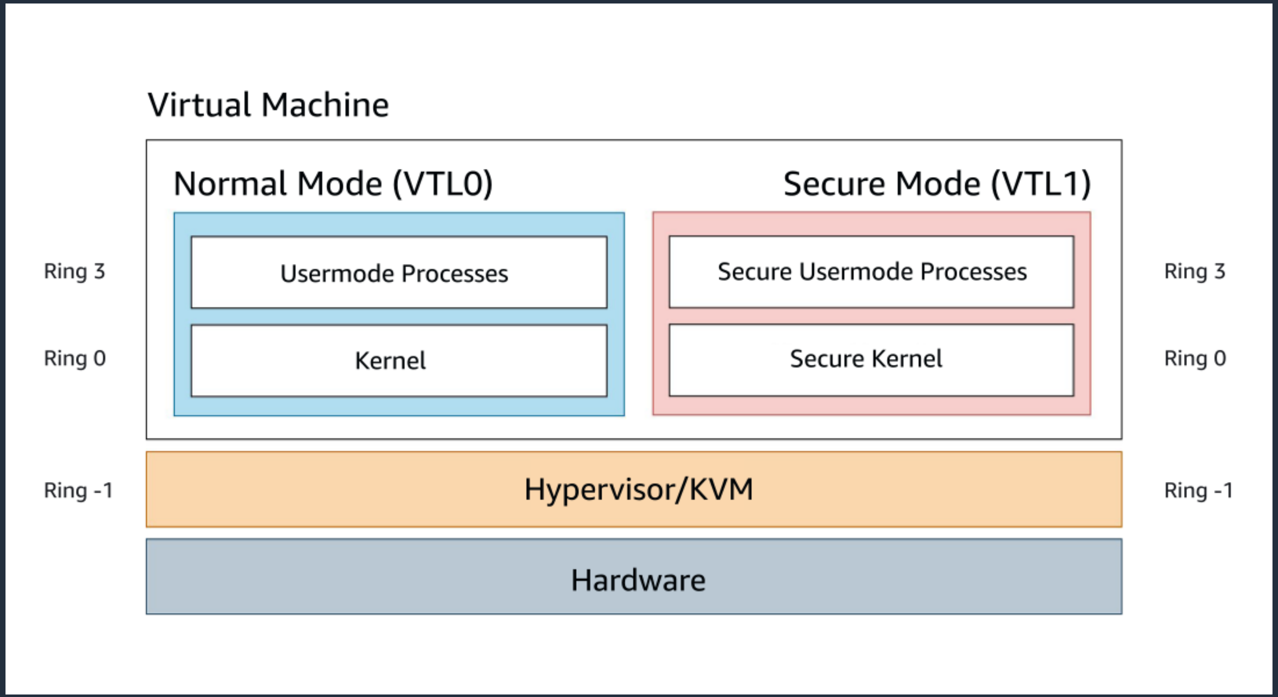
VSM Overview

Virtual Secure Mode

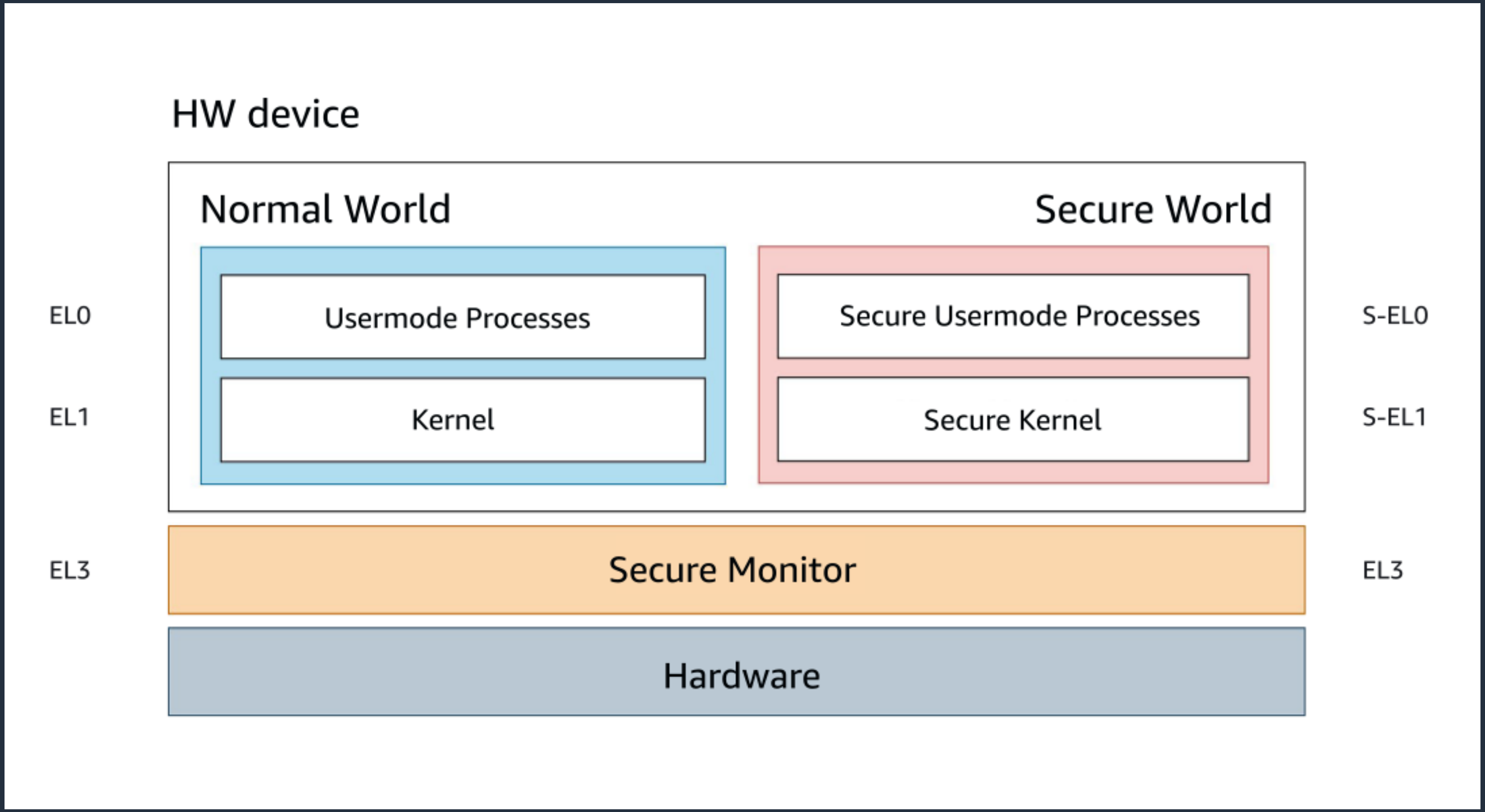
Each vCPU now has a Virtual Trust Level (VTL) attribute: the higher the VTL, the more privileged.

Each VTL has its own CPU state, a private interrupt controller, and a set memory protections.

Microsoft implemented this on both x86_64 and arm64 (Documentation only public for x86).



ARM TrustZone



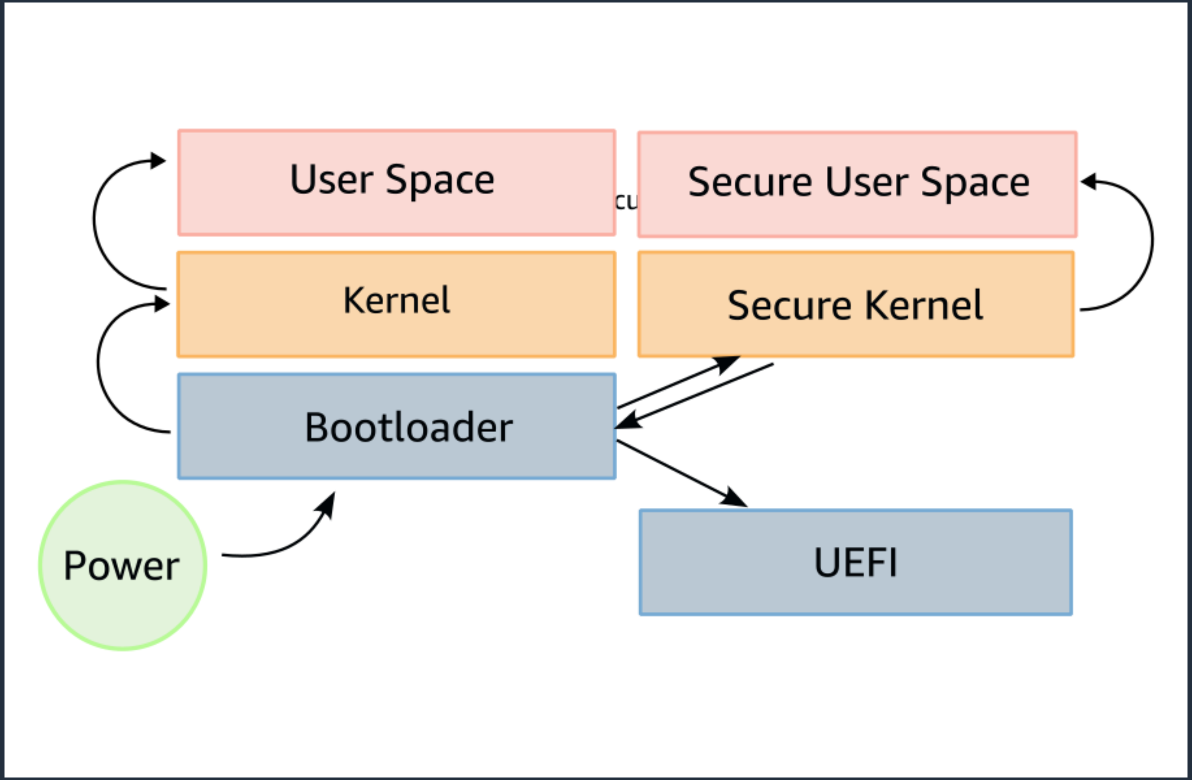
Booting with VSM

The bootloader queries the VSM/CG/DG configuration from UEFI.

If enabled it loads the Secure Kernel (SK), which enables VSM, moves into VTL1, bootstraps the rest of vCPUs and launches secure processes.

It then jumps back into the bootloader in VTL0, which continues the regular boot process.

Windows' kernel communicates with SK through mailboxes and shared memory.`

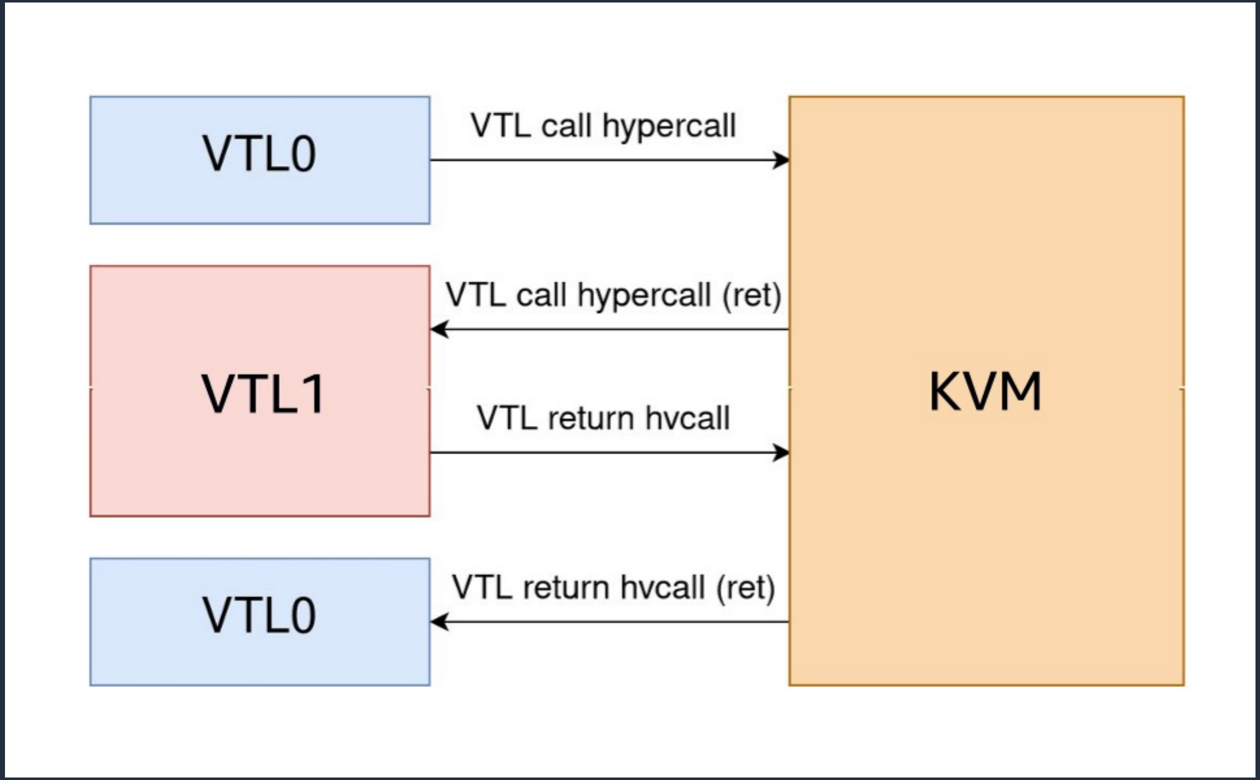


Synchronous VTL Switching

VSM introduces two hypercalls for VTL switching: VTLCALL and VTLRETURN.

The hypervisor saves the calling CPU state, then loads the target VTL's context (IP, SP, CRs, etc…) as well as the VTL's APIC device model state. The hypervisor goes back to guest mode.

It's only possible to move up/down one VTL per hypercall.



Asynchronous VTL Switching

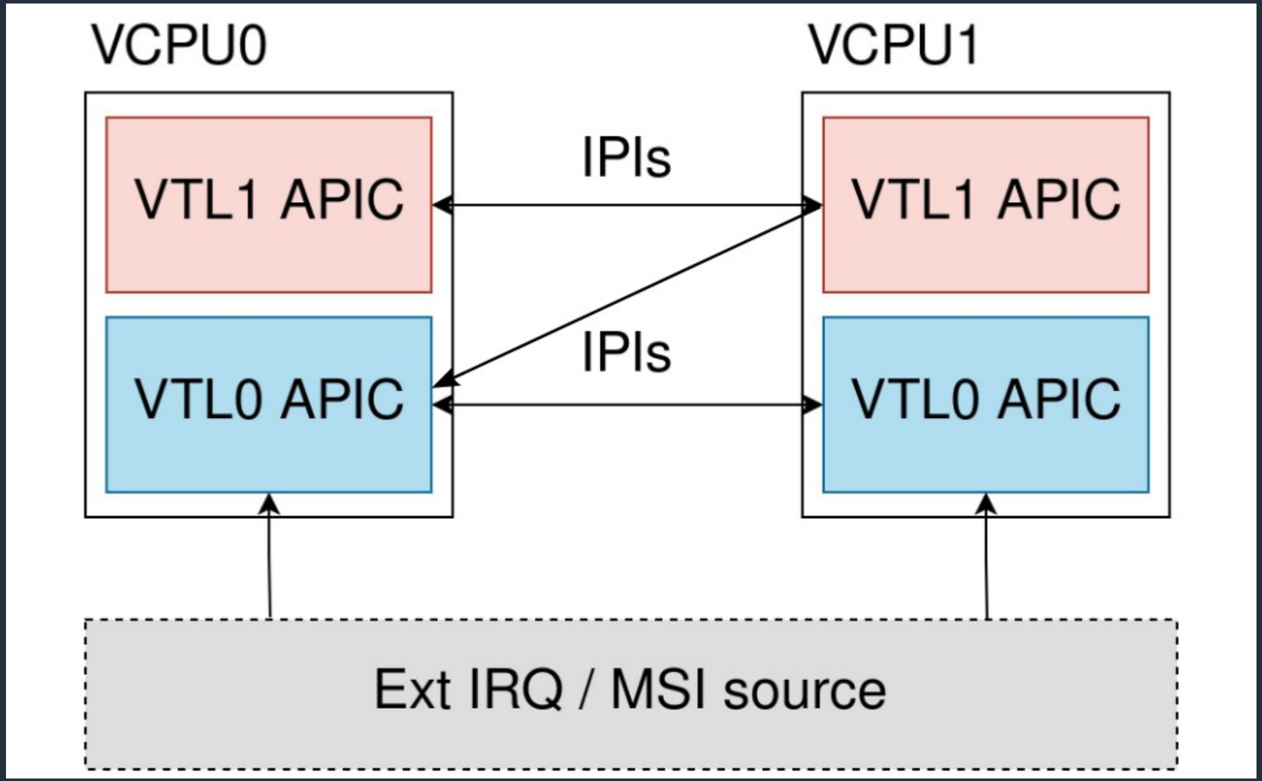
VTL switching can happen through interrupts and intercepts.

Interrupts are delivered to the same VTL that originated them.

VTL1 interrupts have precedence over VTL0 execution.

If VTL1 is running while a VTL0 interrupt is injected, the interrupt is saved and triggered on VTLRETURN.

External interrupts target VTL0.



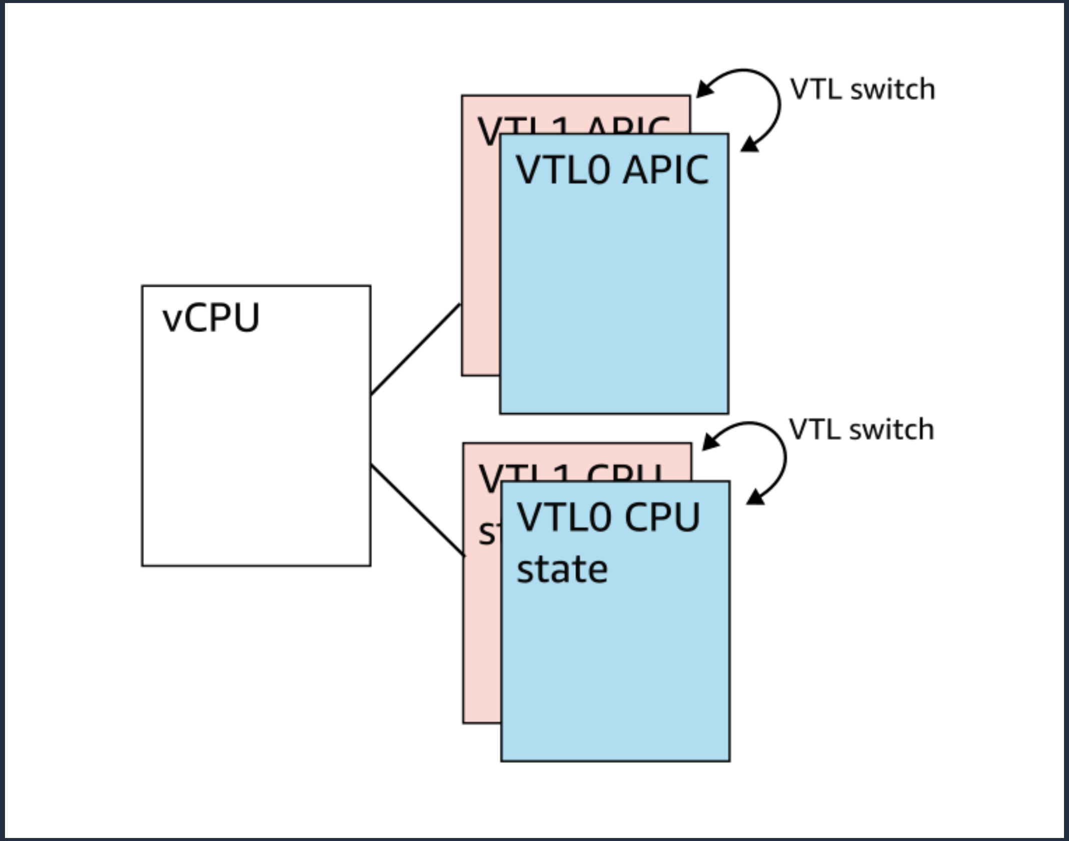
VTL Switching (implementation notes)

The vCPU state was extended to have per-VTL sets of private registers and MSRs.

vCPUs maintain an interrupt controller per each VTL.

This breaks the core assumption of vCPUs having one set of registers, MSRs and an interrupt subsystem, the result complicates core x86 KVM code and is hard to reason about.

It also doesn't fit the current vCPU/APIC state serialization/deserialization APIs.



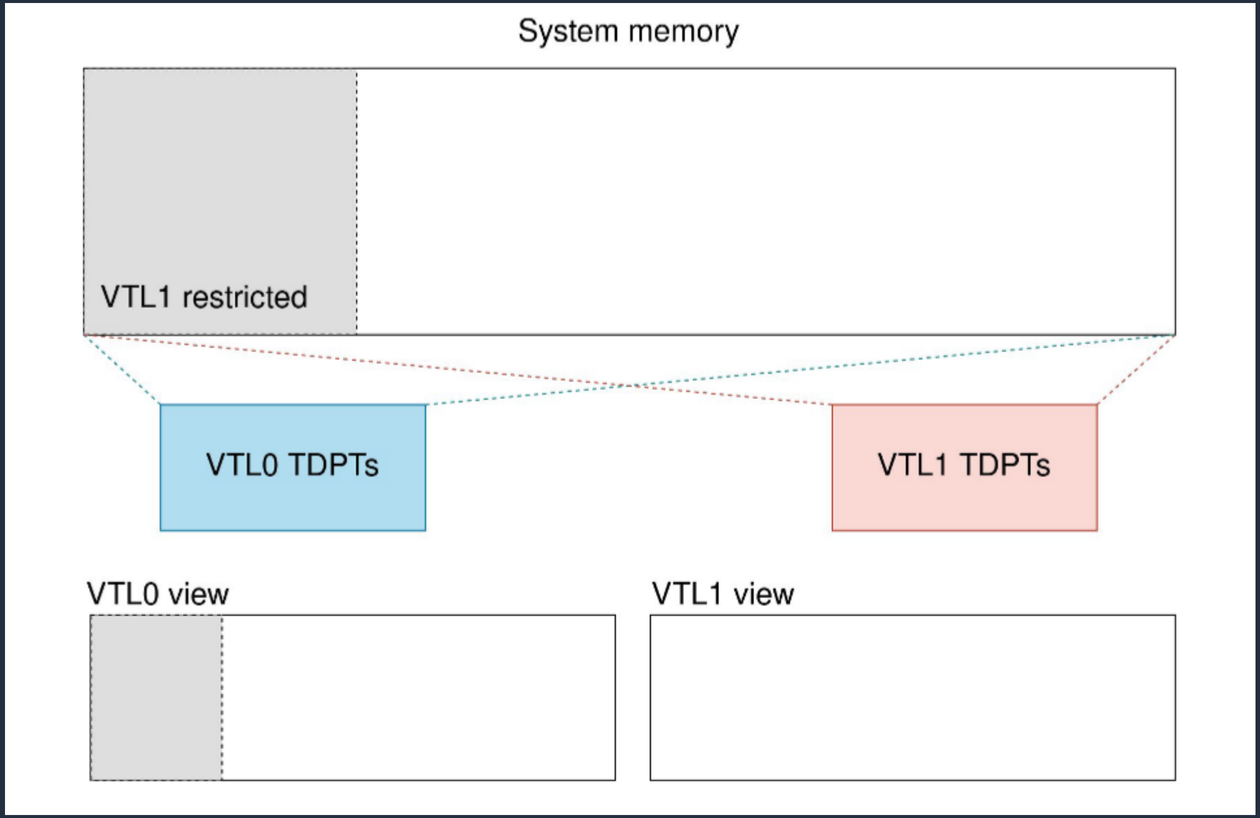
VTL Memory Protections

Memory accesses are now contextualized by the VTL level of the CPU is running on.

GPA->HPA map is still common regardless of CPU and VTL (except overlay pages).

VTL1 can request the hypervisor to revoke R/W/X(+MBEC) access from VTL0.

If VTL0 accesses protected memory, the hypervisor injects an intercept into VTL1.



VTL Memory Protections (implementation notes)

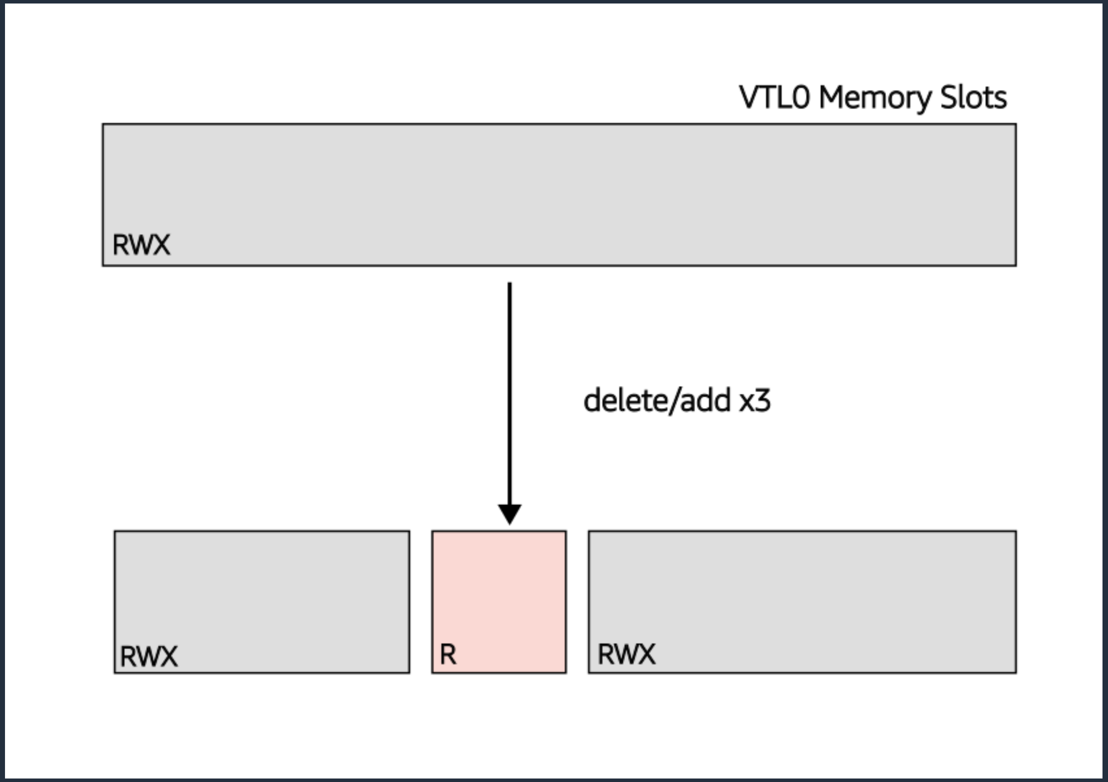
Each VTL operates in its own address space for KVM memory slots, similar to SMM.

Memory protection changes are channeled through memory slot modifications.

These modifications require either serialization across vCPUs or atomic slot update support.

We observed 100,000+ protection changes during boot-time.

We introduced atomic slot updates with targeted zapping.



VTL I/O Memory Protections

All I/O memory accesses happen in VTL0.

Most HW is unable take I/O page faults.

Protection changes happen concurrent to I/O, all IOMMU page tables updates have to be atomic and the mappings remain valid at all times.

HW should support *no-access* mappings, if not the alternative is redirecting I/O into a *scratch page*.

VTL I/O Memory Protections (implementation notes)

We introduced a new IOMMU driver *remap* callback.

We implemented for both Intel and AMD. It’s the only place where vendor specific code was necessary.

remap also merges IOVA ranges into huge pages when possible to avoid fragmentation.

It’s exposed through a new VFIO ioctl.

Upstreaming VSM

Path to upstream

Work-in-progress code available in the following repositories (see *vsm* branch):

https://github.com/vianpl/linux

https://github.com/vianpl/qemu

https://github.com/vianpl/kvm-unit-tests

The code is in a very rough state, is insecure, but it can can boot CG with a fully upstream virtualization stack.

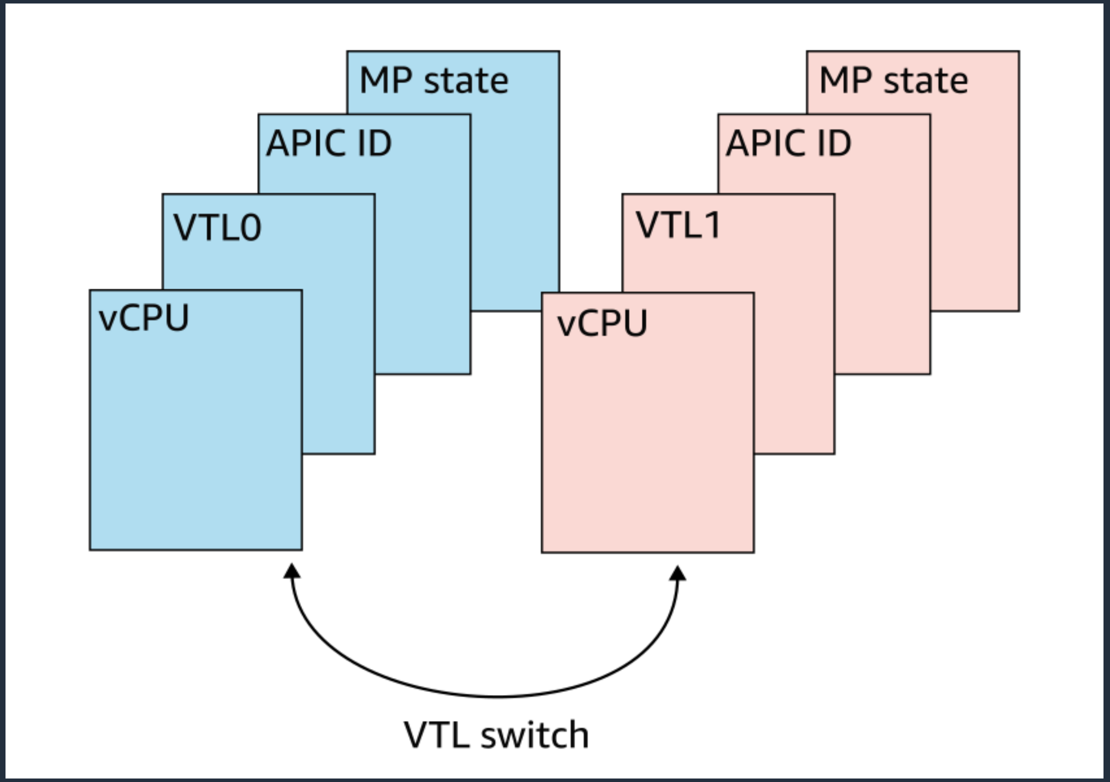
Path to upstream

Three challenges: handling VTL state in a sane way, memory protections, and I/O memory protections.

Having a vCPU per VTL should simplify VTL state handling.

Memory protections could piggyback on the proposed *MEMORY_ATTRIBUTES ioctl*. Although, it cannot target different VTLs.

For I/O memory protections, we will rebase our IOMMU driver work and propose an IOMMUFD Interface.



Thanks!

References

Hypervisor Top Level Functional Specification, v6.0b. Microsoft Feb, 2020 ([GitHub])

Battle of the SKM and IUM: How Windows 10 Rewrites OS Architecture. Alex Ionescu, Chief Architect, CrowdStrike ([Black Hat USA 2015])

AWS official Credential Guard documentation (*docs.aws.amazon.com*)

Dead slides

Per VTL State

Private

SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, STAR,
LSTAR, CSTAR, SFMASK, EFER, PAT,
KERNEL_GSBASE, FS.BASE, GS.BASE, TSC_AUX
HV_X64_MSR_HYPERCALL, HV_X64_MSR_GUEST_OS_ID
HV_X64_MSR_REFERENCE_TSC, HV_X64_MSR_APIC_FREQUENCY
HV_X64_MSR_EOI, HV_X64_MSR_ICR
HV_X64_MSR_TPR, HV_X64_MSR_APIC_ASSIST_PAGE
HV_X64_MSR_NPIEP_CONFIG
HV_X64_MSR_SIRBP, HV_X64_MSR_SCONTROL
HV_X64_MSR_SVERSION, HV_X64_MSR_SIEFP
HV_X64_MSR_SIMP, HV_X64_MSR_EOM
HV_X64_MSR_SINT0 – HV_X64_MSR_SINT15
HV_X64_MSR_STIMER0_CONFIG –
HV_X64_MSR_STIMER3_CONFIG
HV_X64_MSR_STIMER0_COUNT – HV_X64_MSR_STIMER3_COUNT

Local APIC registers (including CR8/TPR)
RIP, RSP
RFLAGS
CR0, CR3, CR4
DR7
IDTR, GDTR
CS, DS, ES, FS, GS, SS, TR, LDTR
TSC

Shared

HV_X64_MSR_TSC_FREQUENCY
HV_X64_MSR_VP_INDEX
HV_X64_MSR_VP_RUNTIME
HV_X64_MSR_RESET
HV_X64_MSR_TIME_REF_COUNT
HV_X64_MSR_GUEST_IDLE
HV_X64_MSR_DEBUG_DEVICE_OPTIONS
HV_X64_MSR_BELOW_1MB_PAGE
HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE
HV_X64_MSR_STATS_VP_RETAIL_PAGE
MTRRs
MCG_CAP
MCG_STATUS

Rax, Rbx, Rcx, Rdx, Rsi, Rdi, Rbp
CR2
R8 – R15
DR0 – DR5
X87 floating point state
XMM state
AVX state
XCR0 (XFE)