软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(2):473−497 [doi: 10.13328/j.cnki.jos.006211]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于硬件虚拟化的内核同层多域隔离模型*

钟炳南[1,2], 邓 良[3], 曾庆凯[1,2]

[1](计算机软件新技术国家重点实验室(南京大学), 江苏 南京 210023)

[2](南京大学 计算机科学与技术系, 江苏 南京 210023)

[3](华为技术有限公司, 上海 201206)

通信作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 为了解决内核不可信带来的问题, 很多工作提出了同层可信基的架构, 即, 在内核同一硬件特权水平构建可部署安全机制的唯一保护域. 但是, 实际过程中往往面临多样化的安全需求, 将多种对应的安全机制集中于唯一的保护域必然导致只要其中任何一个安全机制被攻陷, 同一个保护域内其他所有安全机制都可能被攻击者恶意篡改或者破坏. 为了解决上述问题, 提出了内核同层多域隔离模型, 即在内核同一硬件特权水平构建多个保护域实现了不同安全机制的内部隔离, 缓解了传统方法将所有安全机制绑定在唯一保护域带来的安全风险. 实现了内核同层多域隔离模型的原型系统 Decentralized-KPD, 其利用硬件虚拟化技术和地址重映射技术, 将不同安全机制部署在与内核同一特权水平的多个保护域中, 并不会引起较大的性能开销. 总体而言, 实验结果展示了内核同层多域隔离模型的安全性和实用性.

关键词: 硬件虚拟化; 内存隔离; 多域隔离

中图法分类号: TP306

中文引用格式: 钟炳南, 邓良, 曾庆凯. 基于硬件虚拟化的内核同层多域隔离模型. 软件学报, 2022, 33(2): 473–497. http://www.jos.org.cn/1000-9825/6211.htm

英文引用格式: Zhong BN, Deng L, Zeng QK. Kernel-level Multi-domain Isolation Model Based on Hardware Virtualization. Ruan Jian Xue Bao/Journal of Software, 2022, 33(2): 473–497 (in Chinese). http://www.jos.org.cn/1000-9825/6211.htm

Kernel-level Multi-domain Isolation Model Based on Hardware Virtualization

ZHONG Bing-Nan[1,2], DENG Liang[3], ZENG Qing-Kai[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[3](Huawei Technology Co., Ltd., Shanghai 201206, China)

Abstract: In order to solve the problem caused by untrusted kernel, the trusted base architecture at the same privilege of the kernel has been proposed by a lot of works. It provides the only one protection domain to deploy security mechanism at the same hardware privilege level of the kernel. However, in practice, it is often faced with diversified security requirements. Moreover, it is high risk to make multiple corresponding security mechanisms concentrated into a single protection domain. All other security mechanisms in the same protection domain may be maliciously tampered or destructed, as long as any one of the security mechanisms is compromised by the attacker. To address this problem, a kernel-level multi-domain isolation model isproposed in this study, which constructs multiple protection domains at the same hardware privilege level with the kernel to achieve internal isolation of different security mechanisms, and it will alleviate the security risks of traditional method which bind all security mechanisms into a single protection domain. This study has implemented the decentralized-KPD prototype system of the kernel-level multi-domain isolation model, which uses hardware virtualization technology and address remapping technology to deploy different security mechanisms in multiple protection domains at the kernel privilege level and it

* 基金项目: 国家自然科学基金(61772266, 61431008)

收稿时间: 2020-05-14; 修改时间: 2020-07-06, 2020-08-10, 2020-09-13; 采用时间: 2020-11-23; jos 在线出版时间: 2021-01-15

will not cause a large performance overhead. Overall, the experimental results demonstrate the security and utility of the kernel-level multi-domain isolation model.

Key words: hardware virtualization; memory isolation; multi-domain isolation

现有的内核保护方案[1−28]都是将安全机制绑定在系统中唯一的保护域中, 并且采用多种技术强化这种内存隔离. 传统的基于虚拟化的方法[2−11]引入了一个更高的特权层, 并利用底层支持虚拟化的 VMM(virtual machine monitor)强化透明的内存隔离. 但与此同时, 频繁的特权层切换会带来较大的性能开销. 为了提高内核监控的效率, SIM[9]利用影子页表和虚拟化技术在内核同一硬件特权水平创建了系统中唯一的保护域, 并利用 MOV_TO_CR3 指令实现内核与保护域之间的切换; 同时, SIM 引入了 CR3-Target 机制, 避免执行 MOV_TO_CR3 指令时引起 VM Exit, 从而避免了频繁且昂贵的 non-root/root 模式之间的切换. ShadowMonitor[10]中, 用 EPT(extended page table)技术在 Guest VM 层中创建一个与内核同层的保护域, 部署 VMI(virtual machine introspection)机制实现对内核的有效监控, 避免了传统 VMI 机制带来的频繁的特权层切换与语义鸿沟问题. Nested Kernel[12]利用 CR0 寄存器中的 WP 位实现了对页表只读的控制, 并且通过二进制重写的方法消除了内核代码中 MOV_TO_CR0 的指令, 从而构建了内核同层的保护域. Virtual Ghost[13]则在 SVA(secure virtual architecture)架构的基础上构建了内核同层的保护域, 与 Nested Kernel 一样, 也无需虚拟化功能的支持. 最近的一些工作利用页保护[14]或者 SFI(software fault isolation)[15]将现有的操作系统划分到多个保护域中, 但同时也会导致较大的性能开销. EqualVisor[16]与 Nested Kernel 一样, 利用 CR0 寄存器的 WP 位构建了页表锁定技术, 但却是在 VMM(virtual machine monitor, 或 hypervisor)同层创建的唯一的保护域, 实现了对 Hypervisor 的有效内存保护. Dancing[17]利用随机化技术在 Hypervisor 层创建了唯一保护域, 实现了对 VMM 的主动监控, 其方法同样可以应用到内核的监控中.

现有的单保护域方案的主要工作是构建一个与内核隔离的可信执行环境(保护域), 并在此保护域内部署相应的安全机制, 而且一般会假设部署在此保护域内安全机制的代码和数据都是可信的. 保护域内主要包括构建保护域的代码和数据以及保护域内安全机制的代码和数据. 但是, 现实中的安全需求是多种多样的, 比如内核的关键数据保护, 而且内核的关键数据有很多, 包括页表和 task_struct 等数据结构, 现实中往往不可能只考虑其中的一种. 页表是内核中的关键数据, 攻击者可较容易地利用内核中的内存破坏漏洞任意地破坏或者篡改内核页表数据. SecPod[24]利用虚拟化技术保护在内核同层建立了一个保护域, 并利用分页委托代理(paging delegation delegates)和审计内核页表操作(audits the kernel's paging operations)技术实现了在保护域内完成内核页表的截取和验证, 防止其被攻击者恶意破坏或者篡改. 最近, Microsoft 发布了 Windows 10 补丁[29], 该补丁程序随机化了用于计算页表项虚拟地址的基地址, 但是不能抵御关键信息泄露攻击. PT_Rand[30]利用随机化技术保护内核页表, 同时利用 DR3 寄存器保护随机化过程中产生的基值以抵御信息泄露攻击. 而且在 PT_Rand 的实现过程中, 其联合了 PT_Rand 和内核 CFI(control flow integrity)保护(RAP[31])共同抵御对页表的仅数据攻击(data-only attack). PrivWatcher[32]利用保护域保护内核关键数据结构 cred 抵御内存破坏攻击, 同时假设内核页表是受到保护的. 这说明, 在现实中, 我们仍然需要联合多种安全机制共同提供一种安全服务或者分别提供多种安全服务.

在现有的单保护域方案中, 同时满足多样化的安全服务必然导致多种安全机制集中在系统中唯一的保护域中, 而将所有的安全功能绑定在单保护域将会面临如下问题.

1) 在单保护域内部署多种多样的安全机制, 意味着所有的代码和数据都集中在一个地址空间内, 默认共享了保护域中所有的特权. 如果保护域内任何一个安全机制的代码存在漏洞, 都可能使得攻击者轻易地利用此漏洞而攻陷此保护域. 之后, 攻击者便可以轻易地获取系统全部的超级权限[12], 继而对整个保护域的地址空间进行任意的读写, 甚至破坏保护域的可信执行环境;
2) 虽然针对特定的内核安全问题部署相应的安全策略可以满足 TCB(trusted computing base)规模最小化的要求. 但现实中内核需要的安全功能往往都很多样化, 因此在满足多样化的安全功能需求时, 单一的保护域必然导致系统整体 TCB 规模的膨胀. 如果将多种安全机制绑定在唯一的保护域内, 那

么在不考虑每个安全机制存在共享代码的情况下, 系统整体的 TCB 规模则是每个安全机制的 TCB 规模直接相加, 即, 系统整体 TCB 规模与系统内部署安全机制的数量是呈正相关的.

为了缓解现有上述问题, 将不同的安全机制部署在不同的保护域内是一个值得研究的问题. TEEv[33]提出了多可信执行环境概念(trusted execution environment, TEE, 也称为保护域), 在 ARM 平台上利用 TrustZone 硬件特征将单一用途的可信执行环境扩展为多个可信执行环境, 将来源众多的可信应用程序(trusted application, TA)分散部署在不同可信执行环境中. 但其解决的是应用层面的问题, 而且针对的也是 ARM 平台.

本文提出了基于硬件虚拟化的内核多域隔离模型 Decentralized-KPD, 即: 在 x86 平台上, 利用虚拟化硬件特征(VMFUNC)和地址重映射技术, 在内核同一硬件特权水平构建多个保护域, 将基于事件驱动的多个安全机制部署在不同的保护域中, 保证每个安全机制都可以独立地运行. Decentralized-KPD 采用了多种技术, 强化了多个保护域之间的内存隔离, 并确保内核和各个保护域的执行环境的完整: (1) 利用 Intel 提供的虚拟化技术实现内核与多个保护域之间的内存隔离以及多个保护域之间的内存隔离; (2) 必须保证 Decentralized-KPD 中保护域执行环境的完整性, 保证其中一个被攻陷不会影响到其他保护域和地址翻译完整性; (3) 在 Decentralized-KPD 中, 利用 EPTP-Switching 技术和地址重映射技术实现内核域与保护域之间的快速切换, 可以同步切换 EPT (extended page table)、IDT(interrupt description table)以及 Guest 页表(Guest page table).

实验结果表明, 其性能开销与传统的单一保护域相比不会引起太大的性能开销. 总之, 本文的关键贡献如下:

1) 本文提出了基于硬件虚拟化的内核同层多域隔离模型, 其在内核同一硬件特权层构建了多个基于内存隔离的保护域, 可以在不同的保护域中部署不同的具有针对性的安全策略;
2) Decentralized-KPD 利用硬件虚拟化特征 EPT 和地址重映射技术实现了对保护域的内存隔离和地址访问的完整性, 保证了每个保护域执行环境的完整性; 而且, Decentralized-KPD 利用 VMFUNC 一条指令实现了内核和保护域之间快速而有效的切换;
3) 本文在 Linux 平台上实现了 Decentralized-KPD, 并进行了一系列的实验, 证明了 Decentralized-KPD 在内核保护时的有效性和可行性. 与基于单保护域的方法相比, 并不会引起较大的性能开销. 同时, 通过实验分析了 Decentralized-KPD 的安全性和实用性.

1 技术背景

为了易于理解 Decentralized-KPD 的设计和实现原理, 本文首先简单介绍 Decentralized-KPD 中应用到的关键技术. 在 x86 中, 硬件虚拟化技术(Intel VMX[18])提供了两种 CPU 的运行模式: root 模式和 non-root 模式. 运行在 root 模式的程序(一般称其为 hypervisor)可以通过 VMCS(virtual machine control structure)数据结构控制 non-root 模式中的运行行为.

在硬件辅助内存虚拟化技术中, 为了实现地址的完整翻译, 其一共设有两级页表: 第 1 级是在 Guest 空间的内核负责管理的 GPT(Guest page table), 其实现 GVA(guest virtual address)到 GPA(guest physical address)的映射; 而第 2 级页表则是 Host 空间的 Hypervisor 负责管理的 EPT(extended page table), 其实现 GPA(guest physical address)到 MPA(machine physical address)的映射. 在内存访问过程中, 硬件 MMU 将会检查两级页表中的访问权限, 且在 EPT 页表中, 实现了对页面读写执行权限更明确的控制. 传统的 VMM 通常会为每个 Guest 分配一个 EPT, 而 Intel CPU 允许最多可为 1 个 Guest 分配 512 个 EPT. 当 Guest OS 访问未分配的资源, 如内存页时, 会引起 EPT 层的缺页异常, 并导致系统从 non-root 模式进入到 root 模式中. VMM 从 EPT_VIOLATION 类 VM Exit 退出信息域中获取具体的缺页信息, 由 VMM 中 EPT 的缺页处理程序完成实际物理页的分配和 EPT 页表操作, 并最终返回到引起 EPT_VIOLATION 的地方继续执行. 同样, 当内存访问违反 EPT 页表项中设置的读写执行权限时, 一样会引起 EPT_VIOLATION 而导致 VM Exit.

在 x86 中, 硬件虚拟化提供了 EPTP-switching[18]功能, 即允许在 non-root 模式中通过 VMFUNC 指令直接实现 EPT 页表的切换, 而不会引起 VM Exit. 通过在 VMCS 中相关的域进行设置, 开启硬件虚拟化支持的

EPTP-switching 功能. 执行 VMFUNC 指令之前需要设置 RAX 与 RCX 寄存器, 其中, RAX 值置零时指定 EPTP Switching 功能, 而 RCX 指定 EPTP list 中的项(VMFUNC 在 ECX≥512 时会产生 VM Exit). VMCS 中的 EPTP 域指向一个 4 KB 大小的页, 包含 EPT 的地址(每一个页中包含 512 个 8-byte 的项).

- CR3-Target Control 机制[18].

通过 VMCS 的设置, 可以禁止在 non-root 模式运行 MOV-TO_CR3 指令, 任何对 CR3 寄存器的修改, 都会引起 VM Exit 而陷入到 root 模式. 而 CR3-Target Controls 机制则可以绕过此限制, 当向 CR3 寄存器写入的特定目标值存储在 VMCS 中 CR3_TARGET_LIST 域内时, 执行 MOV_TO_CR3 指令修改 CR3 寄存器则不会陷入到 root 模式. VMCS 的 CR3_TARGET_LIST 中最多支持 4 个 CR3 寄存器目标值.

2 系统框架模型

2.1 Decentralized-KPD框架模型

Decentralized-KPD 架构如图 1 所示, 其核心是构建多个与内核处于同一硬件水平的保护域.

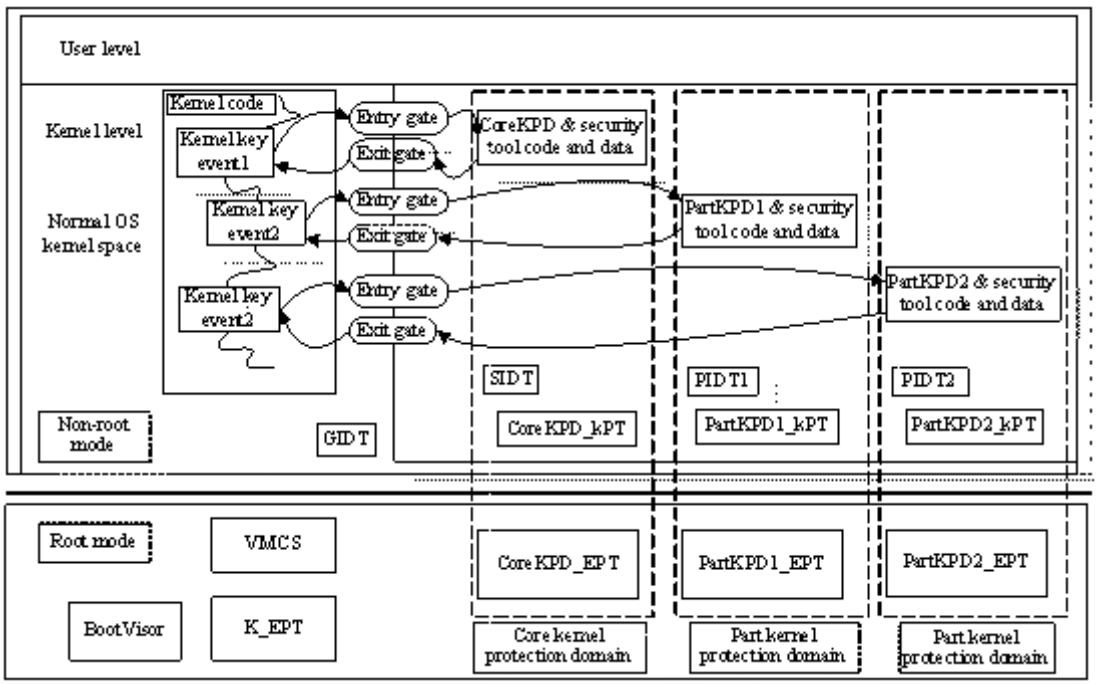


图 1 Decentralized-KPD 内核保护整体框架

在 Decentralized-KPD 中, 多个保护域由一个基本保护域(CoreKPD)和多个功能保护域(PartKPD*n*, *n* 是功能保护域的索引)构成. 保护域内的安全机制提供安全服务, 实现对内核事件的截取和验证, 并不替代内核的服务. 基本保护域的主要目的是建立多域的控制流保护功能以及 Guest OS 页表的保护, 并假设其是可信的. 页表是内核最为关键的数据之一, 其直接决定了内核正常的内存访问, 因此, 保护页表是提供内核层保护服务的基础. 而且在基本保护域内, 负责页表操作和中断的截取与验证, 代码量本身较小, 可以通过形式化的方法验证其安全性. 另外, 在系统启动过程中, 首先初始化完成的是基本保护域, 之后才是其他功能保护域. 而且底层 BootVisor 可以通过 EPT 页表的项设置保护基本保护域中的代码和数据, 防止攻击者利用内核漏洞恶意破坏基本保护域. 功能保护域主要是为内核提供一般的安全服务, 其中, 安全机制的代码可能是存在漏洞的. 本文为了验证对多域的支持, 设计了两个功能保护域, 并且在每个功能保护域内部署不同的安全机制.

相关组件.

- 保护域(protection domain): 保护域是一个可信执行环境. 保护域与 Guest OS 共享同一个虚拟地址空间, 保护域内包含必要的代码和数据用于保护域的构建, 同样也包括对 Guest OS 提供安全服务的代码和数据. 根据不同的需求, 可以在不同的保护域内部署相应的安全机制. 在 Decentralized-KPD 中, 将多个保护域分为一个基本保护域和若干个功能保护域. 在基本保护域中, 其还包含 Guest 层内核页表和中断处理的程序. 而功能保护域(KPD*n*)内安全工具的代码和数据(security tool code and data, ST code and data)是针对不可信内核的多个安全机制的代码和数据;
- Gateway(entry/exit gate): Gateway 是内核与保护域之间进行控制切换的唯一合法接口. 保护域内的安全机制通过在 Guest OS 中关键事件处设置钩子(hook)截取控制流, 通过 Gateway 完成控制流由内核到保护域的转换, 从而实现关键事件的截获. 在 Decentralized-KPD 中设置了多个保护域, 为了避免内核与保护域之间错误的控制流转换, 每个 Gateway 都对应唯一的保护域;
- 被隔离的系统(Guest OS): Guest OS 在系统完成初始化之后, 与所有保护域一样, 都运行在 non-root 模式中, 是不可信的操作系统, 而且需要隔离和保护的对象;
- 虚拟机监控器(BootVisor): BootVisor 运行在 root 模式, 主要任务是完成系统的初始化. 在 Guest OS 运行时, BootVisor 允许其直接访问硬件资源, 但是不直接干预 Guest OS 的正常运行.

2.2 威胁模型和假设

本文假设多保护域其中一个保护域内的安全机制的代码存在漏洞, 攻击者可能利用这些漏洞而攻陷该保护域, 从而获得该保护域拥有的所有权限. 多保护域中的每个保护域不仅要面对不可信内核的威胁, 同时也要面对可能被攻陷的保护域的威胁. 本文利用 Intel TXT[26]技术实现可信启动, 同时在系统初始化的过程中配置完善内核和所有保护域的隔离执行环境, 此时, 操作系统和应用程序均未启动, 不存在攻击面. 保护域带来的威胁与其具体安全服务相关, 不像内核那样具有所有软/硬件资源的管理权限, 但同样需要限制其安全风险的扩散. 对于攻击者而言, 其一旦攻陷内核而获得内核权限, 便可以轻易地使内核宕机从而拒绝提供任何服务, 因此, 本文不考虑拒绝服务攻击. 本文假设硬件和固件都是可信的, 而且也不考虑物理层面的攻击.

2.3 挑 战

Decentralized-KPD 在设计与实现时面临如下挑战.

1) 没有支持多域隔离的硬件支持. X86 芯片提供的 Late Launch[28]机制和 SMM 机制虽然可以创建一块隔离内存区域, 但并不能同时支持多个内存区域的隔离. 在 x86 芯片架构中, 通过分页模式可以在 ring3 环用户层创建基于虚拟地址隔离的多域隔离, 但其页表的管理都是由 ring0 环的内核负责的, 而且在内核中内核虚拟地址空间是共享的. 因此, 需要一种方法实现 Decentralized-KPD 中内核同层的多个内存区域的隔离, 并且在每个保护域上附加严格的限制;
2) Decentralized-KPD 在内核同一硬件特权水平构建多个保护域, 不仅需要确保多个保护域与内核的安全隔离, 也要确保保护域之间的安全隔离, 同时还需要保证每个保护域的独立可靠执行, 保证不会影响到彼此之间的控制流;
3) 快速、有效的切换. Decentralized-KPD 为每个保护域设置独立的执行环境, 包含了必要的组件. 当内核与保护域切换时, 必须保证切换时所有相关组件的同步以减小攻击面, 同时也有利于提高性能.

2.4 基本思路

本文的核心思想是: 在内核同一硬件特权层创建多个具有独立可信执行环境的保护域, 包括基本保护域以及功能保护域, 并且将内核中需要的多个安全机制分散部署在不同的保护域中. 多域隔离模型的关键在于一方面限制不可信内核攻击其他保护域, 另一方面限制功能保护域攻击其他的保护域, 包括通过攻击内核而使得其他保护域提供的安全机制被破坏.

- 首先, 在系统的初始化阶段, 利用硬件虚拟化技术创建多个隔离的内存区域, 完整配置多域的隔离执

行环境, 不仅内核与保护域之间相互隔离, 保护域之间也是相互隔离的. 初始化过程中, 将每个安全机制部署在不同的保护域中. 系统初始化完成之后, 整个系统运行在 non-root 模式中. 并且, 基于硬件虚拟化的设置, 整个系统只能在 non-root 模式下运行;

- 其次, 在系统运行阶段, 本文建立了“硬件自锁”机制, 即确保建立多域隔离机制的代码和数据只能在 root 模式下进行访问, 并且确保 non-root 模式下的关键组件是不可恶意篡改的, 而系统在正常运行过程中并不会陷入到 root 模式中, 从而获得较强的内存隔离. Decentralized-KPD 中任何引起 VM Eixt 导致系统陷入 root 模式中的行为都被认为是恶意的, 并会导致整个系统的重启.

基于上述两点, 本文建立了基于硬件虚拟化的内核同层多域隔离模型, 可将实际过程中针对内核的多样化安全机制隔离在不同的功能保护域中, 从而有效分散了安全机制集中于一个保护域所面临的安全风险.

3 系统设计

本节将详细介绍内核同层多域隔离模型 Decentralized-KPD 的设计, 并阐述它是如何解决上述所面临的挑战的.

3.1 多域内存隔离与保护

Decentralized-KPD 中的 BootVisor 是一个代码量很小的 Hypervisor, 其负责系统的初始化, 在初始化阶段运行在 root 模式, 并采用预分配[27]的方法为内核和每个保护域分配物理内存. 同时, BootVisor 利用硬件支持多套 EPT 特征在初始化过程中为内核与每个保护域都配置了一套页表 EPT. 其中, 如图 1 所示, BootVisor 为 Guest OS 配置了 K_EPT, 而为保护域分配的是 KPD*n*_EPT(*n* 表示第 *n* 个保护域的编号). Decentralized-KPD 利用 EPT 页表项中 RWX 权限设置多域中每个保护域内不同组件的内存视图, 实现多域的内存隔离.

Decentralized-KPD 中, Guest OS 与保护域的内存映射关系如图 2 所示. 本文依据保护域的数量将整个 GVA 地址空间划分为多个内存区域, 并为内核与保护域创建不同的内存视图. 其中, 内存区域主要包括 Guest OS 内存区域、多个保护域区域以及 Gateway 区域. Guest OS 内存区域主要用于内核和应用程序. 而对于保护域内存区域, 每个保护域都有自己独立的内存区域并不共享. 与此同时, 为了获得较强的内存隔离, Decentralized-KPD 会如图 2 所示为每个保护域的内存区域设置不同权限. 在 Decentralized-KPD 中, Guest OS 运行时, 其所属的 K_EPT 只映射 Guest OS 内存区域, 对于其他保护域的内存区域, 则要么是不映射要么映射为只读. 因此, Guest OS 实际上并不能任意地访问其他保护域内的代码和数据. 而在所有 EPT 页表项中, 都是不映射 BootVisor 所在的内存区域的. 当然, 如果保护域的安全机制涉及内核需要保护的关键数据, 也可以在 K_EPT 中将该数据的内存区域设置为只读. 当保护域内的安全机制运行时, 其利用所属的 KPD*n*_EPT 将 Guest OS 内存区域设置为只读.

Gateway 是 Guest OS (内核)与保护域之间唯一的通道, 实现内核与保护域的地址空间的切换. 同时, Gateway 也是内核与保护域唯一共享的页面. Decentralized-KPD 在初始化阶段就将所有 Gateway 的页面在 K_EPT 设置为只读和可执行的, 而 KPD*n*_EPT 中则只映射 KPD*n* 对应的 Gateway 为只读和可执行的, 其他保护域对应的 Gateway 则不映射.

Decentralized-KPD 通过分离地址空间和配置 EPT 页表的表项实现了内核与保护域以及保护域之间的内存隔离. 由于保护域在 Guest OS 的 K_EPT 中没有映射或者映射为只读, 因此内核对任意保护域任意恶意的修改都会引起 EPT_VIOLATION 从而导致 VM Exit, 而在 Decentralized-KPD 任何恶意操作引起的 VM Exit 都会导致系统的重启, 从而阻止攻击者的行为. 而在单个保护域中, 其他保护域的内存区域在此保护域所属 EPT 页表项中是没有映射的, 而 Guest OS 是映射为只读的. 因此, 任何保护域对内核区域和其他保护域的任意写操作都会同样引起 EPT_VIOLATION 从而导致 VM Exit. 当然, 在保护域内的安全机制为了实现针对内核的安全服务, 可以对内核区域进行读写.

地址翻译完整性. Decentralized-KPD 利用硬件辅助内存虚拟化技术 EPT 页表实现了内存隔离, 但是只控制了第 2 层地址转换, 而内核与保护域以及保护域之间是共享第 1 层地址转换的 GPT 页表. Decentralized-KPD

在配置每个保护域的执行环境时为其配置了一套 GPT 页表. 当运行保护域内的安全机制时, 其 GVA-to-GPA-to-MPA 都是依据 Decentralized-KPD 分配的页表, 无论是在第 1 层页表还是在第 2 层页表, 都不用与内核或者其他保护域共享一套页表, 并从而保证每个保护域的地址翻译的完整性. 并且, 现有的保护域都是针对内核的, 其地址空间仍位于内核地址空间的范围之内, 不需要考虑用户空间的映射. 同时, 保护域的内存区域都是预先分配页表, 并且设置了页表的映射, 运行过程不需要处理缺页的情况.

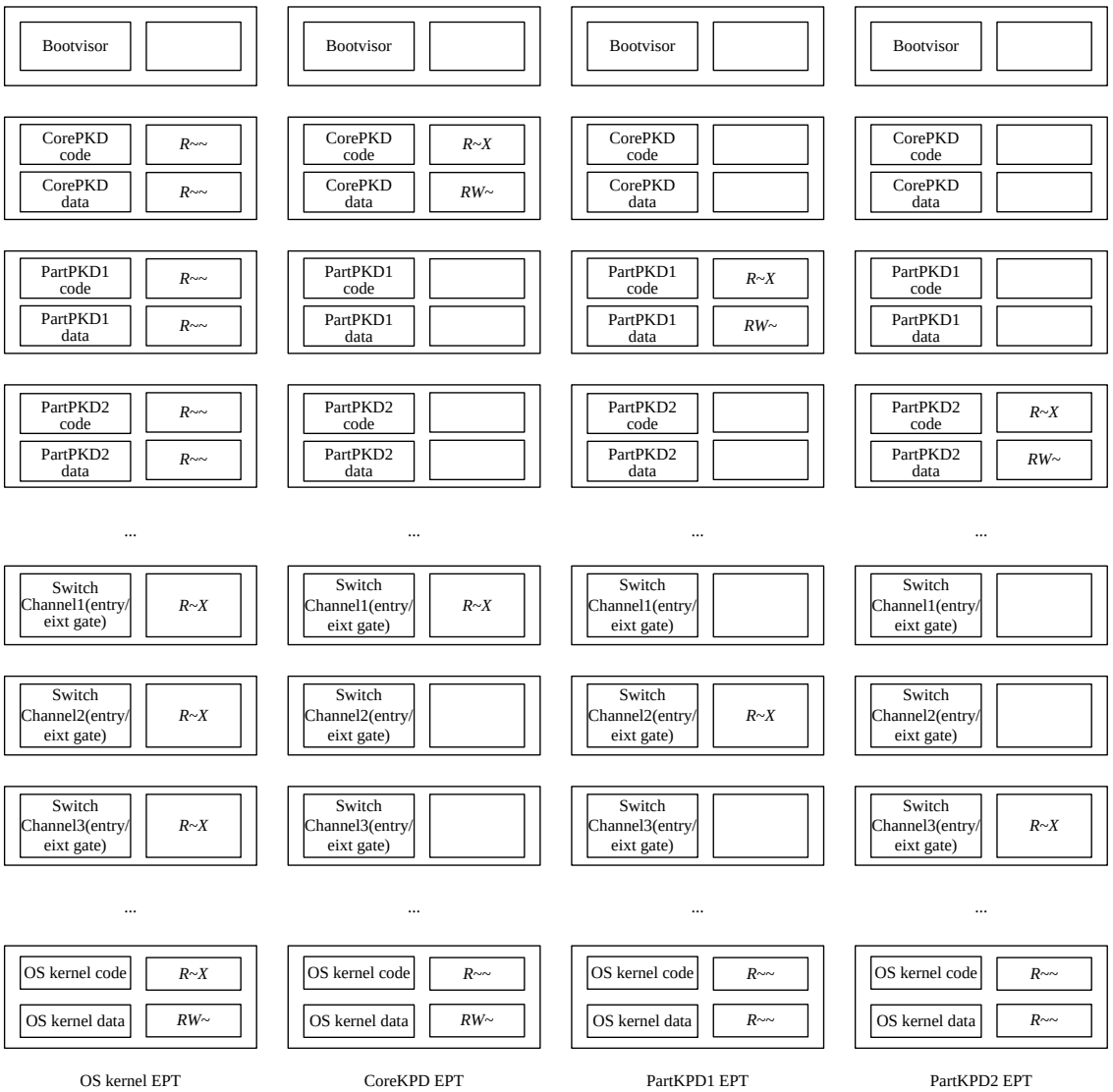


图 2 内核域与多保护域的内存视图

3.2 多域的固定入口点隔离

3.2.1 Gateway 的设计

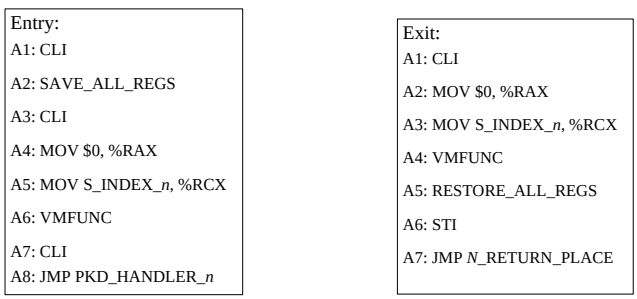
如图 1 所示, Decentralized-KPD 架构中, 每个保护域都有安全机制的代码和数据, Guest 层的页表 KPD*n*_GPT 和 root 模式下的 KPD*n*_EPT 页表, 以及用于内核与保护域之间切换通道 Gateway (entry/exit gate). 为实现内核与不同保护域之间快速、有效的切换, Gateway 的设计与实现必须符合不可绕过、不可更改以及原

子性这 3 条属性.

在 Gateway 的设计中, Decentralized-KPD 利用 Intel 硬件辅助虚拟化提供的 EPTP-Switching 功能. 即: 通过执行 VMFUNC 指令, Guest 可以在不引起 VM Exit 时直接完成底层 Hypervisor 中 VMCS 的 EPTP 域内值的加载, 完成 EPT 的切换, 通过 EPT 的切换, 则完成了内核与保护域之间的切换. Decentralized-KPD 在初始化过程中即将内核的 K_EPT 以及保护域的 KPD*n*_EPT 的目标值加载到 VMCS 中 EPT_LIST_ADDR 对应的页, 其中, K_EPT 索引为 0, 而 KPD*n*_EPT 则是依据索引 *n* 依次排列.

如图 3 所示, Gateway 主要包括 Entry/Exit Gate. 在 Entry Gate 中, 首先都是关中断, 然后保存内核上下文信息, 并将 RAX 寄存器置零. S_INDEX_*n*(*n* 是对应保护域的索引)则是保护域 *n* 对应的 KPD*n*_EPT 在 VMCS EPTP list 中的索引. KPD_HANDLER_*n* 则是指向保护域内对应的安全机制.

如图 3 所示, Gateway 中 Exit Gate 的执行过程与 Entry Gate 正好相反, 所不同的是, 所有保护域退出时都直接退出到内核域, 因此 RCX 均赋值为 0, 并且在控制流转换回内核的同时开启中断.



(a) 入口通道 (b) 出口通道

图 3 Gateway 转换通道

上面介绍了 Decentralized-KPD 中 Gateway 的设计与实现, 下面详细介绍 Decentralized-KPD 中的 Gateway 是如何满足其 3 条属性的.

1) 不可绕过. Decentralized-KPD 在内核代码的关键位置放置钩子, 然后通过 Gateway 转移到确定的保护域. 在多域内存保护中, 利用 Root 模式锁机制保护的 EPT 页表在内核与所有保护域中设置了 $W \oplus X$ 保护. 内核代码是不可更改的, 因此, 内核执行到这里时必然会触发该钩子. 虽然 Decentralized-KPD 有多个 Gateway, 但是每个 Gateway 都与对应的保护域配对, 任意错乱的调入执行都会引起 VM Exit;
2) 不可更改. 在 Decentralized-KPD 中, Gateway 被设计处在一个页面上, 并且在对应的 EPT 页表项中被设置为只读且是可执行的. 同时, Decentralized-KPD 利用 root 模式锁机制保护域 EPT 不被恶意修改, 因此, 任意对 Gateway 的恶意修改都会引起 VM Exit;
3) 原子性. Gateway 中, Entry/Exit Gate 的起始处第 1 条指令就是 CLI 关中断指令, 而且 Entry Gate 中执行 VMFUNC 指令前又添加了一条 CLI 指令, 确保 Gateway 在执行 VMFUNC 指令是原子性的. 为了进一步保护 Gateway 的执行, 本文利用 SIM[9]中提到的 Gateway 保护机制强化了对 Gateway 的保护.

3.2.2 多域的切换

在第 3.1 节中, Decentralized-KPD 为了维护内核与保护域地址翻译的完整性, 在每个保护域内为其配置第 1 层页表 GPT. 在 Intel x86 芯片架构中, CR3 指向当前程序运行过程使用的页表, 并且通过 MOV_TO_CR3 指令修改 CR3 寄存器目标值, 实现页表的切换. 但是, 如果在 Gateway 中添加 MOV_TO_CR3 指令, 不仅容易增加内核与保护域之间切换的性能开销, 而且也无法实现内核域与保护域的 Guest 层和 Host 层两套页表的同步切换. 更重要的是, 将 MOV_TO_CR3 指令添加到 Gateway 中会增加指令的数量使其更难以简化, 而且也可能会增加新的攻击面. 攻击者可能会实现破坏 CR3 寄存器的目标值. 因此, 本文在 Decentralized-KPD 中利用地

址重映射技术实现了 EPT 页表与 CR3 指向的 GPT 页表的同步切换.

Decentralized-KPD 通过修改 EPTs 中 CR3 寄存器目标值对应的页表项, 实现同步完成 GPT 与 EPT 的切换. 首先, CR3 寄存器赋予一个固定的值(我们称其为 cr3-gpa), 在内核域 K_EPT 对应的页表项中, cr3-gpa 最后指向了预先配置的 Guest 页表的页目录的实际物理页(sk-gpt); 而在保护域 1 的 KPD1_EPT 中, cr3-gpa 最后指向了预先为保护域 1 配置的 Guest 页表的页目录的实际物理页(s1-gpt). 这样, 当 EPT 页表在 K_EPT 与 CoreKPD_EPT(或者 PartKPD*n*_EPT)之间进行切换时, 同时也更新了 CR3 寄存器实际指向的 Guest 页表. 内核与其他保护域之间也是类似地完成两层页表的切换. 因此, 当 VMFUNC 的功能是 EPTP-Switching 时, 结合地址重映射技术, 在执行 VMFUNC 指令时, 可以完成 EPT 和 GPT 的同步切换. 当然, 在系统运行过程中必须确保 CR3 寄存器的目标值(cr3-gpa)一直是固定不变的. Decentralized-KPD 在系统初始化过程中就为 CR3 寄存器预先配置了 cr3-gpa, 并在运行过程中利用硬件虚拟化技术禁止对 CR3 寄存器的修改, 任何对 CR3 寄存器的修改都会引起 VM Exit. 于是, 在 Decentralized-KPD 中, 一方面消除了所有 MOV_TO_CR3 指令, 另一方面禁止内核或者保护域引入新的 MOV_TO_CR3 指令. 但与此同时, 消除 MOV_TO_CR3 指令又带来了内核进程切换和 TLB 刷新问题, 其解决思路如下.

1) 进程切换问题. Decentralized-KPD 中固定了 CR3 寄存器目标值, 同时, 通过在 VMCS 中设置相关的域, 禁止 MOV_TO_CR3 指令的执行. 但在内核正常运行过程中, 进程的切换需要通过 MOV_TO_CR3 指令完成页表的切换. 为了解决上述问题, Decentralized-KPD 在内核中借鉴了 APPISO[19,34]中软切换机制, 即, 利用 x86-64 页表分级中最高级目录页的复制操作代替 MOV_TO_CR3 指令操作. 由于进程共享内核地址空间, 只需要复制用户地址空间的页表, 也就是半个页大小, 从而提高软切换的效率;
2) TLB 刷新问题. MOV_TO_CR3 指令不仅实现了页表的切换, 同时也在页表切换的同时完成了 TLB 的刷新. 软件切换虽然解决了页表的切换问题, 但不能实现 TLB 的刷新. Decentralized-KPD 为了解决上述问题, 引入了硬件辅助虚拟化提供的 CR3-Target List 技术. 其中, 硬件虚拟化技术在 VMCS 中提供 4 个 CR3_TARGET_LIST 地址域, 用于存放 CR3 寄存器的值. 当 MOV_TO_CR3 指令更新 CR3 与 CR3_TARGET_LIST 中的值相等时, 执行 MOV_TO_CR3 指令时不会引起 VM Exit. 于是, Decentralized-KPD 在初始化过程将之前预先加载到 CR3 寄存器中的值赋予在 VMCS 的 CR3_TARGET_LIST, 同时保留了内核代码中 TLB 刷新代码中的 MOV_TO_CR3 指令. 于是, 当需要通过 MOV_TO_CR3 指令刷新 TLB 时, CR3 寄存器目标值已经提前在 Decentralized-KPD 初始化过程中写入 VMCS 的 CR3_ TARGET_LIST 中, 这样, 既可以利用 MOV_TO_CR3 指令实现 TLB 刷新, 同时又不会影响到进程的正常切换.

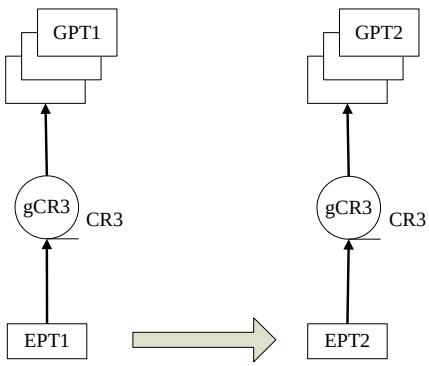


图 4 CR3 寄存器页表重映射图

3.3 多域的控制流隔离

Decentralized-KPD 中的基本保护域与功能保护域中的影子 IDT 的表项都设置为中断门, 从而使得控制流从内核转入到保护域时直接进入关中断状态, 使得控制流从内核转向保护域时保护域就屏蔽了可屏蔽中断.

为了控制内核中断与异常对保护域内控制流的影响, Decentralized-KPD 中内核依然维护其自身的 IDT, 而 CPU 实际使用的是基本保护域维护的 Shadow IDT (shadow interrupt description table, SIDT). 在内核运行过程中发生中断和异常时, 基本保护域会通过影子 IDT 截取中断和异常, 然后将其发送给内核, 由其完成中断的处理. 而对于不可屏蔽中断, 为了防止外部组件(包括被攻陷的保护域)劫持未被攻陷保护域内的控制流, Decentralized-KPD 为每个保护域配置了一个影子 IDT, 避免了不同保护域之间共享同一个 IDT. 当保护域内发生不可屏蔽中断时, 利用已经配置好的 IDT 暂时阻塞 NMI, 当返回到内核时, 保护域将该 NMI 转发给内核, 由其进行处理. 在 Decentralized-KPD 中, 为基本保护域预先配置了影子 IDT(SIDT), 而功能保护域则是 PIDT (protection interrupt description table), 并且确保每个保护域的 IDT 不被恶意的内核或者其他被攻陷的保护域所破坏.

1) x86 芯片中, IDTR 寄存器确定了 CPU 正在使用的 IDT, Decentralized-KPD 利用硬件虚拟化技术禁止内核与任何保护域执行 lidt 指令修改 IDTR 寄存器;
2) 影子 IDT (SIDT 和 PIDT*n*)与其处理程序通过 EPT 页表都被设置为只读, 禁止内核与任何保护域对其进行恶意修改. 而且 EPT 页表是被 Decentralized-KPD 中利用 Root 模式锁机制进行保护的;
3) 由于 IDTR 与影子 IDT 表项均是虚拟地址, 其在页表项中映射(EPT 页表和 Guest 层页表 KPT)均设置为只读. 同时, 基本保护域中通过截取内核页表的操作实现对页表项的保护, 从而实现内核对应的 IDT 保护.

基本保护域与功能保护域都拥有自己的 IDT. 在 Decentralized-KPD 利用 EPT 页表的重映射, 与 Guest 层 CR3 寄存器指向的 GPT 一样, 固定 IDTR 寄存器的目标值, 当 EPT 页表改变时, IDTR 寄存器指向不同的 IDT. 因此, 在 Decentralized-KPD 中内核与保护域切换的同时, 也切换了不同保护域的 IDT.

3.4 多域的Root模式锁机制

图 2 详细描述了 Decentralized-KPD 的内存映射关系. BootVisor 在初始化阶段负责管理所有的 EPT 页表, 为其分配内存, 并在配置 EPT 页表时只能映射分配给 Guest 层的实际物理内存. 因此, 在运行阶段, 无论是 Guest OS, 还是保护域内的安全机制都不能访问 BootVisor, 亦即不能访问其管理的 EPT, 更不能对其进行任意的改写. 攻击者不能任意篡改或者破坏底层的 EPT, 也就不能破坏内核与多个保护域以及多个保护域之间的内存隔离.

同时, 为了防止攻击者通过特定事件恶意陷入到 root 模式而破坏 Decentralized-KPD 的运行, 在 Decentralized-KPD 中严格限制运行过程中 non-root 模式到 root 模式的切换. 现有的 Intel VT 技术将产生 non-root/root 模式切换事件主要分为 3 类: 执行有条件产生 VM Exit 的指令、执行无条件产生 VM Exit 指令以及非指令事件导致的 VM Exit.

为了避免频繁的特权层切换, Decentralized-KPD 允许 Guest OS 直接访问硬件资源并负责管理. 通过在 VMCS 中相应执行控制域进行设置, Guest OS 可以处理诸如中断、异常等事件. 为了防止攻击者通过恶意修改相应的寄存器从而破坏内核与多保护域的隔离完整性, Decentralized-KPD 通过在 VMCS 中设置相应域的值, 使得攻击者对寄存器相应位(比如 CR0 寄存器中的 WP 位、CR4 寄存器的 PAE、VMXE 位等)的修改都会引发 VM Exit 从而陷入到 root 模式中.

在 non-root 模式中, CPUID 指令会无条件地产生 VM Exit. 在 Decentralized-KPD 中的处理方式是: 在 BootVisor 过程中(仍处于 root 模式)执行 CPUID 指令, 并将所有结果保存在特定的内存区域. 之后, 内核和应用程序获取 CPUID 信息, 只需要从保存的数据区域中直接读取而不用执行 CPUID. 而且 CPUID 指令只在极少库函数(libc)中执行, 只需要在系统启动前将 libc 库中的指令替换掉即可. 当然, 更简单的办法就是在

BootVisor 中保留处理 CPUID 指令的极小的一段代码.

为了防止 Decentralized-KPD 中 BootVisor、KPD 和 Gateway 等组件所在的物理内存被 DMA 攻击, Decentralized-KPD 利用 IOMMU 硬件机制保护其物理内存数据. 同时, 为了防止设备中已经分配的 I/O 内存基地址被破坏, 在初始化之后, 禁止对 PCI/PCIe 空间的访问.

Decentralized-KPD 中实现了 root 模式锁机制以及为每个保护域创建了独立的双层页表, 有效维护了内核与保护域以及保护域之间的内存隔离完整性, 同时也保证了内核与保护域内的地址访问完整性. 另外, Decentralized-KPD 通过 root 模式锁机制确保了系统整体在正常运行时只能运行在 non-root 模式下.

3.5 多域中保护域内的安全机制

- 内核页表保护. Decentralized-KPD 中创建了基本保护域, 在基本保护域中, 利用影子页表实现了对内核页表的保护. Guest OS 依旧会维护其自身的页表, 但实际使用的是基本保护域中维护的页表. 在基本保护域中, 利用半虚拟化接口对所有页表进行截取和验证, 从而实现对内核页表 GPT 的保护;
- 系统调用跟踪. Decentralized-KPD 在 Guest OS 中系统调用处设置钩子, 当在 Guest OS 中发生系统调用时, 将控制转换到功能保护域中, 由功能保护域中的安全机制代码实现对系统调用进行跟踪, 并最后回到 Guest OS, 由其完成系统调用的具体工作. 在 64 位系统中, 往往利用 syscall/sysret 快速系统调用指令完成系统调用, 因此, 此时只考虑这种情况下的系统调用. 因此, 在系统初始化过程中, 将 MSR_LSTAR 寄存器中的值重定向到系统调用追踪保护的功能保护域. 同时, 为了保证截取系统调用的完整性, 对于 Decentralized-KPD, 在 EPT 页表中将系统调用表设为只读保护, 同时利用 Root 模式锁机制禁止攻击者对 MSR_LSTAR 等寄存器的修改;
- 进程创建监控. 在 Decentralized-KPD 中设置了一个功能保护域, 用于实现对进程创建的监控. 在内核 *fork* 函数中插入钩子, 监控进程创建的过程, 并最后由 Guest OS 完成进程的创建.

4 系统实现

本文在 Linux 3.12.1 (64 bit)上实现了 Decentralized-KPD 原型系统. 在目前的实现方案中, Decentralized-KPD 采用了简单的、预设定的保护策略, 即, 对所有运行在保护域内的安全机制都是预先定义的. 在实现的原型系统 Decentralized-KPD 时, BootVisor 模块则根据 Intel 手册的内容实现, 并增加了系统内存静态配置的内容. Decentralized-KPD 系统实现页表的截取时, 需要 Linux 内核中半虚拟化接口的支持, 因此在编译内核时, 需要开启半虚拟化的支持. 同时, 为了实现第 3.5 节中功能保护域内所提供的安全服务, 需要在内核源代码相应的位置插入钩子. 本文中, Decentralized-KPD 是在单核体系下实现的.

在当前实现中, 整个原型系统 Decentralized-KPD 包含大约 6.5K 行代码.

4.1 初始化阶段

BootVisor 负责系统初始化, 其主要工作如下.

1) 在系统初始化过程中, 首先检查系统硬件是否支持虚拟化功能, 尤其是对 VMFUNC 指令的支持. 同时, 构造 BootVisor 初始化所需运行环境, 包括所需要的内存空间. 然后运行 VMXON 指令, 开启硬件虚拟化功能. 分配 VMCS 区域(VMCS region)并设置相应的字段, 尤其是要注意 HOST_RIP、GUEST_RIP 以及 CR3 Target-List 等, 其中, 针对 non-root 模式下 Guest OS 运行时的一些控制可见表 1;
2) 如图 1 所示, BootVisor 在配置 non-root 模式执行环境之后, 为每个保护域分配所需的内存空间, 根据分配的内存区域配置保护域和内核的 EPT 页表结构以及所有保护域的 Guest 层页表, 并在 EPT 页表中设置相应的权限. 同时, 将所有 EPT 的基址赋值到 EPTP-list 中. IDTR 寄存器中的虚拟地址指向系统所使用的 IDT 表, 修改 EPT 页表中相应的页表项, 使其对应的地址最终在切换不同的 EPT 时指向不同的 IDT 表. Decentralized-KPD 借鉴 APPISO[19]中的软切换技术, 首先会分配一个物理页

作为 CR3 寄存器一直指向的物理页, 并将物理页的物理地址(GPA)写入 VMCS 中的 CR3_ADDRESS 域中;

3) 使用 VMLAUNCH 指令将 Guest OS 运行状态切换到 non-root 模式. 依据 VMCS 的设置, 可以保证 Guest OS 在运行过程中无需受到 BootVisor 干扰. 初始化完成之后, BootVisor 让整个系统运行在 non-root 模式, 任何非授权的访问都会引起 VM Exit 而陷入到 root 模式.

表 1 BootVisor 启动时 VMCS 的配置

VMCS配置	作用
MSR bitmap	禁止non-root模式修改EFER等相关MSR寄存器
I/O bitmap	禁止non-root模式访问PCI/PCIe配置空间
CR0 CR4 shadow	禁止non-root模式对CR0 CR4寄存器相关位的修改
CR3 Target controls	启动non-root模式MOV_TO_CR3指令不会产生VM Exit
Enable EPT	开启EPT机制
Enable VM functions	开启VM functions机制
Mov to CR0	禁止CR0操作指令, 需要与CR0 shadow配合使用
Mov to CR3	禁止CR3操作指令, 需要与CR3 target-list配合使用
Mov to CR4	禁止CR4操作指令, 需要与CR4 shadow配合使用
Mov to CR8	禁止CR8操作指令
Descriptor-table exiting=1	禁止non-root模式修改IDTR寄存器
CR3 target count	开启CR3 Target List功能

4.2 基本保护域的实现

BootVisor 初始化过程中, 开启了内核的半虚拟化接口. 在 Decentralized-KPD 中, 利用 Linux 的半虚拟化接口实现对内核 MMU 操作的截取. Decentralized-KPD 在内存操作的半虚拟化接口中设置 Gateway, 将控制流截取到基本保护域内, 完成内核 Guest 层页表的操作. Decentralized-KPD 的基本保护域(CoreKPD)完成了内核 Guest 层页表的同步和必要的验证, 且使用半虚拟化接口会带来明显的好处, 包括: (1) 减小 Decentralized-KPD 的 TCB 规模; (2) 提高了系统的性能; (3) 不需要修改原有的内核代码. 同时, 利用基本保护域维护的影子 IDT 截取和验证内核中的中断和异常.

4.3 功能保护域的实现

在系统初始化过程中, Decentralized-KPD 将重写 MSR_LSTAR 寄存器, 将其指向功能保护域 1(PartKPD1), 从而实现对系统调用追踪的保护. 同时, 在内核进程创建的关键函数 *fork* 中插入钩子, 通过功能保护域 2 (PartKPD2)对应的 Gateway, 将控制转换到功能保护域 2 中, 从而实现对进程创建的监控.

5 系统评价

本节将从安全性和性能两个方面对基于硬件虚拟化的内核同层多域隔离模型 Decentralized-KPD 加以详细分析.

5.1 安全性分析

为了确保针对内核的多样化安全机制的保护, 基于硬件虚拟化的内核同层多域隔离模型在内核同层构建了多个保护域, 对于每个保护域而言, 其必须具备与传统单保护域一样的安全性; 而对于多个保护域而言, 其在具备前者的基础之上还必须保持各自的独立性.

5.1.1 多域的安全属性分析

在内核同层多域隔离模型的安全假设下, 对于其中的每个保护域, 须满足如下安全属性.

1) 数据代码完整性. 单个保护域内的数据代码不能被外部不可信组件(不可信内核或者其他保护域)读写或者执行;

2) 入口点完整性. 外部组件(内核)只能从指定的入口点实现组件与保护域之间的切换;

3) 执行流的完整性. 保护域内的安全机制在运行时, 其控制流不受外部组件(包括内核和其他保护域)的干扰.

接下来, 将从 Decentralized-KPD 启动以及运行过程表现分析每个保护域在内核同层多域隔离模型中均满足上述基本安全属性.

系统环境初始化: 系统启动时, 运行在 root 模式的 BootVisor 依据保护域的数量为其预先分配好内存和相应的 EPT 页表, 并在 EPT 页表中, 根据图 2 中设计的内存视图设置每个保护域数据和代码区域相应的权限. BootVisor 初始化完成之后, 构建了整个系统的安全执行环境, 将 Decentralized-KPD 固定运行在 non-root 模式, 确保 Decentralized-KPD 中每个保护域的所有安全属性都能满足. 在 non-root 模式中, Decentralized-KPD 通过在内核代码中设置钩子控制进入保护域的入口点. 而此时, 内核和应用程序还未运行, Decentralized-KPD 可以通过可信启动硬件保证其映像的完整性. 因此, 由 Decentralized-KPD 设置的执行环境认为是可信的.

系统运行时, Decentralized-KPD 依据 BootVisor 为内核以及每个保护域建立 EPT 页表, 实现了内核与每个保护域的内存隔离. 同时, 为每个区域的代码和数据设置了不同的权限. 在内核的内存权限视图中, 保护域内的数据代码都是只读或者不映射的, 因此内核不能执行保护域内的任何代码, 也不能任意修改保护域内的数据和代码. 任何非法的访问都会引起 EPT_VIOLATION 触发 VM Exit 从而陷入到 root 模式. Decentralized-KPD 建立 root 模式锁机制, 将系统固定运行在 non-root 模式, 防止攻击者恶意破坏底层的 EPT 页表, 保证了保护域内代码和数据的完整性. Decentralized-KPD 中, 利用硬件虚拟化技术确保内核与保护域之间的切换只能通过对应的 Gateway 来实现, 任何通过其他非法方式实现的切换都会引起系统的 VM Exit. 其中, Gateway 包括 Entry Gate 与 Exit Gate, 每个保护域都配置独立的 Gateway, 其中, EPT 对应索引(由 Gateway 中 RCX 中的值确定)即是保护域的索引, 二者之间是一一对应的. 对于正常的切换(比如请求针对内核的安全服务), 内核通过 Entry Gate 切换进入保护域, 并利用 JMP 指令跳转到保护域内安全机制代码, 完成内核安全服务请求, 比如验证系统调用. 当保护域内安全机制完成了相应服务时, 会通过 Exit Gate 退回到内核. Gateway 的切换页面对于内核与保护域而言都是受到保护的, 被设置为只读和可执行, 禁止任何恶意的修改. 此外, 当保护域内的安全机制运行之前, 在 Gateway 的 Entry Gate 中首先就会执行 CLI 指令. 与此同时, 保护域内代码执行时有对 NMI 进行延期处理, 因此内核无法破坏保护域内安全机制的控制流. 当保护域从内核获得控制之后, 其运行只会按照保护域内的代码进行.

从上面分析可以看出, 每个保护域都与单保护域一样具有同样的安全属性. 但在多域隔离环境下, 不仅要保证每个保护域的可信度, 同时必须保证多个保护域整体的独立性.

5.1.2 多域抵御攻击行为分析

由于在 Decentralized-KPD 的安全假设中, 功能保护域内安全机制的代码中可能存在漏洞, 攻击者可以利用其中的漏洞攻陷其所在的保护域. 攻击者在攻陷一个功能保护域之后, 可能利用该保护域攻击其他功能保护域, 而攻击行为包括攻击内存数据、IDT 重映射攻击、破坏系统硬件执行环境、恶意 VMFUNC 攻击等. 接下来, 本文进一步分析 Decentralized-KPD 是如何抵御上述攻击的. 本文的假设主要考虑这 4 类攻击, 如第 2.2 节所述, 本文不考虑拒绝服务攻击和物理层面的攻击. 其中, 攻击者具体的攻击行为如下所示.

1) 攻击内存数据.

- 直接内存访问攻击: 攻击者利用保护域内安全机制代码中可能存在的漏洞攻陷该保护域, 其可以访问整个保护域内的地址空间, 可以直接读写、执行保护域内的所有代码和数据, 破坏其他安全机制的私密性和完整性;
- 修改页表攻击: 页表是保护域内保护的关键数据, 一旦攻击者利用保护域内安全机制代码中可能存在的漏洞攻陷保护域, 攻击者就可以通过修改页表实现对保护域内的页框进行恶意映射(修改现有映射或者重映射), 破坏其他内存区域设定的权限保护(如对关键数据的只读保护, 对栈和代码的 $W \oplus X$ 保护);
- DMA 攻击: 直接内存访问(direct memory access, DMA)是现代 I/O 设备中普遍支持的一种数据传输机

制, 允许在内存和 I/O 设备之间直接高速传送数据. 攻击者可能操作被攻陷的保护域内与 DMA 设备的相关的代码, 恶意读写保护域内任意物理内存数据;

2) IDT 重映射攻击: 攻击者可以利用 lidt 指令加载新的 IDT, 或者修改页表, 重定向攻击者自定义的 IDT, 从而截取内核的控制权.

3) 硬件上下文攻击: 攻击者仍可利用共享的硬件破坏保护域的安全执行, 其可以直接修改 CR0 寄存器的 PE、PG, WP 位、CR4 中的 PAE 位以及 IA32_EFER MSR 寄存器中的 LME 位, 从而破坏内核或者保护域的执行环境.

4) 恶意 VMFUNC 指令攻击: VMFUNC 指令可以在内核态运行, 也可以在用户态运行. 攻击者可以利用用户态或者内核加载模块中存在的 VMFUNC 指令, 从而实现恶意的 EPT 切换.

接下来, 结合上述具体攻击行为, 我们系统地分析 Decentralized-KPD 是如何防御上述每一类攻击的.

1) 攻击内存的防御.

为了防御直接内存访问攻击, Decentralized-KPD 将不同的安全机制部署在不同的保护域内, 并对每个保护域设置不同的权限, 构建不同的内存视图(如第 3.1 节所述). 攻击者利用被攻陷的功能保护域对其他保护域进行任意的写操作都会触发 EPT VIOLATION, 从而引起 VM Exit 陷入到 root 模式. 因此, 攻击者不能对其他功能保护域进行直接访问攻击.

为了防御页表破坏攻击, Decentralized-KPD 同时为每个保护域分配了 Guest 层和 Host 层这两层独立的页表, 避免不同保护域之间共享页表. 如第 3.4 节所述, Decentralized-KPD 利用 Root 模式锁机制禁止对底层 EPT 作任何形式的修改. 同时, 在保护域内其只能访问自身的 Guest 页表, 不能访问其他功能保护域的页表.

为了抵御 DMA 攻击, Decentralized-KPD 利用 IOMMU 机制防止任何恶意的 DMA 操作向所有保护域的内存区域所属的物理内存进行读写. 同时, 为了保护分配给设备 I/O 内存的基址防止其被修改, Decentralized-KPD 中禁止 Guest OS 访问 PCI/PCIe 的配置空间.

2) IDT 重映射攻击的防御.

如第 3.3 节所述, Decentralized-KPD 为了防止攻击者对 IDT 表的攻击, 为每个保护域都分配了独立的 IDT, 并且在初始化时将 IDT 设置为只读保护. 另外, Decentralized-KPD 中在 VMCS 设置禁止 lidt 指令的执行, 在初始化之后, 任何 lidt 指令的执行都会引起 VM Exit, 从而禁止攻击者执行 lidt 指令加载新的 IDT.

3) 硬件上下文攻击防御.

如第 3.4 节所述, Decentralized-KPD 在系统初始化过程中已经设置了相应寄存器的状态, 其硬件上下文包括 CR0 中的 PE、PG, WP 位、CR4 中的 PAE, VMXE 位以及 EFER MSR 中的 LME 位等. 在 Decentralized-KPD 中, 通过对 VMCS 相应域的设置, 禁止攻击者对 CR0、CR4 等寄存器的恶意修改, 攻击者的任何修改都会引起 VM Exit. 同时, 为了保证功能保护域的正常运行, 在完成系统初始化之后, Decentralized-KPD 也禁止攻击者对 MSR_ LSTAR 和 MSR_STAR 寄存器的修改.

4) 恶意 VMFUNC 指令攻击的防御.

Decentralized-KPD 在初始化过程中设置 SMEP, 攻击者利用用户态可能存在的 VMFUNC 指令时会产生缺页异常. 为了阻止攻击者利用加载的内核模块中可能存在的 VMFUNC 指令, 当前的实现方案要求系统管理员首先完成对所有内核可加载模块的线下分析, 防止其混入 VMFUNC 指令. 当内核需要加载内核模块时, Decentralized-KPD 可以截取这一过程并验证其代码中是否存在 VMFUNC 指令. 一旦发现, Decentralized-KPD 将禁止内核模块加载. 这一方法同样适用于内核模块中可能存在的 MOV_TO_CR3 指令.

5.2 性能分析

本节将从 Decentralized-KPD 内存访问、切换开销以及内存空间消耗这 3 个方面进行分析.

5.2.1 Decentralized-KPD 内存访问性能分析

Decentralized-KPD 中, 通过合理设置虚拟化环境, Guest OS 可以直接管理可访问的物理内存, 控制和访问设备. Decentralized-KPD 的设计中, Guest OS 可以直接执行特权指令, 但是一些影响 Decentralized-KPD 正常

运行的特权操作会被限制, 一旦触发 VM Exit 便会陷入到 root 模式. 为了限制 Guest OS 对 CR3 寄存器的恶意修改, Guest OS 引入软切换技术实现进程切换 MOV_TO_CR3 指令的替换. 而为了支持 TLB 刷新, 利用硬件虚拟化的 CR3 Target-List 技术, 使得 Guest OS 在执行 CR3 时具有特定值的 MOV_TO_CR3 指令, 不会陷入 root 模式, 从而避免了不必要的频繁的特权层切换. 另外, Decentralized-KPD 中内核与保护域运行过程中运用了内存硬件虚拟化技术, 不需要通过软件模拟实现内存虚拟化, 从而可以提高系统的性能.

5.2.2 Decentralized-KPD 切换开销分析

Decentralized-KPD 利用 Intel VT 提供的 EPTP_Swtiching 机制, 可以在 non-root 模式下完成 EPT 页表的切换, 而无需陷入 root 模式. 同时, 利用 Host 层 EPT 地址重映射技术, 在 Decentralized-KPD 的 Gateway 中实现 EPT 页表切换时, 将 CR3 寄存器目标值指向另一个实际的物理页面, 实现 Guest 层页表的切换. 于是, 即通过 EPTP Switching 机制和地址重映射实现了 VMFUNC 一条指令, 实现了 Guest 层与 Host 层上下两层页表的切换, 有利于内核与保护域之间的快速切换.

5.2.3 内存空间消耗分析

Decentralized-KPD 通过静态方式创建了多个保护域, 并为每个保护域分配了一定大小的内存. 因此, Decentralized-KPD 所消耗的内存与保护域的数量是呈线性相关的.

相比单保护域, 将多种安全机制集中在一个保护域可以共享部分代码或者数据(比如标准的函数库). 因此, 共享的代码和数据可以减小系统内存的消耗. 于是, 单保护域内集中部署多种安全机制所消耗的内存是由所有安全机制消耗的内存减去共享内存和数据占据的内存. 而在多域隔离模型中, 每个保护域都是自包含的, 而且各个保护域之间不共享代码和数据. 于是, 多域隔离模型部署多种安全机制所消耗的内存是各个保护域所消耗的内存直接相加. 因此, 我们很容易发现, 多域隔离模型在部署同样的安全机制时相比传统的单保护域模型会消耗更多的内存.

6 实 验

本节测试 Decentralized-KPD 内核多域保护模型的效率以及整个系统的运行效率. 测试使用的硬件环境: CPU 为 Intel Core i7-9700, 3.00 GHz, 内存为 8 GB DDR3, 1 TB SATA 硬盘. 所使用的操作系统为 Ubuntu14.04 64-bit LTS, 内核版本为 Linux 3.12.1.

6.1 安全测试实验

根据本文的假设(如第 2.2 节所述), 在内核同层多域隔离模型系统中, 攻击者已经攻陷了其中的一个功能保护域, 并以被攻陷的功能保护域为基础攻击其他的保护域. 为了验证被攻陷的功能保护域可能的攻击行为, 本文在内核同层多域隔离模型的原型系统 Decentralized-KPD 中假定其中一个功能保护域(PartKPD1)已被攻陷, 通过在假定被攻陷的保护域内执行相应的攻击代码, 验证可能的攻击行为. 如第 3.1 节所述, 任何引起 VM Eixt 而陷入到 root 模式的行为都被认定是恶意的, 并且会导致系统的重启. 而系统一旦陷入到 root 模式导致重启, Decentralized-KPD 便能够轻易识别攻击者的行为, 并能有效阻止攻击的行为(所有攻击实例的 PoC 都在附录中).

6.1.1 攻击实例

根据第 5.1.2 节对系统面临攻击行为的分析以及原型系统的具体情况来考虑攻击实例的设置. 可以看出, 攻击行为主要包含几种关键攻击操作, 例如写内存、修改寄存器、跳转等. 攻击行为依赖关键攻击操作的执行, 攻击实例可以由关键攻击操作组合来构成. 本文所进行的安全测试实验涉及的关键攻击操作包括写内存、加载 IDTR、修改控制寄存器、修改 MSR 寄存器以及跳转地址等. 在原型系统 Decentralized-KPD 中, 选择 PartKPD1 作为被攻陷保护域, PartKPD2 为攻击对象. 实验测试过程中, 当 Decentralized-KPD 中的功能保护域 PartKPD1 被触发执行时, 经由其对应的 Gateway 进入保护域, 会引起攻击实例 PoC 的运行, 从而模拟第 5.1.2 节所述的攻击行为.

安全测试实验中涉及的关键攻击操作: 写内存、修改制定地址空间的代码或者数据、加载 IDTR、执行 lidt

指令使 IDTR 指向新的 IDT、修改控制寄存器、修改控制寄存器(CR0, CR4 等)的值、修改 MSR 寄存器、修改 MSR(IA32_EFER 等)的值、跳转、改变控制流以执行制定地址空间内的指令或者代码.

安全测试实验中的攻击实例.

- 攻击实例 1(直接访问内存攻击). 在安全测试中, 攻击者在被攻陷的功能保护域 1(PartKPD1)内调用函数 *test_direct_mem_access_attack*(·)执行写内存的关键攻击操作, 对功能保护域 2(PartKPD2)的内存进行修改, 模拟被攻陷的保护域对其他保护域的直接内存访问攻击;
- 攻击实例 2(修改页表攻击). 修改页表攻击与直接访问攻击类似, 其关键攻击操作是写内存操作. 因此, 攻击者在被攻陷的功能保护域 1(PartKPD1)内调用函数 *test_modify_pagetable_attack*(·), 利用写内存的关键攻击动作修改功能保护域 2(PartKPD2)中页表所在的内存, 模拟被攻陷的保护域修改其他保护域页表的攻击;
- 攻击实例 3(DMA 攻击). DMA 攻击的关键是操作外设的 DMA 请求, 使其中请求的地址指向保护域的物理地址. 攻击者在被攻陷的保护域, 通过调用内核中操作外设的代码向磁盘发送恶意的 I/O 命令, 利用 DMA 操作访问所有其他保护域的内存. Decentralized-KPD 中允许 Guest OS 直接访问外设, 只有内核才有足够的权限操作外设, 而 Decentralized-KPD 机制保护所有保护域的物理内存. 因此, 攻击者在被攻陷的功能保护域 1(PartKPD1)内调用函数 *test_DMA_attack*(·), 利用跳转内核中控制外设的函数实施 DMA 攻击;
- 攻击实例 4(重映射 IDT 攻击). 在 IDT 重映射攻击的攻击实例的 PoC 主要包括两部分: 一是利用内存的写操作实现 IDT 表修改和页表的修改; 二是利用 lidt 指令加载新的 IDT. 从攻击实例 1 和攻击实例 2 的分析可以看出: 在 Decentralized-KPD 中, 可以抵御攻击者对 IDT 和页表的攻击. 因此, 攻击者在被攻陷的功能保护域 1(PartKPD1)内调用函数 *test_idt_redirect_attack*(·), 利用加载 IDTR 的关键攻击操作, 模拟 IDT 重映射攻击;
- 攻击实例 5(修改硬件上下文攻击). 硬件上下文攻击的关键操作是修改相应控制寄存器和关键 MSR 寄存器中的特定位. 因此, 攻击者在被攻陷的功能保护域 1(PartKPD1)中调用函数 *pkd_set*/*clear_cr*0/4(·)和 *pkd_msr*(·)修改控制寄存器或者 MSR 寄存器, 模拟在被攻陷的保护域内实施硬件上下文攻击;
- 攻击实例 6(恶意执行 VMFUNC 指令攻击). 恶意 VMFUNC 指令的关键操作是攻击者执行非法的 VMFUNC 指令, 攻击者只能通过执行用户层或者加载模块中可能存在的 VMFUNC 指令实现攻击. 因此, 攻击者在被攻陷的功能保护域 1(PartKPD1)内调用函数 *test_vmfunc_attack*(·)试图将控制流跳转到用户空间, 以执行用户空间代码中可能存在的 VMFUNC 指令. 而对于内核加载模块, Decentralized-KPD 则采用线下分析的方法消除其中可能存在的 VMFUNC 指令.

6.1.2 实验结果分析

攻击实例 1(直接访问内存攻击). 实验中, 如第 5.1.2 节所述, 直接访问内存攻击在 Decentralized-KPD 原型系统触发了 EPT 层的缺页异常, 引起 VM Exit, 如第 3.1 节所述, 使系统陷入到 root 模式导致系统重启, 从而阻止了被攻陷的保护域写其他保护域的内存.

攻击实例 2(修改页表攻击). 实验中, 如第 5.1.2 节所述, 修改页表攻击在 Decentralized-KPD 触发了 Host 层 EPT 的缺页异常, 如第 3.1 节所述, 使系统陷入到 root 模式导致系统重启, 从而阻止了被攻陷的保护域写其他保护域的内存.

攻击实例 3(DMA 攻击). 实验中, 如第 5.1.2 节所述, 在 Decentralized-KPD 中, 会利用 IOMMU 禁止外设对所有保护域的物理内存写操作, 同时禁止对 PCI/PCIe 空间的破坏, 因此在测试 DMA 攻击时, 不能访问到保护域的物理地址空间.

攻击实例 4(重映射 IDT 攻击). 在测试实验中, 如第 5.1.2 节所述, 重映射 IDT 攻击在 Decentralized-KPD 中会触发 EPT 缺页异常或者执行 lidt 指令引起 VM Exit 陷入 root 模式使系统重启, 从而阻止被攻陷的保护域实施重映射 IDT 攻击.

攻击实例 5(修改硬件上下文攻击). 在测试实验中, 如第 5.1.2 节防御分析所述, 在 Decentralized-KPD 原型系统中, 对 CR0 等寄存器的修改都会引起 VM Exit 陷入 root 模式而导致系统重启, 从而阻止了被攻陷的保护域对 CR0、CR4 和 IA32_EFER 等关键寄存器的重要位的修改.

攻击实例 6(恶意执行 VMFUNC 指令攻击). 在测试实验中, 如第 5.1.2 节所述, 在 Decentralized-KPD 中, 被攻陷的保护域内试图执行用户层可能存在的 VMFUNC 指令, 会导致缺页异常, 并不能执行用户空间的任何指令, 当然也包括可能存在的 VMFUNC 指令.

可见, 表 2 所示的攻击实验测试结果符合第 5.1.2 节的防御功能的预期, 验证了在 Decentralized-KPD 原型系统中, 一个假定被攻陷的功能保护域(如 PartKPD1)对其他保护域实施攻击是不能成功的这一结论.

表 2 攻击行为实验测试结果

	攻击行为	实验测试结果
1.	直接内存攻击	VM Exit (EPT VIOLATION)
2.	页表攻击	VM Exit (EPT VIOLATION or Access to CR)
3.	DMA攻击	异常
4.	IDT重映射攻击	VM Exit (EPT VIOLATION or Access to IDTR)
5.	硬件上下文攻击	VM Exit (Access to CR or WRMSR)
6.	VMFUNC恶意攻击	异常(#PF)

6.2 性能测试实验

6.2.1 切换效率

本文使用微测量基准(lmbenchmark)来测量不同保护域与被保护对象之间的切换开销. Decentralized-KPD 中使用 EPTP switching 机制实现 Guest OS 到不同保护域(KPD)的切换. SIM[9]使用 Intel 硬件提供的虚拟化功能 CR3-Target Control 机制实现影子页表(shadow page table)的切换, 核心是通过 MOV_TO_CR3 指令修改 CR3 寄存器的值, 但又不会引起 VM Exit. Nested Kernel 中, 普通内核域与保护域之间在同一个内核特权层, 而其核心则是 MOV-TO-CR0 指令修改 CR0 寄存器中的 WP 位, 实现对页表的锁定. Application Kernel 则展示的是 ring3 与 ring0 之间的切换开销.

相比在更高特权层构建保护域, 在内核同一硬件水平构建保护域消除了内核与更高特权层之间的切换, 明显减小了内核与保护域之间的切换开销, 显著提高了系统性能. 在所有内核同层构建保护域的方案中, 由于实现机制的不同, 内核与保护域之间的切换开销也存在一定的差异. 表 3 给出了几种不同机制内核与保护域之间切换的时间. 在 Decentralized-KPD 中, 内核与保护域之间切换所花费的时间最少, 约为 0.042μs. 在 SIM[9]中, 内核与保护域之间的切换所花费时间约为 0.077 μs. 在 Nested Kernel 中, 内核与保护域的切换时间为 0.072 μs. 相比利用 VMFUNC 指令实现内核与保护域的切换, 其他实现方案的切换所需时间基本上是其两倍甚至更多.

表 3 几种不同切换机制开销(单位: μs)

EPTP switching	MOV_TO_CR3 (CR3-Target)	Non-root/root	Kernel (same privilege)	Application/Kernel
0.042	0.077	0.236	0.072	0.088

6.2.2 系统整体运行效率

为了评估 Decentralized-KPD 的性能, 本文使用 lmbench 测试用例集进行测试. 为了验证多域方案的可行性, 本文选择与现有的典型单保护域方案 Nested Kernel 进行比较. 在引言中已经介绍了 Nested Kernel 构建单保护域时的基本原理, 为此, 本文依据其思想实现了一个简易的原型系统 SKPD (single kernel protection domain). 为了验证 Decentralized-KPD 与 SKPD 二者之间的性能差异, 本文将 Decentralized-KPD 中多保护域内部署的安全机制也都部署在 SKPD 中唯一的保护域内.

为了测试 Decentralized-KPD 运行时防御的开销, 使用 lmbench 测试用例集重新测试了原型系统 Decentralized-KPD 中多种安全服务不开启时的性能开销, 以展示运行防御机制实施保护时系统性能的开销, Decentralized-KPD(normal)为系统运行时防御性能开销.

实验结果如下.

1) lmbench 中不同系统调用集测试结果.

表 4 显示了 lmbench 基准测量操作系统常见的系统调用以及实际的内存操作所需时间. 通过分析表 3 与表 4 的数据可以得出: Decentralized-KPD 在内存操作性能上不如 SKPD(如 page fault、fork、fork+exec 等系统调用的测试结果), 而对系统调用与中断异常的结果好于 SKPD(如 NULL CALL、Open/close 等系统调用的测试结果). 虽然 Decentralized-KPD 中域切换的时间明显小于 Nested Kernel 中域切换的时间, 但 Decentralized-KPD 中的测量进程切换消耗要多于 SKPD 中的进程切换. 这是因为 Decentralized-KPD 利用软切换技术[19]实现进程切换, 进程切换是通过内存复制操作完成的; 而在 Nested Kernel 中, 是通过 MOV_TO_CR0 指令实现进程切换的.

表 4 lmbench 中的性能测试结果(单位: μs)

Benchmarks	Native Linux	Decentralized-KPD (normal)	Decentralized-KPD	SKPD	Decentralized-KPD (normal) overhead	Decentralized-KPD overhead	SKPD overhead
Null call	0.05	0.052 3	0.172 7	0.216 8	1.05X	3.45X	4.34X
Open/close	0.91	0.989 5	1.257 2	1.254 4	1.09X	1.38X	1.38X
stat	0.35	0.370 1	0.496 9	0.532 3	1.06X	1.42X	1.52X
page fault	0.168 1	0.346 8	0.351 1	0.297 6	2.06X	2.09X	1.77X
protection fault	0.281 8	0.467 4	0.496 3	0.447 7	1.66X	1.76X	1.59X
signal installation	0.132 1	0.137 4	0.253 8	0.304 9	1.04X	1.92X	2.31X
signal delivery	0.884 5	0.910 2	1.046 2	1.069 7	1.03X	1.18X	1.21X
fork+exit	57.284 2	131.650 0	132.547 6	102.074 1	2.30X	2.31X	1.78X
fork+exec	193.185 2	379.866 7	384.000 0	302.833 3	1.97X	1.99X	1.57X
fork+/bin/sh	539.909 1	940.500 0	954.833 3	780.375 0	1.74X	1.77X	1.45X
ctxsw 2p/0k	0.710 0	1.460 0	1.460 0	1.210 0	2.06X	2.06X	1.70X
ctxsw 8p/16k	1.110 0	2.040 0	2.050 0	1.575 0	1.84X	1.85X	1.42X
ctxsw 16p/16k	1.290 0	2.240 0	2.200 0	1.970 0	1.74X	1.71X	1.53X

2) lmbench 中通信带宽测试结果

表 5 显示了 Decentralized-KPD 与 SKPD 运行时相应的通信带宽. Decentralized-KPD 与 SKPD 运行过程中均无需更高的特权层介入, 二者整体都具有良好的性能开销, 并且与 native Linux 性能相接近. 这也说明 Decentralized-KPD 与 SKPD 性能相接近.

表 5 lmbench 中的通信带宽测试结果

Benchmarks	Native (MB/S)	Decentralized-KPD (normal) (MB/S)	Decentralized-KPD (MB/S)	SKPD (MB/S)	Decentralized-KPD (normal) overhead (%)	Decentralized-KPD overhead (%)	SKPD overhead (%)
Pipe	6 779	5 517	5 610	6 127	81	83	90
AF UNIX	8 766	7 660	7 525	8 073	87	86	92
TCP	5 653	6 087	6 051	6 443	108	107	114
File reread	8 020.8	7 928.2	7 662.5	7 785.6	99	96	97
Mmap reread	13.8k	13.3k	13.4k	13.8k	96	97	100
Bcopy (libc)	7 967.8	7 885.9	7 968.3	7 985.1	99	100	100
Mem read	12.0k	11.0k	12.0k	12.0k	92	100	100
Mem write	8 219	8 119	8 186	8 224	99	99.6	100

综合以上测试结果可以得出, Decentralized-KPD 与 SKPD 整体上的性能差异并不是很大, 也进一步验证了多域隔离模型相比传统单保护域部署多样化的安全服务并不会引起较大的性能开销.

7 相关工作

Flicker[20]利用 Intel TXT[27]技术中的 Late Launch 机制建立与内核异步的执行环境, 保护应用程序的安全敏感的代码免受不可信内核的影响. SPECTRE[21]则利用 SMM 建立保护域对内核进行透明的检查, 从而检测可能存在的内核攻击, 包括堆喷射攻击、栈溢出攻击. Haven[22]利用 SGX 将应用程序部署在内核都不可访问的内存区域中, 从而阻止不可信内核对安全敏感应用程序的攻击. Nested Kernel[12]提出了内核同一硬件水平的保护域架构, 利用二进制重写以及 CR0 寄存器的 WP 位, 在内核同层建立了保护域. 利用上述的硬件可以构

建保护域, 但并不支持构建多个保护域. 本文则利用硬件虚拟化特征, 同时构建了多个保护域.

TZ-RKP[23]利用 ARM 平台的 TrustZone 提供的安全模式和普通模式, 将安全机制运行在安全模式, 不可信内核运行在普通模式. vTZ[35]利用 TrustZone 实现了 VM(virtual machine)层的可信执行环境(guest trusted execution environment, Guest TEE), 避免了云环境下 Guest VMs 共享 TrustZone 中的安全代码. vTZ 直接利用 TrustZone 构建了 VM 层面多个保护域, 具有更高的安全性以及灵活性. 但是 vTZ 在实现域切换时需要陷入更高的特权层, 具有较大的性能开销. TEEv[33]为了保护来源众多的可信应用程序(trusted application, TA), 将单一用途的 TEE(trusted execution environment, 可信执行环境, 也可称为保护域)扩展为可以开放的 TEE, 并为其中一个或者多个应用程序构建可信执行环境, 保证可信应用程序的安全执行. TEEv 在 TrustZone 提供的安全环境中(secure world)创建可信执行环境时都需要一个完整版的内核, 极大地增加了系统的复杂性和攻击面. 上述工作都是针对 ARM 平台的, 而本文的工作主要基于 x86 平台.

SIM[9]、ShadowMonitor[10]、SecPod[24]、Secage[25]利用虚拟化的硬件特征在内核同一硬件水平构建了一个保护域, 但在系统运行时仍然需要底层更高特权层 Hypervisor 的支持. 本文的工作在运行过程中并不需要引入更高的特权层, 保护域与内核处于同一个硬件特权层. MEMSentry[28]构建了用于保护数据的确定性保护域, 并且利用 VMFUNC 指令实现了不同域之间的切换. xMP[36]则是在引入更高的特权层 Hypervisor 后, 利用 EPT 构建了可选择的内存保护域以保护内核中的关键数据. HyperNE[37]则利用硬件的虚拟化特征 EPT 构建了内核同层的单保护域, 实现了高效的内核监控.

但是上述工作都是为了特定的安全服务而在系统中建立了唯一的可信执行环境(保护域), 而本文的工作则面向多样化的安全服务, 将不同的安全服务部署在不同的功能保护域中, 从而缓解了将多样化安全机制集中在唯一保护域内带来的安全风险, 有效避免了攻击者利用其中任意一点漏洞就可破坏整个系统的安全这一风险.

8 未来工作

如第 5 节所述, 在目前的实现方案中, Decentralized-KPD 采用的是静态的、预先定义的策略, 每个保护域内存的大小和安全服务都在系统初始化时就已确定. 未来, 我们将研究保护域的动态分配以及运行时的扩展性, 提高内核保护的灵活性和实用性. 此外, 应用程序往往需要内核的紧密支持, 而 Decentralized-KPD 主要考虑内核的保护. 在本文的研究中, 我们对利用多域隔离模型实现应用程序的保护有了一些初步研究, 并将在今后的工作中继续加以研究.

9 总 结

本文提出了一个在内核同一特权水平构建基于硬件虚拟化的多域隔离模型, 并实现了其原型系统 Decentralized-KPD. 相比传统的基于单保护域的方法, Decentralized-KPD 避免了将多样化的安全机制绑定在唯一的保护域中从而导致攻击者只要攻陷其中任何一个安全机制就能轻易破坏在同一个保护域内的所有安全机制这一风险. 因此, Decentralized-KPD 有效降低了为内核提供多样化安全服务伴随的安全风险, 而且也不会引起较严重的性能开销. 更重要的是: 现有大部分 Intel 系列芯片基本都支持硬件虚拟化, 而且 ARM 硬件平台也开始支持虚拟化, Decentralized-KPD 中多域隔离的思想同样也适用于 ARM 平台. 虽然 Decentralized-KPD 的实现过程中要求对内核源码进行一定程度的修改, 但 Linux 和 Android 都是开源软件, 具有较好的适应性.

References:

[1] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. In: Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP). New York: ACM, 1993. 203–216. [doi: 10.1145/168619.168635]

[2] Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Proc. of the Network and Distributed Systems Security Symp. (NDSS). Rosten: Internet Society, 2003. 191–206. https://www.ndss-symposium.org/ndss2003/virtual-machine-introspection-based-architecture-intrusion-detection/

[3] Laureano M, Maziero C, Jamhour E. Intrusion detection in virtual machine environments. In: Proc. of the 30th EUROMICRO Conf. Piscataway: IEEE, 2004. 520−525. [doi: 10.1109/EURMIC.2004.1333416]

[4] Petroni NL Hicks M. Automated detection of persistent kernel control-flow attacks. In: Proc. of the ACM Conf. on Computer and Communications Security (CCS). New York: ACM, 2007. 103−115. [doi: 10.1145/1315245.1315260]

[5] Joshi A, King ST, Dunlap GW, Chen PM. Detecting past and present intrusions through vulnerability-specific predicates. In: Proc. of the ACM Symp. on Operating Systems Principles (SOSP). New York: ACM, 2005. 91−104. [doi: 10.1145/1095810.1095820]

[6] Jiang X, Wang X, Xu D. Stealthy malware detection and monitoring through VMM-based out-of-the-box semantic view reconstruction. In: Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS). New York: ACM, 2007. 128−138. [doi: 10.1145/1315245.1315262]

[7] Payne BD, Carbone MDPDA, Lee W. Secure and flexible monitoring of virtual machines. In: Proc. of the Annual Computer Security Applications Conf. (ACSAC). Piscataway: IEEE, 2007. 385−397. [doi: 10.1109/ACSAC.2007.10]

[8] Payne BD, Carbone M, Sharif M, Lee W. Lares: An architecture for secure active monitoring using virtualization. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland). Piscataway: IEEE, 2008. 233−247. [doi: 10.1109/SP.2008.24]

[9] Sharif MI, Lee W, Cui W, Lanzi A. Secure in-VM monitoring using hardware virtualization. In: Proc. of the 16th ACM Conf. on Computer and Communications Security (CCS). New York: ACM, 2009. 477−487. [doi: 10.1145/1653662.1653720]

[10] Shi B, Cui L, Li B, Liu XD, Hao ZY, Shen HY. ShadowMonitor: An effective in-VM monitoring framework with hardware-enforced isolation. In: Proc. of the Int'l Symp. on Recent Advances in Intrusion Detection (RAID). Berlin: Springer, 2018. 670–690. [doi: 10.1007/978-3-030-00470-5_31]

[11] Xiong X, Liu P. SILVER: Fine-grained and transparent protection domain primitives in commodity OS kernel. In: Proc. of the Int'l Symp. on Recent Advances in Intrusion Detection (RAID). Heidelberg, Berlin: Springer-Verlag, 2013. 103−122. [doi: 10.1007/978-3-642-41284-4_6]

[12] Dautenhahn N, Kasampalis T, Dietz W, *et al*. Nested kernel: An operating system architecture for intra-kernel privilege separation. In: Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM, 2015. 191−206. [doi: 10.1145/2694344.2694386]

[13] Criswell J, Dautenhahn N, Adve V. Virtual ghost: Protecting applications from hostile operating system. In: Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM, 2014. 81–96. [doi: 10.1145/2541940.2541986]

[14] Swift MM, Bershad BN, Levy HM. Improving the reliability of commodity operating systems. In: Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP). New York: ACM, 2003. 207–222. [doi: 10.1145/945445.945466]

[15] Erlingsson U, Abadi M, Vrable M, Budiu M, Necula GC. XFI: Software guards for system address spaces. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2006. 75−88. [doi: 10.5555/1298455.1298463]

[16] Deng L, Zeng QK, Wang WG, Liu Y. EqualVisor: Providing memory protection in an untrusted commodity hypervisor. In: Proc. of the 13th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom). Piscataway: IEEE, 2014. 300−309. [doi: 10.1109/TrustCom.2014.41]

[17] Deng L, Liu P, Xu J, Chen P, Zeng QK. Dancing with wolves: Towards practical event-driven VMM monitoring. In: Proc. of the 13th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments (VEE). New York: ACM, 2017. 83−96. [doi: 10.1145/3050748.3050750]

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corp., 2019.

[19] Deng L, Zeng QK. Inner TCB based application protection. Ruan Jian Xue Bao/Journal of Software, 2016, 27(4): 1042−1058 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5016.htm [doi: 10.13328/j.cnki.jos.005016]

[20] McCune JM, Parno B, Perrig A, Reiter MK, Isozaki H. Flicker: An execution infrastructure for TCB minimization. In: Proc. of the ACM European Conf. in Computer Systems (EuroSys). New York: ACM, 2008. 315−328. [doi: 10.1145/1352592.1352625]

[21] Zhang F, Leach K, Sun K, Starvrou A. Spectre: A dependable introspection framework via system management mode. In: Proc. of the 43rd Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). Piscataway, 2013. 1−12. [doi: 10.1109/DSN.2013.6575343]

[22] Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with haven. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2014. 267−283. [doi: 10.5555/2685048.2685070]

[23] Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G, Ma J, Shen W. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security (CCS). New York: ACM, 2014. 90−102. [doi: 10.1145/2660267.2660350]

[24] Wang X, Chen Y, Wang Z, Qi Y, Zhou Y. SecPod: A framework for virtualization-based security systems. In: Proc. of the USENIX Conf. on Usenix Annual Technical Conf. (ATC). Berkeley: USENIX Association, 2015. 347−360. https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang

[25] Liu YT, Zhou TY, Chen KX, Chen HB, Xia XB. Thwarting memory disclouse with efficient hypervisor-enforced intra-domain isolation. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS). New York: ACM, 2015. 1607−1619. [doi: 10.1145/2810103.2813690]

[26] Intel. Intel Trusted Execution Technology Preliminary Architecture Specification. 2006.

[27] Szefer J, Keller E, Lee RB, Rexford J. Eliminating the hypervisor attack surface for a more secure cloud. In: Proc. of the 18th ACM Conf. on Computer and Communications Security (CCS). New York: ACM, 2011. 401−412. [doi: 10.1145/2046707.2046754]

[28] Koning K, Chen X, Bos H, Giuffrida C, Athanasopoulos E. No need to hide: Protecting safe regions on commodity hardware. In: Proc. of the 12th European Conf. on Computer Systems (EuroSys). New York: ACM Press, 2017. 437−452. [doi: 10.1145/3064176.3064217]

[29] Ionescu A. Owning the image object file format, the compiler toolchain, and the operating system: Solving intractable performance problems through vertical engineering. 2016. http://www.alex-ionescu.com/?p=323

[30] Davi L, Gens D, Liebchen C, Sadeghi AR. PT-rand: Practical mitigation of data-only attacks against page tables. In. Proc. of the 24th Annual Network and Distributed System Security Symp. (NDSS). Rosten: Internet Society, 2017. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/pt-rand-practical-mitigation-data-only-attacks-against-page-tables/

[31] PaX Team. RAP: RIP ROP. 2015. https://pax.grsecurity.net/docs/PaXTeamH2HC15-RAP-RIP-ROP.pdf

[32] Chen Q, Azab AM, Ganesh G, Ning P. PrivWatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In. Proc. of the 2017 ACM on Asia Conf. on Computer and Communications Security (AsiaCCS) New York: ACM, 2017. 167−178. [doi: 10.1145/3052973.3053029]

[33] Li WH, Xia YB, Lu L, Chen HB, Zang BY. TEEv: Virtualizing trusted execution environments on mobile platforms. In. Proc. of the 15th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments (VEE). New York: ACM, 2019. 2−16. [doi: 10.1145/3313808.3313810]

[34] Deng L. Study on system security techniques in an untrusted kernel environment [Ph. D. Thesis]. Nanjing: Nanjing University, 2018 (in Chinese with English abstract).

[35] Hua ZC, Gu JY, Xia YB, Chen HB, Zang BY, Guan HB. vTZ: Virtualizing ARM TrustZone. In: Proc. of the 26th USENIX Security Symp. (USENIX Security). Berkeley: USENIX Association, 2017. 541−556. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua

[36] Proskurin S, Momeu M, Ghavamnia S, Kemerlis VP, Polychronakis M. xMP: Selective memory protection for kernel and user space. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). Piscataway: IEEE, 2020. 233−247. [doi: 10.1109/SP40000.2020.00041]

[37] Huang X, Deng L, Sun H, Zeng QK. Secure and efficient kernel monitoring model based on hardware virtualization. Ruan Jian Xue Bao/Journal of Software, 2016, 27(2): 481−494 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4866.htm [doi: 10.13328/j.cnki.jos.004866]

附中文参考文献:

[19] 邓良, 曾庆凯. 引入内可信基的应用程序保护方法. 软件学报, 2016, 27(4): 1042−1058. http://www.jos.org.cn/1000-9825/5016.htm [doi: 10.13328/j.cnki.jos.005016]

[34] 邓良. 不可信内核环境下的系统安全技术研究[博士学位论文]. 南京: 南京大学, 2016.

[37] 黄啸, 邓良, 孙浩, 曾庆凯. 基于硬件虚拟化的安全高效内核监控模型. 软件学报, 2016, 27(2): 481–494. http://www.jos.org.cn/1000-9825/4866.htm [doi: 10.13328/j.cnki.jos.004866]

附 录

- 关键攻击操作

根据第 5.1.2 节对于攻击行为的分析, 关键攻击操作包括写内存、加载 IDTR、修改控制寄存器、修改 MSR 寄存器以及跳转地址等. 关键攻击操作的伪代码如下.

① 写内存操作.

```
void pkd_write(void*destin, void*source, size_t num)
{
  long d0, d1, d2;
  asm volatile("rep; movsq\n\t"
  "movq%4, %%rcx\n\t"
  "rep; movsb\n\t"
  : "=&c"(d0), "=&D"(d1), "=&S"(d2): "0"(num>>3), "g"(n&7), "1"(destin), "2"(source)
  : "memory");
}
```

其中, *destin* 是目标地址空间, 而 *source* 是指向写入目标地址空间数据的源地址, *num* 则是写入数据的大小.

② 加载 IDTR 操作

```
void pkd_loadidt(struct desc_ptr*idtr)
{asm volatile("lidt%0\n\t": : "m"(*dtr)); }
```

函数 *pkd_loadidt* 中, 参数 *idtr* 指向新的 IDT.

③ 修改寄存器(CR0 或 CR4)操作

```
void pkd_set_cr0(unsigned long mask)
{
  unsigned long cr0;
  asm volatile("mov%%cr0, %0\n\t": "=r"(cr0): : );
  cr0=cr0|mask;
  asm volatile("mov%0, %%cr0\n\t": : "r"(cr0): );
}
void pkd_clear_cr0(unsigned long mask)
{
  unsigned long cr0;
  asm volatile("mov%%cr0, %0\n\t": "=r"(cr0): : );
  cr0=cr0 & (~mask);
  asm volatile("mov%0, %%cr0\n\t": : "r"(cr0): );
}
void pkd_set_cr4(unsigned long mask)
{
  unsigned long cr4;
  asm volatile("mov%%cr4, %0\n\t": "=r"(cr4): : );
```

```
    cr4=cr4|mask;
    asm volatile("mov%0, %%cr4\n\t": : "r"(cr4): );
}
void pkd_clear_cr4(unsigned long mask)
{
    unsigned long cr4;
    asm volatile("mov%%cr4, %0\n\t": "=r"(cr4): : );
    cr4=cr4 & (~mask);
    asm volatile("mov%0, %%cr4\n\t": : "r"(cr4): );
}
```

函数 *pkd_set_cr*0 和 *pkd_set_cr*4 将 CR0/CR4 中相应的位置 1, 而 *pkd_clear_cr*0 和 *pkd_clear_cr*4 则将 CR0/CR4 中相应的位清零. *Mask* 标记 CR0/CR4 寄存器中相应的位, 其可以取值包括 X86_CR4_SMEP、X86_CR0_ PE、X86_CR0_PG 等值(X86_CR4_SMEP 等用来标记 CR0/CR4 上不同的位).

④ 修改 MSR 寄存器操作

```
void pkd_write_msr(unsigned int msr, u64 value)
{asm volatile("wrmsr": : "c"(msr), "a"((u32)value), "d"((u32)(value>>32)): ); }
unsigned long long pkd_read_msr(unsigned int msr)
{
    u64 val;
    u32 low, high;
    asm volatile("rdmsr": "=a"(low), "=d"(high): "c"(msr): );
    value=(u64)(low|((high)<<32));
    return value;
}
```

MSR 是不同 MSR 的地址, 其中, MSR_EFER 的地址为 0xc0000080, #define_EFER_NX 用来标记 MSR_EFER 中的 NX 位.

⑤ 跳转到其他地址空间执行操作

```
void call_func(void*address)
{asm volatile("jmpl %0": : "r"(*address): ); }
```

- 攻击实例的 PoC

攻击实例 1(直接访问内存攻击). 其 PoC 模拟攻击行为的关键伪代码是:

```
void test_direct_mem_access_attack(·)
{
    char*local="test";   /*用于测试写入内存的数据*/
    char*dest;
    dest=KPD2_add;   /*KPD2_add 地址是功能保护域 2 地址空间内的地址, 是写内存的目的地址*/
    pkd_write(dest, local, 4);
}
```

其中, KPD2_add 是功能保护域 2 地址空间内任意地址, 因为保护域都是在初始化过程中预分配的, 其地址空间布局都是已知的.

攻击实例 2(修改页表攻击). 其 PoC 模拟攻击行为的关键伪代码是:

```
void test_modify_pagetable_attack(·)
```

```
{
  unsigned long addr=0xffffffffffff0001;
  unsigned long*local;   /*用于被修改的页表项的地址*/
  unsigned long*dest;
  local=&addr;
  dest=KPD2_pt;   /* KPD2_pt 地址是功能保护域 2 内页表的地址, 用于指向页表项的地址*/
  pkd_write(dest, local, 8);
}
```

其中, KPD2_pt 是指向功能保护域 2(PartKPD2)内 Guest 层页表项的地址.

攻击实例 3(DMA 攻击). 在 DMA 攻击实例中, *dma_address* 指向内核中 DMA 操作的代码. 其伪代码如下:

```
Void test_DMA_attack(·)
{
  void*address;
  address=dma_address;
  call_func(address);
}
```

攻击实例 4(重映射 IDT 攻击). 在 IDT 重映射攻击的攻击实例的 PoC 主要包括两部分: 一是利用内存的写操作实现 IDT 表修改和页表的修改, 二是利用 lidt 指令加载新的 IDT. 从攻击实例 1 和攻击实例 2 可以看出, 在 Decentralized-KPD 中, 可以抵御攻击者对 IDT 和页表的攻击. 因此, 在测试 IDT 重映射攻击时, 主要考虑 lidt 指令执行的攻击, 主要验证初始化之后, lidt 指令是否能够有效执行, 其 PoC 伪代码如下:

```
void test_idt_redirect_attack(int n)
{
  struct desc_ptr*new_idtr;
  new_idtr=NEW_IDT;   /*NEW_IDT 是攻击者构造的一个新的 IDT 表*/
  pkd_loadidt(new_idtr);
}
```

攻击实例 5(修改硬件上下文攻击). 在硬件上下文攻击实例的 PoC, 本文的测试实验验证攻击能否修改 CR0、CR4、IA32_EFER 等寄存器的值.

```
void test_hardware_context_attack(int n)
{
  if (n==1)
  {pkd_clear_cr0(X86_CR0_PE); }
  if (n==2)
  {pkd_clear_cr4(X86_CR4_SMEP); }
  if (n==3)
  {u64 value;
  value=pkd_read(MSR_EFER);
  value=value &(~(1<<_EFER_NX));
  pkd_msr(MSR_EFER, value);
  }
}
```

其中, MSR_EFER 的地址为 0xc0000080, #define_EFER_NX 11 用来标记 MSR_EFER 中的 NX 位. 安全测

试实验需要多次调用 *test_hardware_context_atttack*(·)函数, 每次只需要输入不同的参数即可. 同时, 对于寄存器的不同位, 只需更改 *pkd_write_cr*0/4(·)或者 *pkd_msr*(·)函数的参数即可.

攻击实例 6(恶意执行 VMFUNC 指令攻击). 恶意 VMFUNC 指令的关键操作是攻击者执行非法的 VMFUNC 指令, 攻击者只能通过执行用户层或者加载模块中可能存在的 VMFUNC 指令实现攻击. 因此, 攻击实例 6 的 PoC 的关键伪代码如下:

```
void test_vmfunc_attack(·)
{
   void*addr;
   addr=User_address;   /*User_address 是用户空间地址, 验证攻击者能否将控制流转向用户空间*/
   call_func(addr);
}
```

钟炳南(1991－), 男, 博士生, 主要研究领域为信息安全, 操作系统.

邓良(1987－), 男, 博士, 工程师, 主要研究领域为虚拟化.

曾庆凯(1963－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为信息安全, 分布计算.