# 0. 摘要

虚拟机级别的容器中，每个容器运行在虚拟机虚拟出的独立内核上，因此隔离性很强。但其依赖于通用虚拟机虚拟出的虚拟化硬件，与 OS 级别的容器相比，会导致不可忽略的性能开销。而在嵌套虚拟化场景下，secure container 运行在虚拟机中，这个性能的 gap 会显著地扩大。

本篇文章基于两个角度提出容器内核隔离（CKI），一个软硬协调的高效机密容器设计。

- 首先，Protection Keys for Supervisor（PKS）可以帮助我们构建一个新的权限级别，用于在 Host Kernel 中安全地配置多个容器内核，而不涉及 non-root ring 0（Intel 中的 Guest Kernel 所处级别）；
- 其次，secure container 使用的通用虚拟化技术提供很多容器实际隔离并不需要的特性，例如二阶段页表翻译，这引入了可以避免的性能开销；

因此容器内核隔离技术在跑容器内核时：

1. 避免使用虚拟化硬件，并移除不必要的虚拟化技术（像二阶段地址翻译）。它使用 PKS 来构建一个新特权级，用来隔离不同的容器内核，并且提供了跨特权级更高效的交互方式；
2. 给每个容器内核使用了 single-stage address translation，并且用轻量级的方式监听这些页表，来确保跨容器虚拟内存隔离；

实机实验证明了 CKI 技术的高效性，结果显示对于内存密集型应用相对于硬件辅助虚拟化 hardware-assisted virtualization (HVM) 和基于软件虚拟化 software-based virtualization (PVM) 技术分别提升了 72% 和 47%；

# 1. 介绍

由于容器具有可移植性和可扩展性等优势，因此被广泛应用于云中构建和部署应用程序。有两种典型的容器架构：

- OS 级别容器。由于互不信任的容器之间共享操作系统内核中的漏洞，操作系统级容器因安全性差而饱受诟病。恶意的操作系统级容器可能会利用系统调用接口暴露的巨大攻击面来逃避隔离；
- VM 级别容器。相比之下，虚拟机级容器在自己的客户操作系统内核上运行每个容器，具有更强的安全隔离性，在云计算中越来越受欢迎。破坏客户内核对主机内核或其他容器无害。

  尽管进行了许多优化，但与操作系统级容器相比，虚拟机级容器仍显示出性能上的劣势，这是因为涉及到为通用虚拟机设计的虚拟化硬件。例如，遍历 walk 过 two-stage page table 可将内存密集型应用的延迟平均增加 46%；

此外，性能差距随着嵌套虚拟化的增加而增大。据谷歌和阿里巴巴等知名云提供商称，基于公共基础设施即服务（IaaS）的云构建服务 (building cloud services) 的需求日益增长。在这种嵌套云中，虚拟机级容器必须在虚拟机内运行，由于 L2 VM（容器）、guest hypervisor（L1 内核）和 host hypervisor（L0 内核）之间的上下文切换过多，导致运行时开销很大。根据我们的评估，这种开销使内存密集型和 I/O 密集型应用的性能分别降低了 28%~226% 和 1.8×~4.3× 。

我们认为，性能开销源于 **secure container 架构所需的权限级别与 CPU 硬件所提供的权限级别之间的不匹配**。具体来说，主机内核需要隔离多个容器访客内核，而每个访客内核又需要隔离多个容器应用进程。因此，需要三个权限级别。然而，CPU 硬件通常为运行操作系统内核和应用程序提供两种权限级别，例如 x86 ring-0/ring-3 和 Arm EL1/EL0。因此，现有的安全容器（如 Kata Containers 和 Firecracker）利用硬件虚拟化扩展来获得额外的权限级别，从而导致性能开销。

一些 secure container 架构，如 PVM 和 gVisor，不需要虚拟化硬件。它们将容器 guest kernel 授权为用户模式，并将 guest kernel 和 container app 隔离在不同的地址空间中。不过，这些架构会产生更多上下文切换开销，因为容器内的系统调用和异常一定会 **redirect** 给 **host kernel**。例如，空系统调用在操作系统级容器中需要 90 ns 的时间，而在 PVM 容器中则需要 336 ns。

在本文中，我们基于两个观点提出了一种新的安全容器设计，称为 CKI（容器内核隔离）：

1. 首先，最近的 CPU 功能（Protection Keys for Supervisor，PKS 或 MPK）可以改造为在 kernel mode 中创建另一个特权级别，以容纳 guest（容器）kernel。这一新的权限级别允许客户内核有效地为其容器应用服务，从而带来性能上的优势，例如将裸机云和嵌套云中的上下文切换最小化；
2. 其次，为每个容器分离内核是为了安全隔离，而不是通用虚拟化。具体来说，内存虚拟化中使用的两阶段地址转换机制为任何客户内核提供了透明的物理地址空间，确保了兼容性，但这与容器隔离要求无关。因此，支持任意虚拟机的虚拟化机制是不必要的（例如二阶段地址翻译可以被移除，以此提升性能）。

但在创建新的权限级别时，CKI 面临着三个挑战。我们逐个应对：

1. 首先，PKS 仅用于内存隔离，而恶意 guest kernel 在 kernel mode 下执行时，可能会执行任意特权指令。为了解决这个问题，我们提出了针对 PKS 的轻量级硬件扩展，以实现指令隔离；
2. 其次，CKI 中 PKS switch gates 的安全目标，即防止容器逃逸或主机拒绝服务（host DoS），需要在基本 **MPK** 门之外进行额外设计。例如，switch gates 需要新的机制来定位每 vCPU 区域，防止中断垄断或伪造；
3. PKS 在一个内核地址空间内只支持 16 个内存域，而一台机器可能承载数十到数百个安全容器。为了支持任意数量的容器，CKI 结合了 **PKS** 和地址空间隔离技术，以隔离不同的客户内核。

   具体来说，它为每个 guest kernel 创建了一个单独的地址空间，并在每个地址空间中映射一个内核安全监控器（KSM, kernel security monitor），然后使用 PKS 将 KSM 与 guest kernel 隔离。PKS 隔离会剥夺 guest kernel 的权限，使其只能通过 KSM 或 host kernel 提供的接口执行特权操作。KSM 实现的特权操作只能访问一个安全容器的私有数据（如页表更新），这些操作可通过快速 PKS 开关门调用。

   我们只会在 KSM 中映射这些私有数据，这样 PKS gate 在切换时就不需要进行侧信道保护（PTI, IBRS, ...），这能节省上百个时钟周期；

最后，我们实现了 CKI 的原型，并利用真实世界的容器应用对其进行了评估。我们在裸机云（bare-metal）和嵌套云中对 CKI 进行了评估，并将其与硬件辅助虚拟化（HVM）和基于软件的虚拟化（PVM）进行了比较。在裸机云中，与 HVM/PVM 相比，CKI 可将内存密集型应用的延迟时间最多减少 18%/47%。在嵌套云中，与 HVM/PVM 相比，CKI 最多可将内存密集型应用的延迟降低 72%/47%，并且与 HVM/PVM 相比，I/O 密集型应用最多可获得 6.8 倍/1.2 倍的吞吐量。

总之，本文做出了以下贡献：

- 全面探索了安全容器的设计空间，揭示了 CPU 权限级别与容器内核分离需求之间的不匹配；
- 一种名为 CKI 的新型安全容器设计，通过引入软硬件共同设计的权限级别实现了高效的内核分离；
- 一个系统原型，其在实际应用中的实验结果证明了它的有效性。

# 2. 背景和动机

## 2.1 Secure Container Models

操作系统级容器通过在多个容器之间共享单个操作系统内核来实现轻量级隔离。然而，操作系统级隔离机制很薄弱，因为在 Linux 等商品级操作系统内核中发现了许多漏洞。复杂的 syscall interface 向 userspace 暴露了巨大的攻击面，容器可利用这些攻击面进行权限升级、信息泄漏或拒绝服务（DoS）。

而 secure container 通过限制 OS kernel 被 compromised 的影响来加强容器隔离。这可以通过两种容器架构来实现： 虚拟机级容器和基于 enclave 的容器，如图所示。

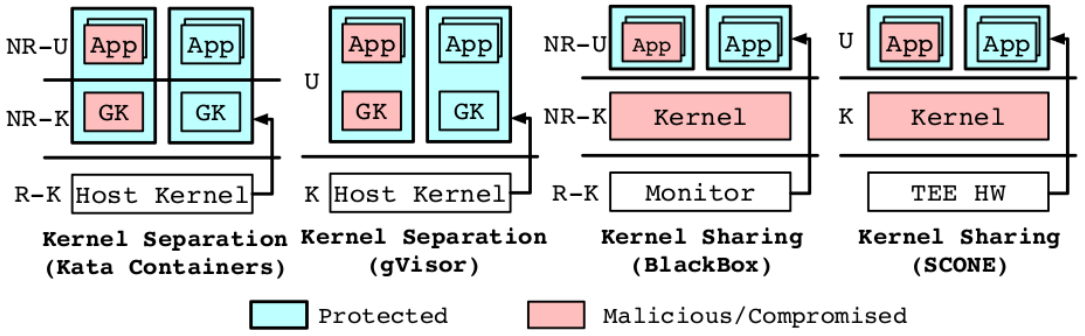



**Figure 1.** Secure containers: VM-level (kernel separation) and enclave-based (kernel sharing). GK: guest kernel, NR: non-root, R: root, U: ring-3, K: ring-0.

- 虚拟机级容器。虚拟机级容器在单独的 guest kernel 上运行每个容器。它们要求用户应用程序、guest kernel 和 host kernel 具有三级权限。容器内的恶意应用程序可利用内核漏洞入侵其 guest kernel，但这对 host kernel 或其他容器无害。guest kernel 与 host kernel 之间的接口可能比系统调用接口简单得多，因此恶意 guest kernel 很难入侵 host kernel。
- 基于 enclave 的容器。和 OS 级容器一样，基于 enclave 的容器在共享的 kernel 上运行所有容器。但是 Secure Monitor（BlackBox，有一个运行在 Root Kernel 中的监控器，类似虚拟机监控器）或基于硬件的可信执行环境（TEE），都会剥夺共享内核任意访问受保护容器的内存数据或执行上下文的权限。

不过，如果考虑了 CVE 之后，虚拟机级容器就是首选方案了。在收集近两年（2022-2023 年）可在容器中利用的 Linux 内核 CVE，并按安全影响对其进行分类（共 209 个）后发现，这些 CVE，97.3% 可导致 DoS 攻击，包括破坏系统状态（如越界写入、use-after-free）、导致不可恢复的错误（如空指针延迟、kernel panic）或独占硬件资源（如内存泄漏、死锁）。虽然 enclave-based 的容器可以保护容器数据的机密性和完整性，但由于内核共享设计，它们无法抵御 DoS 攻击。相反，虚拟机级容器由于采用了内核分离设计，可以防止 DoS。

## 2.2 Secure Containers in Nested Clouds

根据先前的研究，在基础设施即服务（IaaS）云租用的虚拟机中构建容器平台的需求日益增长。在这些嵌套云中，VM 级容器部署在另一个虚拟机内。例如，阿里巴巴云正在将 secure containers 从 bare-metal 实例转移到通用（虚拟化）实例，以实现更高的隔离度、节约成本以及更灵活、更有弹性的 Kubernetes 集群管理。谷歌的 gVisor 通常会考虑虚拟机内部署，并设计相应的优化措施。

在 IaaS 虚拟机内运行安全容器需要嵌套虚拟化，其中一个 L0 内核（host hypervisor）运行一个带有 L1 内核（guest hypervisor）的虚拟机，然后隔离多个安全容器（L2 虚拟机，guest hypervisor 上的 VM），每个容器都有自己的 L2 内核。内存密集型和 I/O 密集型容器应用都可以在嵌套虚拟化下部署。我们的目标是减轻这些应用的嵌套虚拟化开销。

## 2.3 Memory Protection Keys

Memory Protection Keys，内存保护密钥（MPK）是 x86 CPU 上用于内存隔离的新硬件功能。MPK 将虚拟地址空间中的页面最多分为 16 个域，并利用页表项 (PTE) 中以前未使用的四个位来表示每个页面的 domain ID。它还引入了一个 32 位 per-core protection key register，用于配置每个域的访问权限。权限可设置为只读、读写或不可访问。

MPK 有两种变体： Protection Keys for Userspace（PKU）控制用户页面的权限，而 Protection Key for Supervisor（PKS）控制内核页面的权限。PKU 和 PKS 的 protection key register 分别称为 PKRU 和 PKRS。PKRU 可以使用名为 `wrpkru`（write PKRU）的高效指令进行配置，而 PKRS 则可以使用 `wrmsr`（write MSR）指令进行配置。

## 2.4 Issues of VM-Level Containers

虚拟机级容器需要三个权限级别，而 CPU 硬件只为内核和应用程序提供两个权限级别。虚拟机级容器的一个问题是缺少第三个 CPU 权限级别来有效地适应客户（容器）内核。具体来说，下图展示了现有的虚拟机级容器设计，表格则显示了这些设计的比较。现有设计存在的问题归纳如下。

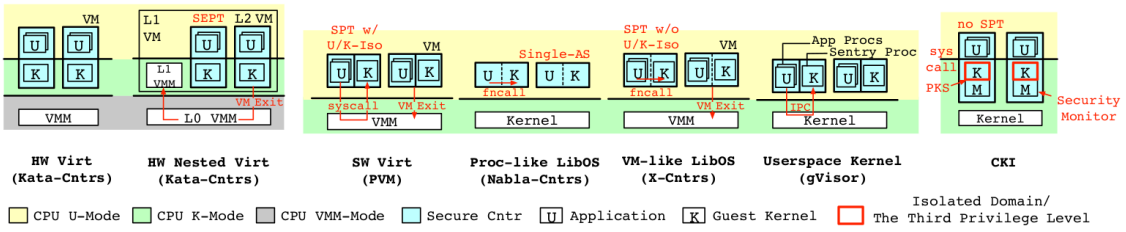



**Figure 3.** Comparison of different VM-level containers. Cntr: container, S(E)PT: shadow (extended) page table, AS: address space, fncall: function call, U/K-Iso: user/kernel isolation, VMM: hypervisor.

**Table 1.** Comparison of different VM-level containers. BM: bare-metal, NST: nested.

| Aspect | | HW-Assisted VM | SW-Based VM | Proc-Like LibOS | VM-Like LibOS | Userspace Kernel | CKI |
|---|---|---|---|---|---|---|---|
| Performance | Memory Intensive Applications | ◐ (BM) ◔ (NST) | ◐ | ● | ◐ | ● | ● |
| | I/O Intensive Applications | ● (BM) ◔ (NST) | ◐ | ● | ● | ◐ | ● |
| Security | Guest User-Kernel Isolation | ● | ● | ○ | ○ | ● | ● |
| Compatibility | Nested Cloud Deployment | ◐ | ● | ● | ● | ● | ● |
| | Container Binary Compatibility | ● | ● | ◐ | ● | ◔ | ● |

- 硬件辅助虚拟化（HVM）通过专用的虚拟机控制结构（VMCS）和 EPT 隔离了 guest kernel，导致 EPT 转换和管理造成内存性能不理想，以及虚拟机退出缓慢造成嵌套云中 I/O 性能不佳；
- 基于软件的虚拟化（PVM）将 guest kernel 剥夺为用户模式，并使用影子分页隔离容器内存，导致在处理系统调用时出现额外的上下文切换，以及页表更新效率低下；
- 基于 LibOS 的容器打破了应用程序与客户内核之间的隔离，降低了安全保证并导致兼容性问题。

### 2.4.1. HVM 的问题

The container design with HVM faces **both performance and compatibility issues**.

- **EPT 的开销**。
  - 在裸机云中，HVM 处理一次 EPT fault 需要 3µs 的时间，与操作系统级容器相比，这可能会使内存密集型应用的延迟增加 2%~21%。HVM 还会因 two-staged page table walk 导致昂贵的 TLB miss 处理，内存密集型应用的平均超限率为 46%。

    - 在嵌套云中，由于没有硬件支持三级地址转换，L1 内核依赖 L0 内核为每个 L2 虚拟机维护一个影子 EPT（SPTE），从而导致 EPT 管理的高开销。

      > 这不就丢失了一部分半虚拟化的优势了吗？

    - 在嵌套云中，HVM 容器中的页面故障需要 32µs 以上的时间。与操作系统级容器（图 4 中的 HVM-NST / RunC-BM）相比，这使内存密集型应用的延迟增加了 28%~226%。
- 虚拟机退出重定向的开销。
    - 在嵌套云中运行 HVM 容器时，L1 内核和 L2 VM 使用不同的 VMCS 执行，导致 L0 对 VM 退出进行干预。具体来说，当 L2 虚拟机发生 VM Exit 时，会触发一个 trap 到 L0 内核，然后 L0 内核恢复 L1 内核来处理 L2 虚拟机退出。L1 内核处理完虚拟机退出后，再次向 L0 内核发出陷阱，然后 L0 内核恢复 L2 虚拟机。

      在裸机云中，HVM 容器中的空的 hyper-call 只需 1.1us，而在嵌套云中则需要 6.7us。与消除了 L0 干涉的 PVM 相比，这种虚拟机退出开销将 I/O 密集型应用的吞吐量降低了 1.8×~4.3× 倍；
- 兼容性问题。HVM 在嵌套云中也面临兼容性问题。首先，一些 IaaS 云禁用硬件辅助嵌套虚拟化，以减少 L0 内核的攻击面。其次，新兴的 Confidential VM（CVM）不支持硬件辅助嵌套虚拟化，因为 L0 内核不受信任；

### 2.4.2. PVM 的问题

PVM 利用半虚拟化技术实现 VM 级容器。它以用户和内核模式分别运行容器应用程序和 host kernel。它还在单独的地址空间内以 user mode 运行 guest（容器）kernel。PVM 通过避免虚拟机退出到 L0 内核，在嵌套云中获得了比硬件辅助虚拟化（HVM）更好的性能。

- 系统调用重定向的开销。当应用程序调用系统调用时，它 trap 给 host kernel。然后，host kernel 会切换到 guest kernel 的 page table、返回用户态、调用 guest kernel 的 syscall handler。当处理结束后，会以一个相反的路径返回用户态应用程序。

  与本地系统调用相比，该系统调用过程增加了两个 CPU 模式开关和两个页表开关。这将系统调用延迟从 90ns 增加到 336ns；

  与裸机云中的 HVM 相比，I/O 密集型应用的平均开销为 6.6%。
- **shadow paging 的开销**。PVM 通过使用影子分页机制，保留了两阶段地址转换的灵活性：GVA -> GPA -> HPA。Host 为每个容器应用维护一个 SPT（将 GVA 转换为 HPA）。一个容器 page fault 涉及 host kernel 和 guest 之间至少 6 次上下文切换。两次切换用于将 page fault 重定向到 guest kernel，两次用于更新 shadow page table（SPT 由 host 维护 read-only），另外两次用于返回用户应用。

  此外，页面故障处理过程中的仿真逻辑也会产生很高的开销，例如 page table walking（需要确认在哪个翻译阶段）、指令仿真、影子页表管理和异常注入。PVM 容器中的页面故障需要 7µs 的时间，而本地页面故障只需要 1µs 的时间。与操作系统级容器（PVM-BM / RunC-BM）相比，影子分页的开销使内存密集型应用的延迟增加了 6%~73%。

### 2.4.3. LibOS 和 gVisor 的问题

基于 LibOS 的 secure container **将 LibOS attach 到进程或虚拟机中的每个容器上**。它们在 secure container 内不执行用户-内核隔离（user/kernel isolation），而是**在同一地址空间运行应用程序和 libOS**。这种设计避免了系统调用处理过程中的页表切换，但削弱了容器的隔离保证。同时，它们通常兼容性较差，例如无法完全支持容器中的多进程。

> 优点：避免页表切换；缺点：削弱隔离性、安全性，不保证兼容性（如多进程）；

gVisor 实现了一个名为 Sentry 的新用户空间内核，每个容器都运行在一个私有的 Sentry 实例上。gVisor 让 host kernel 处理应用程序的 page fault，避免了影子分页的开销。不过，由于涉及 IPC，Systrap 比本地系统调用慢得多。同时，作为一个重新实现的内核，gVisor 可能缺乏 Linux 内核的完全兼容性和优化功能。

> 优点：避免 shadow page 开销，但是 IPC 导致性能下降，并且 Sentry 缺乏 Linux 一样的兼容性和优化能力；

# 3. 解决方案总览

## 3.1 设计内涵和选择

设计内涵：根据第 2.4 节中的分析，为容器 guest kernel 高效构建新权限级别有两个设计含义（为什么要为容器 guest kernel 构建一个新特权级？）：

1. 权限级别之间的高效切换。由于 guest kernel 经常与应用程序通信（即系统调用/异常）并执行特权操作（如页表更新和 I/O 请求），我们应尽量减少这些过程中的上下文切换开销。应避免 PVM 中的系统调用重定向或嵌套 HVM 中的虚拟机退出重定向等过度开销；
2. 无两阶段地址转换的内存隔离。两阶段地址转换是为通用虚拟化设计的，超出了容器隔离的需求，因为容器不依赖特定的物理内存布局。使用单级转换可以避免影子分页或 EPT 转换/管理的开销。

设计选择：MPK 可在单个 CPU 权限级别内执行高效的域隔离，这可用于构建新的权限级别。有两种可能的设计：

1. 在用户模式下运行 guest kernel，并使用 PKU（Design-PKU）将其与应用程序隔离；
2. 在内核模式下运行 guest kernel，并使用 PKS（Design-PKS）将其与 host kernel 隔离。

这两种设计都支持无需重定向的高效系统调用。它们还能避免嵌套云中的虚拟机退出重定向，因为 guest kernel 和 host kernel 使用相同的 VMCS 执行。

我们选择 Design-PKS 而不是 Design-PKU，原因如下。

- 首先，由于 PKU 已被广泛用于各种应用（进程内隔离），Design-PKU 本身会干扰这些现有用例。这种冲突破坏了 PKU 在用户空间应用中的初衷；
- 其次，Design-PKU 需要替换容器应用中的 `wrpkru` 指令和系统调用指令。然而，正如 Hodor 所强调的，在没有源代码的情况下对任意（容器）镜像进行二进制重写可能是不可判定的，这可能会破坏容器二进制的兼容性或妨碍应用功能（如及时编译）；
- 第三，Design-PKU 在异常处理中会产生额外的性能开销。例如，从 host 向 guest kernel 注入 page fault 需要额外的跨特权级切换，在我们的测试平台上，page fault latency（原本约为 1,000ns）增加了约 750ns；

## 3.2 基于 PKS 隔离的挑战

PKS 最初并不是为隔离容器 guest kernel 而设计的，这导致我们的设计面临以下挑战：

1. 隔离域数量不足。PKS 在一个地址空间中最多只能支持 16 个域，远远低于安全容器的潜在数量。因此，在专用的 PKS 域中隔离每个客户内核是不可行的；
2. 缺乏特权指令隔离。PKS 只提供内存隔离，而在内核模式下运行的恶意 guest kernel 可以使用特权指令破坏隔离。二进制重写是从隔离软件中删除特定指令的常用技术。然而，要消除操作系统内核中未对齐位置的所有特权指令是不可行的。

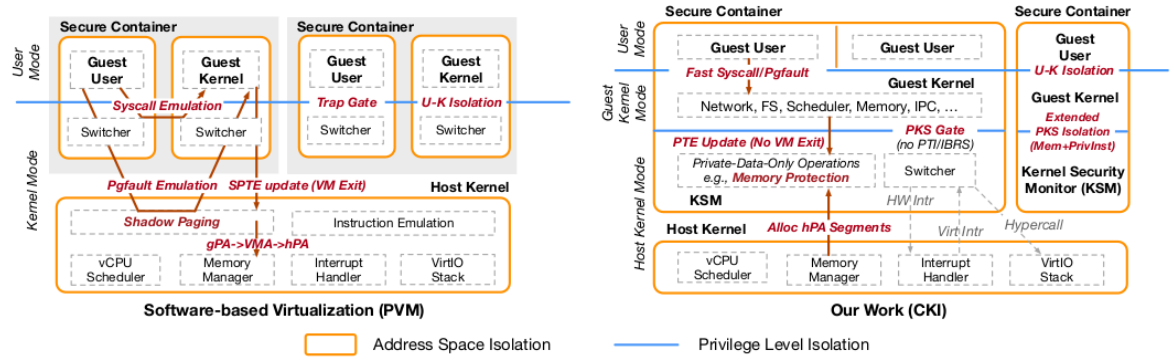
对于嵌套云，一种潜在的解决方案是使用虚拟化硬件拦截和监控 L1 虚拟机中的所有特权指令，但这需要对 L0 内核进行侵入式修改，可能并不可行。

3. **switch gate 功能不完整**。在基于 PKS 的隔离下，guest kernel 使用 PKS switch gate 与 host kernel 通信。但是，这种 switch gate 需要 PKS 原本不支持的功能。例如，host kernel 需要在 guest 执行期间使用 switch gate 拦截硬件中断，但原本实现的 switch gate 设计可能会让 guest kernel 向 host kernel 注入假中断。

> 通俗来说，硬件中断（比如键盘输入、网络数据到达等）需要由主机内核统一处理，这个过程要通过 switch gate 机制来完成。但如果这个机制设计得太简单，恶意的客户内核就可能模仿硬件中断的信号格式，伪造出假的中断请求发送给主机内核。
>
> 可能导致的问题，一是主机内核被大量虚假请求干扰，无法处理真实的硬件中断（DoS）；二是假中断可能携带恶意指令，导致主机内核的安全机制被绕过；

# 3.3 设计架构总览

CKI 是一种虚拟机级容器架构，每个容器都运行在独立的内核上。它在不使用硬件虚拟化扩展的情况下实现了三种权限级别。PVM 是最先进（SOTA）的安全容器设计，不使用虚拟化硬件。下图显示了 CKI 的架构及其与 PVM 的不同之处。简而言之，CKI 避免了基于软件的虚拟化的系统调用重定向和影子分页的开销，因此无论在裸机还是嵌套云中都能获得更好的性能。




## 3.3.1 抽象

每个 CKI 安全容器就像一个虚拟机，有一个 guest kernel 和多个用户进程，运行在 host kernel 上。guest kernel 提供操作系统功能，如内存管理、调度、文件系统和网络堆栈。host kernel 调度 guest 的 vCPU，为其分配内存，并使用 VirtIO 协议为 guest kernel 模拟虚拟设备（磁盘和网卡）。**所有硬件中断都由 host kernel 处理**。当 guest kernel 需要调用 host 功能或发生硬件中断时，guest vCPU 会通过一段名为 switcher 的上下文切换代码退出到 host kernel（第 4.2 节）。**对于嵌套虚拟化，CKI 虚拟机退出过程不涉及 L0 干预**（guest kernel 和 host kernel 均位于内核态）。host kernel 在恢复 guest vCPU 时可能会注入虚拟中断。

## 3.3.2 与 PVM 的区别

CKI 与基于软件的虚拟化（PVM）有两个主要区别。

- 首先，CKI 在内核模式下以新的权限级别运行 guest kernel，使客户用户无需 host kernel 干预即可调用系统调用。**guest kernel 的内存映射在 guest 用户地址空间中，并通过 PTE U/K 位隔离**，从而消除了系统调用时的页表切换；

> 不加 KPTI 的时候，Linux 不也是把 kernel 的内存映射到用户地址空间的吗？

- 其次，**CKI 不实现两阶段地址转换。host kernel 为每个 guest VM 提供一些连续的 HPA 段**，由 guest kernel 中的内存管理器直接管理。因此，guest user page fault 可由 guest kernel 直接处理，而不是在 host kernel 中触发 shadow page fault。通过消除虚拟机退出、GPA 到 HPA 转换和影子 PTE 生成，PTE 更新操作也得到了简化。

### 3.3.3 定义新的权限级别（Guest Kernel Mode）

CKI 利用基于 PKS 的内核内部隔离来构建新的权限级别，以消除 guest kernel 的权限。一方面，它将 PKS 隔离与地址空间隔离相结合，限制 guest kernel 的内存访问权限（PKS 原本就进行内存隔离，但不提供特权指令隔离）。另一方面，它还能监控来自 guest kernel 的特权操作的执行情况（需要扩展，参见下文）。

具体来说，不同的 secure container 和 host kernel 被隔离在不同的地址空间。每个 guest kernel 都是非特权内核，与特权内核安全监控程序（KSM）一起运行，两者运行在同一地址空间，但在 PKRS（内核页保护密钥权限寄存器）中指定的 PKS 权限不同。KSM （PKRS 为零）可以访问所有虚拟内存，而 guest kernel（PKRS 为 `PKRS_GUEST`）不能访问其 KSM 的内存。

> 因此，在每个 secure container 的地址空间内，guest kernel 和 KSM 只需要两个 PKS 域。因此，CKI 可以支持任意数量的安全容器，而不受 PKS 域限制的影响（克服挑战-1）。

此外，CKI 还为 PKS 增加了一个轻量级硬件扩展，使特权指令在 guest kernel 中不可执行（参见 4.1）（克服了挑战-2）。

guest kernel 只能通过其 KSM 或 host kernel 提供的预定义接口执行特权操作。KSM 实现的特权操作只能访问安全容器的私有数据，例如页表更新（参见 4.3，这不需要下陷到 L0）和 `iret` 指令。这些特权操作可通过高效的 **PKS gate（guest kernel 与 KSM 之间的切换）调用**（参见 4.2）。由于只有私有数据才会映射到 KSM 中，CKI 从 PKS gate 中消除了代价高昂的侧信道缓解方案（如 PTI 和 IBRS）。

其他特权操作（如 VirtIO MMIO、定时器设置、`hlt` 指令）依赖于全局数据（如驱动程序/调度程序元数据），由 host kernel 提供。guest kernel 可通过专门设计的 switcher（guest kernel 与 host kernel 之间的切换器）调用此类操作。switcher 还包含中断门，可在 guest VM 执行期间拦截硬件中断，并将其重定向到 host kernel。CKI 依靠多种技术防止中断垄断和中断伪造（参见 4.4）（克服挑战-3）。

## 3.4 威胁模型

CKI 继承了 VM 级容器的威胁模型。host kernel 和 KSM 隔离多个容器，而容器中的 guest kernel 则隔离多个用户进程。一个容器可能会被入侵，然后试图打破容器间隔离，例如执行破坏性特权指令或破坏关键内存结构（如页表、IDT）。由于 KSM 和 host kernel 的攻击面较小（hypervisor 接口），因此假定它们是可信的。

单个安全容器内的瞬时执行攻击不在攻击范围内。容器间的瞬时执行攻击可通过在每个容器自己的地址空间中运行并在 host kernel 中启用 Spectre 缓解功能来缓解。

# 4. 设计细节

## 4.1 基于 PKS 的特权指令隔离

出于性能考虑，CKI 选择在内核模式下构建新的权限级别，也就是说，容器用户进程可以以原有的高效方式与（容器）guest kernel 进行交互。例如，进程仍可通过 syscall 指令直接调用系统调用，而无需额外的上下文切换。

由于 guest kernel 不受信任，CKI 需要防止它执行特权指令，从而破坏安全隔离。目前的 PKS 硬件功能只能在内核模式下提供内存隔离，无法限制特权指令的执行。此外，现有的基于软件的指令隔离技术（如二进制重写）也不适用于 CKI（第 3.2 节、第 3.1 节）。

### 4.1.1 硬件扩展

因此，CKI 引入了一种轻量级硬件扩展，以防止 guest kernel 执行可能导致破坏性序列的特权指令。由于 PKRS 在 guest kernel 执行期间为非零（有限内存视图），而在 KSM 执行期间为零（无限内存视图），因此该扩展可以依靠 PKRS 寄存器的值来确定当前执行的是哪一个。当 PKRS 非零时，扩展会阻止所有破坏性特权指令。在客户内核中执行这些指令会触发异常，并向 host kernel 发出 trap。非破坏性特权指令仍可在客户内核中执行，以尽量减少开销。下表列出了特权指令及其在客户内核中是否被阻止（通过控制这个规则可以控制哪种指令会下陷到 host kernel）。

| Categories | Related Instructions or Registers | Blocked? | Usages in Container Guest Kernels or Brief Explanations |
|---|---|---|---|
| System Registers | IDTR, GDTR, TR ... | Yes | They are required at boot time only and replaced with KSM calls. |
| MSRs | RDMSR/WRMSR | Yes | They are used for updating timer and sending IPI, which are replaced with hypercalls. |
| Control Registers | - MOV CRn, reg<br>- MOV reg, CR0/CR4<br>- MOV reg, CR3<br>- CLAC/STAC | - No<br>- Yes<br>- Yes<br>- No | - It is used for reading CR0 and CR4, which is harmless.<br>- Replaced with KSM call: initializing CR0/CR4, toggling CR0 TS-bit for lazy FPU switching.<br>- Replaced with KSM call: updating CR3 for address space switching.<br>- They are used to toggle the AC-bit (SMAP-enabled) in CR4, which is harmless. |
| TLB States | - INVLPG<br>- INVPCID | - No<br>- Yes | INVLPG is used to flush the TLB. Each secure container and the host are isolated in different PCID contexts to prevent one container from flushing the others' TLB entries with INVLPG. |
| Syscall/ Exception | - SWAPGS, SYSRET<br>- IRET | - No<br>- Yes | - They are handled with special methods for better syscall performance.<br>- It is used for returning from exceptions and replaced with a KSM call. |
| Other Privilege Instructions | - HLT<br>- STI/CLI, POPF<br>- IN/OUT, SMSW ... | - No<br>- Yes<br>- Yes | - It is replaced with a hypercall that pause the current vCPU.<br>- The interrupt enabling/disabling state in the guest is maintained in memory.<br>- They are not used in a para-virtualized container guest kernel. |
| PKRS Register | WRPKRS | No | It is used in the PKS switch gates and protected with binary rewriting. |

**Table 3.** Deprive the container guest kernel of the ability to execute destructive privileged instructions.

> 注：block 表示如果 PKRS 非零时触发是否会阻塞并 trap 给 host kernel。

### 4.1.2 阻塞指令

> 任何写入系统寄存器、控制寄存器、`iret`、控制中断等指令（中断控制通过 host kernel 可见内存位）；

除了上表中列出的无害指令外，大多数特权指令都被阻止。被屏蔽的指令可以使用基于软件的虚拟化中的类似技术进行虚拟化，即用调用宿主内核或 KSM 来代替它们。

我们屏蔽了任何写入系统寄存器（如 `GDTR` 和 `IDTR`）、控制寄存器或特定模型寄存器（MSR）的指令。中断返回指令（`iret`）可能会修改段寄存器，因此会被阻止。我们还屏蔽了对容器 guest kernel 来说不必要的指令，如与 Port I/O 和系统管理模式相关的指令。

操作系统内核使用 `cli`/`sti` 和 `popf` 指令启用或禁用 CPU 上的中断处理。这些指令在 guest kernel 中被阻止，以防止 DoS。**CKI 采用了半虚拟化的中断处理机制**。所有硬件中断都由 host kernel 处理，host kernel 再向 guest kernel 注入虚拟中断。guest kernel 不使用特权指令管理中断启用/禁用状态（guest-aware 所以是半虚拟化），而是通过 host kernel 可见的内存位来管理。

### 4.1.3 无阻塞指令

> 修改 PKRS 的指令。x86 上使用 MSR 寄存器（特定模型寄存器），但不直接用 `wrmsr`（block），用包装后的新指令 `wrpkrs`；
>
> 并且限制 `wrpkrs` 的使用范围，二进制重写消除内核代码中的该指令，并 kernel 只读 + KSM 禁止新的可执行映射来避免安全问题（CKI 值用来提供容器环境，不需要支持那么多东西）。
>
> `sysret / swapgs` 系统调用相关也可以不阻塞。但是需要指令扩展来确保 PKRS 非零时中断不得被禁用（防止 DoS）；

> 关于 TLB flush 相关指令 `invlpg`（仅刷新当前 PCID 的 TLB 条目）是允许的，因为 secure container 和 host 隔离在不同的 PCID 上下文中，这能防止 performance attack；

对 PKRS 寄存器的修改指令应能在 guest kernel 中执行，否则 guest kernel 将无法调用 KSM。现有的 x86 硬件将 PKRS 作为特定模型寄存器 (MSR) 来实现。但是，`wrmsr` 指令应在 guest kernel 中被阻止，以防止对其他 MSR 的任意操作。我们为修改 PKRS 引入了新的硬件指令 `wrpkrs`，其语义类似于现有的 `wrpkru` 指令（修改 PKRU，相当于 PKRS 的用户空间）。

`wrpkrs` 指令只应出现在预先定义的 switch gates 上，因此我们使用先前工作中引入的类似二进制重写技术，消除了访客内核代码中的所有 `wrpkrs` 指令，包括未对齐指令。为防止客户内核动态创建 `wrpkrs` 指令，所有内核代码在 guest kernel 初始化期间都被映射为只读，KSM 禁止在容器执行期间进行新的内核可执行映射。

> **Note**
>
> CKI 不需要支持动态修补或加载 guest kernel 代码，因为这对容器来说是不必要的。请注意，CKI 的目的是提供一个容器环境，而不是支持任意的 guest kernel。

`sysret` 和 `swapgs` 指令用于处理系统调用。为这些指令调用 KSM 会将空系统调用延迟从 90ns 增加到 153ns。为了提高性能，我们允许这些指令在 guest kernel 中执行。sysret 指令可能会被用来修改 RFLAGS 寄存器并禁用内核中断（DoS）。因此，我们为该指令添加了一个轻量级扩展，以确保当 PKRS 非零时，IF（中断启用）标志保持开启。

guest kernel 可以使用 `invlpg` 清除 TLB。我们将每个 secure container 和 host 隔离在不同的 PCID 上下文中，以防止 performance attack，因为 `invlpg` 只刷新当前 PCID 的 TLB 条目。

## 4.2 KSM 中用于 Context-Switch 的 Switch Gates

下图显示了 CKI 中的上下文切换流程。CKI 为最频繁的切换（即系统调用、异常和 KSM 调用）提供了快速路径。它为其他切换（即 host kernel 调用（hypercall）和硬件中断）提供慢速路径。

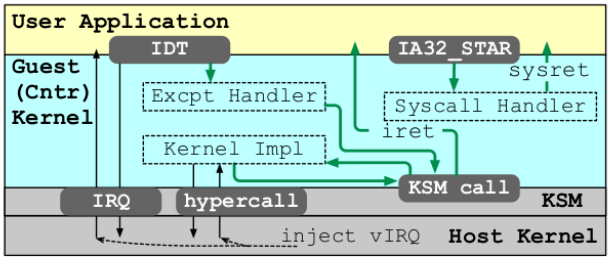



**Figure 7.** The context switches in CKI. Green lines: fast paths. (v)IRQ: (virtual) interrupt. Cntr: container.

在为这些上下文切换设计 PKS switch gates 时，KSM 中每 vCPU 区域的定位是一个难题。由于 guest kernel 可以任意修改 `kernel_gs`（Intel 中存放 per-CPU 信息地址的寄存器，有点像 AArch64 的 `TPIDR_EL1`），因此 KSM 无法依靠该寄存器来识别当前的 vCPU。

### 4.2.1 系统调用和异常

当容器中的用户应用程序调用系统调用时，它会捕获到 `IA32_STAR` 寄存器中定义的 guest kernel 入口点。同样，当应用程序触发 page fault 等异常时，它也会跳转到中断描述符表（IDT）中的 guest kernel 入口点。在用户模式下，PKRS 被设置为 `PKRS_GUEST`，允许入口点调用不受信任的处理程序函数，而无需切换 PKS。系统调用和异常的进入和退出代码使用三条特权指令：`swapgs`、`sysret` 和 `iret`。`swapgs` 和 `sysret` 指令可在 guest kernel 中执行，而 `iret` 指令必须通过调用 KSM 才能执行（参见 4.1）。

### 4.2.2 KSM 调用

guest kernel 使用 **KSM call gate** 调用 KSM 提供的特权操作（下图）。该门将 PKRS 设为 0，切换到 guest kernel 无法访问的安全堆栈，调用处理函数，最后恢复 PKRS 和堆栈指针。

```
switch_pks \pkrs:
xor %rcx, %rcx
xor %rdx, %rdx

mov \pkrs, %rax

// new instruction
wrpkrs

// avoid gate abuse
cmp \pkrs, %rax
jne abort
```

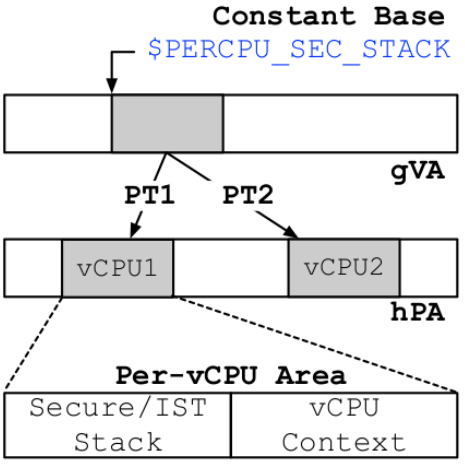
```
KSM_call:
switch_pks $0
mov %rsp, %rcx
// gs is untrusted
mov $PERCPU_SEC_STACK, %rsp
push %rcx

… // KSM handles the request

pop %rcx
mov %rcx, %rsp
switch_pks $PKRS_GUEST
ret
```

攻击者可能会利用类似 ROP 的攻击跳转到门结束时的 `wrpkrs` 指令，任意修改 PKRS 并执行恶意代码。为防止这种攻击，如上图中的 `switch_pks` 宏所示，修改后会检查新的 PKRS 值。

> **Important**
>
> 既然 `kernel_gs` 不可信，如何识别 per-vCPU 区域?
>
> 由于 KSM 可在多个 vCPU 上同时调用，因此每个 vCPU 都有自己的安全堆栈，位于 KSM 内存的每个 vCPU 区域。操作系统内核通常使用 `kernel_gs` 寄存器来定位每个 CPU 的变量，即每个 CPU 上的 `kernel_gs` 寄存器为本地 CPU 变量存储不同的基数。然而，CKI 允许 guest 执行 `swapgs` 指令（参见 4.1），因此恶意客户内核可以任意修改 `kernel_gs`。为了解决这个问题，**CKI 将每 vCPU 区域放在一个恒定的虚拟地址上**，这样就可以在没有 `kernel_gs` 的情况下找到它。
>
> 如下图所示，CKI 为 guest kernel 中的每个页表维护多个 per-vCPU 页表。当 guest thread 在不同的 vCPU 上执行时，会使用不同的 per-vCPU 页表。每个 per-vCPU 页表在相同的恒定的虚拟地址 (GVA) 上映射不同的 per-vCPU 区域 (HPA)。



### 4.2.3 Hypercall

Guest kernel 使用 hypercall gate 调用 host kernel 提供的特权操作。该门首先将 PKRS 切换为零，因为它需要执行特权指令并访问 KSM 内存（每虚拟 CPU 区域）。然后，它执行一次完整的上下文切换，以保存 guest kernel 上下文并恢复 host kernel 上下文，其中包括页表切换、通用/系统寄存器切换和侧信道缓解（如 IBRS）。host 和 guest 上下文存储在每 vCPU 区域。然后，host kernel 从 guest 上下文读取请求并进行处理。请求完成后，host kernel 恢复 guest 上下文，guest kernel 从 hypercall 门恢复。

### 4.2.4 Hardware Interrupt

硬件中断会触发从 guest 到 host kernel 的 trap。硬件中断的 IDT 条目指向一个 interrupt gate；

中断门将中断信息保存到每 vCPU 区域，然后切换到 host kernel。host kernel 读取信息，构建中断上下文，并调用中断处理程序。处理完中断后，host kernel 会恢复被中断的 guest 上下文。

我们添加了一个**硬件扩展，用于在中断进入时保存 PKRS 寄存器，并将 PKRS 切换为零**（参见 4.4）。处理中断后，`iret` 指令应在 PKRS 设置为零时执行（参见 4.1），但在恢复 guest kernel 上下文时，它需要将 PKRS 重新设置成 `PKRS_GUEST`。因此，我们**扩展了 `iret` 指令，允许它修改 PKRS 寄存器**。

```
hypercall:
// request in registers
switch_pks $0
exit_to_host
switch_pks $PKRS_GUEST
ret
```

```
idtentry \irqno:
// HW: switch to IST stack
// HW: save PKRS, switch it to 0
save IRQ info (\irqno, errcode)
exit_to_host
iret // HW: restore PKRS
```

```
exit_to_host:
save guest registers
restore host registers
save guest CR3
switch to host CR3

… // trap into host kernel

switch to guest CR3
save host registers
restore guest registers
```

## 4.3 内存保护机制

恶意 guest kernel 可能会试图通过操纵页表来破坏内存隔离。为了隔离客户机的内存视图，KSM 会拦截并验证 guest kernel 中的所有页表更新（回忆页表更新是不涉及 L0 kernel 并且仅与 KSM private data 有关）。

### 4.3.1 页表监控

为了拦截页表更新，CKI 采用了与嵌套内核类似的机制，该机制基于以下不变式（invariant）：

1. 只有已声明的页才能用作页表页（PTP）；
2. 已声明的 PTP 在 guest kernel 中是只读的（**KSM / host kernel 才能修改**）；
3. 只有已声明的顶级 PTP 才能加载到 CR3 寄存器中。

与嵌套内核不同，CKI 使用 PKS 而不是 PTE writable bit 来控制 PTP 的写入权限。CKI 将客户虚拟地址空间（GVA）中的所有 PTP 都划分到一个特定的 PKS 域中。它将 PKS 域 ID（`pkey_PTP`）添加到映射 PTP 的每个客户 PTE 中。执行 guest kernel 时，该 PKS 域在 PKRS 寄存器中被配置为只读。

> 简言之：**guest 页表所在页使用专用 PKS domain，在 guest kernel 执行时只读**；

**KSM 会为属于 guest 的每个物理页面维护一个描述符**。guest kernel 可调用 KSM 声明 PTP 或更新 PTE。

- 声明 PTP 时，会指定 PTP 级别并记录在描述符中。然后，KSM 会在页表中查找映射该 PTP 的 PTE，并将 `pkey_PTP`（PKS domain ID）添加到该 PTE 中。
- **KSM 还会检查描述符中的引用计数器，以确保 PTP 只被映射一次**。
- 更新 PTE 时，KSM 会验证新 PTE 是否指向有效的下一级 PTP 或属于客户的数据页，而不会映射已声明的 PTP。
- 此外，为防止恶意 `wrpkrs` 指令（参见 4.1），如果新映射是内核可执行的，KSM 将禁止更新。

### 4.3.2 Per-vCPU 的页表

如第 4.2 节所述，CKI 为每个 guest 页表维护多个每 vCPU 页表。每个 per-vCPU 页表映射 KSM 内存中不同的 per-vCPU 区域。具体来说，**KSM 会为客户机中的每个顶级 PTP 维护多个 per-vCPU 副本**。

- 声明顶级 PTP 时，KSM 会将自己的代码和数据映射（包括 per-vCPU 区域）添加到每个副本中（有点像 kernel 给用户态程序配置页表的情形）。
- 当 guest kernel 调用 KSM 更新 CR3 时，KSM 会验证新的 CR3 值是否指向已声明的顶级 PTP，然后将相应的 PTP 副本加载到 CR3 中。
- 此外，KSM 还为读取顶层 PTP 中的 PTE 提供了一个接口，访问/脏位会从副本传播到原始 PTP。

### 4.3.3 与 Shadow Paging 的比较

与影子分页相比，CKI 的性能优势来自于更轻量级的 page fault 和 PTE 更新流程：

1. 轻量级页面故障。由于没有两阶段地址转换，CKI 中的用户页面故障可由客户内核直接处理。相比之下，在影子分页下，用户页面故障会被主机内核拦截，主机内核会执行页表走行以确定页面故障的类型（第一阶段或第二阶段），然后将页面故障注入客户内核。
2. 轻量级 PTE 更新。
    - 首先，在影子分页下，guest 中的 PTE 更新会触发虚拟机退出到 host kernel。相比之下，CKI 中的 guest kernel 可通过轻量级 PKS gate 执行 KSM call 从而进行 PTE 更新；
    - 其次，影子分页将 GPA 与 QEMU 进程的虚拟内存区（VMA）关联起来。向 PTE 写入 GPA 时，必须从 VMA 的映射中找到与 GPA 相关的 HPA，这非常耗时。相反，CKI 将 HPA 委托给 guest kernel，允许 guest kernel 直接填充 PTE 中的 HPA，而不是 GPA。

CKI 的一个局限是，**它为每个 secure container 分配连续的物理内存段，这可能会由于内存碎片化而导致内存利用率较低**。之所以做出这样的设计选择，是因为细粒度的离散内存分配会带来两个问题：

1. 它要求 KSM 在验证 PTE 更新时搜索大量元数据（页面粒度而不是段粒度），从而导致性能下降；
2. 其次，容器内核（Linux）采用 buddy system 来管理物理内存，该系统可通过连续的内存段高效运行；

因此，我们优先考虑运行时间性能，而不是内存利用效率。

## 4.4 防止中断滥用

被入侵的 guest kernel 有三种通过滥用中断发起 DoS 攻击的潜在方法。

1. 中断垄断。它可能会修改中断门的代码，以独占所有中断。这样，主机和其他容器就无法再接收中断；
2. 中断栈破坏。它可能会操纵中断堆栈，引发无法恢复的故障。具体来说，当中断发生时，CPU 会将上下文数据推入中断栈。如果中断发生在内核模式下，CPU 默认使用中断发生时的堆栈作为中断堆栈。恶意访客内核可能会将堆栈指针设置为无效地址，导致 CPU 尝试推送数据时出现三重故障；

3. 中断伪造。它可以伪造中断，用不必要的中断请求压垮系统，从而降低系统性能或导致 host kernel 出现未定义的行为。

CKI 可以防御所有这些攻击。

### 4.4.1 防止中断垄断

CKI 有以下策略防止中断垄断：

1. 门不可修改（位于 KSM）。CKI 在 KSM 内存中分配 IDT 和 interrupt gate 代码，使 guest kernel 无法修改它们；
2. **Guest Kernel 权限剥夺**。它使用 PKS gate 权限剥夺机制（参见 4.1 节）确保 guest kernel 无法禁用中断处理或修改 IDTR（IDT 基地址寄存器）；
3. 门代码映射不可修改（位于 KSM）。guest kernel 不能更改或删除 IDT 或 interrupt gate 代码的映射，因为 KSM 在每个激活的页表中映射了自己的内存（参见 4.3）；

有了这些机制，当中断发生时，CPU 控制流总能切换到正确的中断门。

### 4.4.2 防止中断堆栈破坏

CKI 利用 x86 中断堆栈表 (IST) 功能，确保 CPU 始终使用正确的中断堆栈。具体来说，IST 允许设置特定的中断堆栈，并强制 CPU 在推送中断上下文之前切换到该堆栈。IST 初始化由 KSM 完成（guest kernel 无法执行相关特权指令），相应内存也位于 KSM 中（guest kernel 无法修改相应内存数据）。

简而言之，**x86 IST + KSM 来初始化和内存存放**。

### 4.4.3 防止中断伪造

由于 interrupt gate 需要访问 KSM 内存并执行特权指令，因此当 guest kernel 发生中断时，它需要首先将 PKRS 切换为零。如果在门内通过 `wrpkrs` 指令进行切换，那么恶意 guest kernel 就会直接跳转到其中一个中断门，并向 host kernel 发送伪造的中断。

因此为了防止伪造中断，CKI 扩展了 IDT 配置，除了切换中断堆栈等原有功能外，还进一步支持切换 PKRS 寄存器。如下图中蓝色下划线文本所述，当发生硬件中断时，这一微小的硬件扩展会自动将 PKRS 寄存器置零。

<table>
<tr>
<td>

```
hypercall:
// request in registers
switch_pks $0
exit_to_host
switch_pks $PKRS_GUEST
ret
```

</td>
<td rowspan="2">

```
exit_to_host:
save guest registers
restore host registers
save guest CR3
switch to host CR3

… // trap into host kernel

switch to guest CR3
save host registers
restore guest registers
```

</td>
</tr>
<tr>
<td>

```
idtentry \irqno:
// HW: switch to IST stack
// HW: save PKRS, switch it to 0
save IRQ info (\irqno, errcode)
exit_to_host
iret // HW: restore PKRS
```

</td>
</tr>
</table>

所以，interrupt gate 中没有 `wrpkrs` 指令。如果 guest kernel 跳转到门入口，PKRS 将保持 `PKRS_GUEST`，导致后续上下文切换失败。请注意，应用程序或 guest kernel 可能会使用 int 指令生成软件中断。**硬件扩展只在硬件中断时切换 PKRS，而在软件中断时保持 PKRS 不变**。

此外，guest kernel 也无法滥用 hypercall gate 进行中断伪造，因为 host kernel 可以根据 KSM（每 vCPU 内存区）中保存的信息识别不同的退出原因。

# 5. 实现

## 5.1 Guest Kernel

我们在 CKI 容器中将 Linux 内核作为访客内核运行。我们利用 Linux 内核中的半虚拟化实用程序（即 `pv_ops`）来 hook 特权操作。我们还在 Linux 内核中添加了一个新的启动程序，以移除传统的初始化操作。我们增加了 2000 行代码，修改了不到 80 行代码。

移除两阶段地址转换不需要大量的移植工作。

- 传统内核可能依赖固定的低物理地址来启动真实模式。相反，CKI 通过半虚拟化直接从 long mode 启动虚拟 CPU（vCPU）；
- 传统的虚拟化堆栈使用两阶段地址转换来创建 MMIO 区域，这些区域在第一阶段映射，但在第二阶段不映射。我们用 hypercall 取代了客户内核（VirtIO 前端）中的 MMIO；

在兼容性方面，CKI 有可能支持与基于软件的虚拟化相同的客户内核功能。

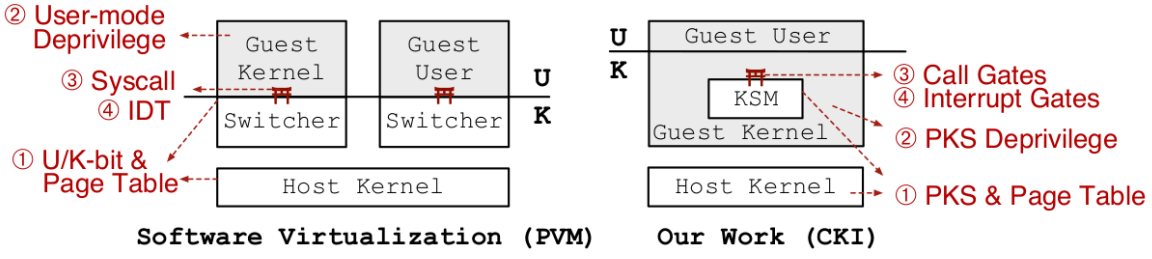
## 5.2 Hardware Extensions

第 7 节中的性能评估是在真实硬件而非模拟器上进行的。在评估中，我们使用 `wrpkru` 指令来模拟 `wrpkrs` 指令。根据我们基于 Gem5 模拟器的评估，在特权指令中添加 PKS 权限检查逻辑产生的开销可以忽略不计，因此我们在评估中直接使用未修改的指令。我们通过添加 wrpkru 指令来模拟中断进入和 `iret` 期间的 PKRS switching 开销。

# 6. 安全性分析

CKI 可以实现与基于软件的虚拟化（PVM）相同的安全目标，因为它实现了相同的隔离基元（见下图）。

1. 页表和内存隔离（PKS）：PVM 通过 PTE U/K 位和单独的页表将 switcher 和 host kernel 内存与 guest kernel 隔离，而 CKI 通过 PKS 和独立页表将 KSM 和 host kernel 内存与 guest kernel 隔离（参见 4.3 节）；
2. 特权指令隔离（PKS EXT 权限剥夺）：PVM 通过在用户模式下运行，防止虚拟机 guest kernel 执行特权指令，而 CKI 通过 PKS 限制 guest kernel 执行特权指令（参见 4.1）；
3. 特权转换防护（Call Gates，包括 KSM call，Hypercall 等）：PVM 为虚拟机提供了一个预定义的系统调用入口点来调用主机内核。CKI 采用二进制重写技术，消除了客户内核中的 `wrpkrs` 指令，只留下有效的入口点来调用 KSM 或主机内核（参见 4.2）；
4. 中断防护（Interrupt Gates）：在 PVM 中，当虚拟机被硬件中断中断时，CPU 会调用 IDT 中定义的相应主内核处理函数。CKI 设计了中断门，将硬件中断重定向到主机内核，确保中断门不会被破坏或滥用（参见 4.4）。



# 7. 总结

CKI 是一种用于构建高效安全容器的软硬件协同设计。

通过利用和扩展用于内核隔离的轻量级 CPU 功能，CKI 为容器内核有效地构建了一个新的权限级别，从而优化了性能开销和安全隔离，并超越了 SOTA 的安全容器设计。虽然我们的 PKS 扩展并不适用于当前的 CPU，但我们相信它们将为未来的使用案例带来新的机遇，包括以下方面： 对不可信的内核驱动程序进行沙盒处理： 直接将驱动程序隔离在 ring-0 内，无需像微内核设计那样将其权限下放到 ring-3，从而避免了用户-内核或进程间通信的额外性能开销。

内核级系统调用优化： 在内核中运行系统调用密集型应用程序，通过消除传统的系统调用开销来获得更好的性能。我们计划在今后的工作中探索这些方向。