安全虚拟机监视器的形式化验证研究

陈 昊 罗 蕾 李 允 陈丽蓉
(电子科技大学计算机科学与工程学院 成都 611731)

摘 要 虚拟化技术为安全关键系统提供了分区隔离等重要特性,虚拟机监视器(Virtual Machine Monitor,VMM)作为其核心组件,对客户系统的安全运行及虚拟机间威胁和故障的屏蔽起着决定性作用。文中从最小特权原则出发,将VMM的设计按是否与安全直接相关划分为内核扩展与用户进程,并采用分层精化的方法对内核扩展中的各关键模块展开了形式化建模与验证,继而以此为基础,证明了虚拟机实现的功能正确性。实验评估表明,原型系统的综合性能负载与主流虚拟化方案接近,安全划分的设计方法与形式化验证在提升VMM安全性的同时,并未对其产生明显负载,可较好地满足应用领域的需求。

关键词 虚拟化技术,虚拟机监视器,安全隔离,形式化验证,功能正确性

中图法分类号 TP316.2 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.03.026

Study on Formal Verification of Secure Virtual Machine Monitor

CHEN Hao LUO Lei LI Yun CHEN Li-rong
(School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China)

Abstract Virtualization equips the security-critical systems with multiple features, including partitioning and separation. As the core component, virtual machine monitor (VMM) serves as a backbone to the secure execution as well as a barrier to isolate the threats and faults of virtual machines. Following the principle of least privilege, this paper presented a method to decouple the VMM into two parts: kernel extension and user processes. Furthermore, a formal method by constructing abstraction layers is used to certify those key components of the VMM kernel extension. Then, the functional correctness property of the VMM are also proved. The experiment results show that the certified prototype achieves comparable efficiency as the mainstream virtualization solution. The decoupled design and formal verification improve the VMM security without imposing obvious performance degradation, and meet the requirement of the application fields.

Keywords Virtualization technology, Virtual machine monitor, Security isolation, Formal verification, Functional correctness

1 引言

虚拟化是一种通过软硬件手段重组和共享计算资源的技术[1]。在面向安全的控制领域中,虚拟化常用于安全域间的隔离[2],即通过虚拟机对物理系统的时间、空间和I/O设备等资源进行划分,实现分区间的故障和威胁屏蔽,以确保即使部分虚拟机遭受入侵,攻击者也无法影响系统中其他分区的安全运行。

虚拟机监视器是实现虚拟化的核心组件,它控制虚拟机的分配及系统资源的访问。在为系统提供分区与隔离的同时,虚拟化技术也引入了新的安全风险。VMM是否能实现既定的功能,并满足所需的安全属性,成为了一项重要的科研课题。形式化方法是目前已知的可以解决该信任假设的最可靠途径[3],它通过严格的数学建模和推导证明了软硬件算法的正确性,从而将软件是否正确实现这一问题从可信计算基中剔除。在ISO/IEC 15408信息安全通用准则中,具备形式化设计和验证的系统被视为拥有最高评估保障等级[4]。VMM的特殊性,使得其形式化验证面临着以下挑战。

(1)VMM代码规模通常比较庞大,难以对底层实现进行功能正确性验证[5]。经验统计表明,用于机器检查的形式化证明代码量与软件规模呈二次关系[6]。以Xen为例,其核心代码规模为270kloc[7],且核心代码不包括Dom 0中的虚拟机管理终端和驱动模块;又例如,KVM虽然依赖于Linux内核服务,但其代码量超过了33kloc[8];此外,即便是采用半虚

到稿日期:2018-01-03 返修日期:2018-03-20 本文受"十二五"核高基重大专项:汽车电子基础软件平台研发及应用产业化项目(2009ZX01038-002-003)资助。

陈 昊(1988-),男,博士生,主要研究方向为计算机系统结构与安全;罗 蕾(1967-),女,硕士,教授,主要研究方向为嵌入式系统与安全,E-mail:lluo@uestc.edu.cn(通信作者);李 允(1971-),男,博士,研究员,主要研究方向为嵌入式系统与网络安全;陈丽蓉(1972-),女,硕士,高级工程师,主要研究方向为嵌入式系统及软件、软件形式化验证。

拟化,将大部分 VMM 功能移入客户操作系统内的 XtratuM 内核,也包含约 10kloc[9] 的 C 代码实现。鉴于此,大量针对 VMM 形式化验证的研究,或者仅针对实现的抽象模型[5,10-11]的研究,难以确保抽象模型与代码实现相匹配;或者仅证明系统部分安全属性[12-15]正确,无法刻画具体实现与规约之间完整的功能正确性。

(2)已有的工作,如 VerisoftXT 项目[16]、BHV(BabyHyperVisor)[17]、L4.verified[18]、XtratuM[9]、XMHF[19]、Prosper 内核[12]等一系列课题[20-23],采用半虚拟化 VMM 的设计架构,针对其中诸如影子页表[20-22]、页面缓存[21]、虚拟 TLB[23]等半虚拟化关键算法进行了建模和验证。其优势是对硬件平台的依赖性低,且 VMM 可完全运行于非特权级的用户进程中,但只能支持专用客户操作系统的执行[24]。近年来,硬件辅助虚拟化逐步受到主流处理器平台的支持,芯片的不可篡改性和相较于软件更高的可靠性,使其成为了实现高效 VMM 的最安全方案[25]。但是,目前尚无研究对此特性形成底层形式的语义规约,这导致难以推导和验证含有硬件辅助虚拟化指令的程序。

(3)VMM 的代码通常由 C 和汇编实现,并分为内存管理、调度、事件处理等多个模块,形式化验证需要在两个语言层面和多个模块间分别进行,但如果所验证的属性无法正确链接,则得出的定理有可能存在内部矛盾,并推导出错误的结论[26]。因此,需要找到一种适合的工具,将 C 和汇编的证明统一到同一语义模型中,并与其他模块的证明组合。

针对上述问题,本文提出了一种可用于形式化验证的 VMM 设计方法,并给出其验证过程。该方法主要针对控制领域中安全域隔离的需求,以及最少特权原则的设计思想,在充分利用硬件辅助虚拟化的同时,采用 I 型和 II 型 VMM 混合架构,即将 VMM 分为安全相关的内核扩展(VMM-SC)及与任务相关的用户进程(VMM-RT),在不影响系统安全性的前提下,精简特权空间的代码量;同时,使用文献[27]提出的分层精化形式化验证方法,通过证明代码实现与其规约之间在不同抽象机层面上的前推模拟关系,来推导出 VMM 内核扩展的功能正确性,最终证明系统的安全属性满足。本文工作具有以下特点:

(1)对基于虚拟化硬件所实现的 VMM 进行功能划分和接口设计。以面向控制领域应用的安全要求为边界,保证 VMM 内核扩展在调用者不可信的条件下依然能满足自身的安全性和虚拟机之间的隔离性,并尽量降低划分所带来的性能负载,以确保 VMM 可用。

(2)验证了 VMM 内核扩展 C 和汇编代码的功能正确性。所验证的系统经过交互式定理证明器 Coq[28] 的检查,直接抽取为可执行汇编程序。

(3)由于所验证的功能正确性满足上下文相关的精化关系,因此规约可与操作系统内核的证明相互链接,形成没有语义间隙的完整定理。此外,规约捕捉 VMM 的完整行为,因此可在后续验证中重复使用。

2　设计动机与系统架构

出于安全考虑,无法将硬件虚拟化功能直接暴露于不可信的宿主用户空间,因为对这些关键指令和数据的操作会导致虚拟机保护机制的失效。因此,基于硬件辅助虚拟化实现的 VMM 通常运行于最高的特权级,如图 1 所示。客户系统通过 *vm entry/exit* 与 VMM 形成交互界面:当处理器在客户模式中执行到可能影响虚拟机外部资源或数据的敏感指令,或接收到异常、外部中断和 *Hypercall* 等请求时,通过 *vm exit* 操作切换到宿主模式,进而由 VMM 协助并完成处理。相应地,由 VMM 所发起的 *vm entry* 操作可切换回客户系统并继续执行先前的操作。

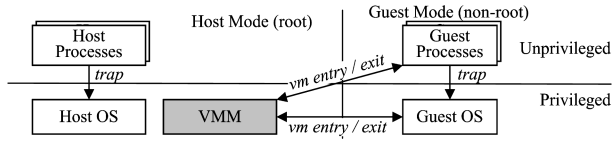


图 1　硬件辅助虚拟化中系统执行空间的划分

Fig. 1　System operation mode partition in hardware-assisted virtualization

由于 VMM 的复杂性,代码规模通常较大,若采用图 1 所示的设计架构,不仅会增加特权空间的安全风险,还提高了形式化验证的难度。而在面向安全的控制系统中,保证虚拟机执行的安全性和虚拟机间的隔离性,比确保虚拟化正确性更为重要[29]。因此,控制系统允许某些虚拟机是不可信的,客户系统崩溃后可通过重启虚拟机进行恢复,但如果恶意代码渗透到 VMM 或其他虚拟机,则会对控制器造成更为严重的影响。

鉴于此,如果将 VMM 中保障安全和隔离的最关键部分抽离,则可将其划分为两个部分:1)与虚拟机安全运行直接相关的 VMM 核心——VMM-SC(Security Critical);2)在 VMM-SC 上搭建实现虚拟机功能的 VMM 运行时——VMM-RT (Run-Time)。VMM-RT、VMM-SC 及客户系统控制界面如图 2 所示。

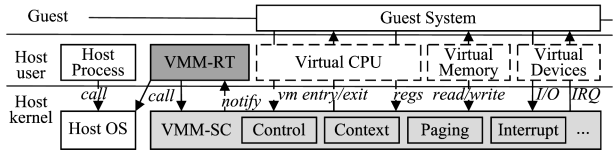


图 2　VMM-RT、VMM-SC 及客户系统控制界面

Fig. 2　Control interfaces between VMM-RT, VMM-SC and guest systems

上述划分使得原本完全运行于宿主内核态中的 VMM 的绝大部分移入到用户进程中,而不必保证该部分安全可信,但这也带来了如下挑战。

(1)攻击面下移到了 VMM 内部。在保持通过 *vm entry/exit* 与客户系统进行控制流转移的同时,VMM-SC 还需要为 VMM-RT 提供安全的访问接口(见图 2),以保证即使恶意的客户系统控制了 VMM-RT,也依然无法入侵宿主和其他虚拟机。

(2)VMM 控制逻辑上移到了用户进程中。某些 *vm exit* 操作可能会由于受到调度、中断的影响,而无法提供正确的服务。例如,在 VMM-RT 进程中不允许通过关闭中断的方式进入临界区,否则 VMM 将不具备对外部事件的接收和夺回处理器控制权的能力。

(3)VMM-RT 和客户系统需要共享内存。虚拟机可访问的物理内存空间受 VMM 保护,一方面 VMM 需要保证其大

小和映射范围不能与其他虚拟机或应用重叠，另一方面 VMM-RT 在处理 *vm exit* 的过程中需要访问部分物理页面。此外，时间可能成为虚拟机信息泄漏的一个隐藏通道[30]，恶意客户系统可能通过测量 *vm exit* 的执行时间来推断宿主系统或其他虚拟机的运行情况。

(4)VMM 的划分除了需要考虑 VMM-SC 是否能保护系统安全和隔离外，还要尽量减少调用接口的数量与频率，以缩小攻击面，提升系统的执行效率。

综合上述考虑，本文 VMM 的总体架构如图 3 所示。其中，VMM-SC 软件模块丰富并扩充了硬件的虚拟化能力，同时对调用接口和操作方法进行了抽象和优化。它将对虚拟机关键数据和相关指令的访问封装为系统调用，进而对所有访问进行安全检查。VMM-SC 由以下模块组成：1)虚拟 CPU，控制对 VMCS 结构和 CPU 寄存器的访问，同时提供在宿主模式和客户模式间进行安全切换的关键代码；2)虚拟中断管理模块，管理并响应虚拟机的外部中断，并将其转交给虚拟机；3)虚拟时钟，屏蔽或改写虚拟机中可观察到的时间；4)虚拟机内存管理，对虚拟内存空间进行分区，并在虚拟机内存缺页时分配新的物理页面；5)生命期管理模块，控制虚拟机的启动和停止，根据虚拟机的运行状态，切换当前宿主操作系统对系统调用、中断响应等模块的行为。

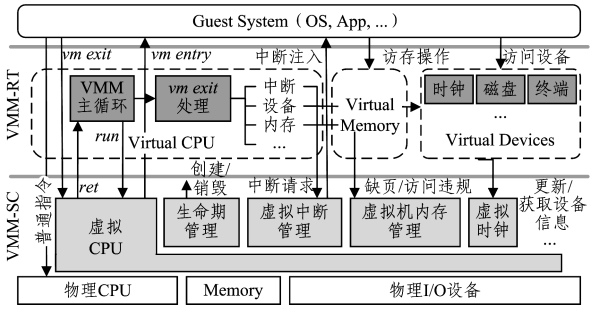


图 3 VMM 总体设计架构、模块组成及依赖关系

Fig. 3 Architecture, composition and dependence of components in VMM

VMM 的执行逻辑 VMM-RT 被封装为进程，运行在宿主用户态。其主要包括：1)初始化模块，主要功能包括设置虚拟机内存结构和虚拟 CPU 寄存器，加载虚拟 BIOS 代码，初始化虚拟终端和虚拟设备；2)VMM 主循环，通过 VMM-SC 进入/退出客户模式，在 *vm exit* 时获取并解析相关信息，并将其派发到相应服务例程，另外还负责刷新虚拟设备并更新虚拟时间；3)虚拟终端，用于在多个虚拟机之间切换与用户交互的前端界面，并为处于后端的交互程序进行数据缓冲；4)*vm exit* 处理，按照不同 *vm exit* 退出的原因和相关数据，处理需要被模拟的操作；5)虚拟设备，包括虚拟 PCI、定时器、中断控制器、I/O 设备等，用于在客户系统对设备进行访问时模拟真实硬件的行为；6)虚拟机内存，提供客户系统可以访问的内存空间和数据，为了提高系统的运行效率，我们使用 virtio 接口[31]模拟部分设备。

本文中 VMM 的实现包含约 7.7 kloc 的 C 和汇编代码，其中 VMM-RT 占 6.4 kloc，为总代码量的 83%。虽然 VMM-RT 的运行还依赖于宿主操作系统提供的进程调度、内存管理等机制，但安全划分的设计有效减小了可信计算基的规模。

3 VMM 的形式建模

软件代码的功能正确性的验证往往通过建立与规约的精化关系来完成[32]。如图 4(a)所示，精化证明通过一个更为抽象的规约 σ 来刻画与实现$[\![\kappa]\!]$相同的行为，从而揭示程序所有可能的行为，使其满足设计要求。

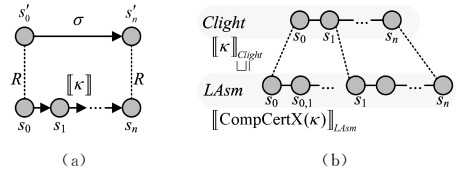


图 4 数据精化证明以及 CompCertX 的编译正确性

Fig. 4 Contextual refinement proof and compiler correctness of CompCertX

VMM 的实现，即 C 和汇编代码，以 *Clight* 和 *LAsm* 的形式引入，其中 *Clight* 是 CompCert 编译器前端[33]所定义的 C 语言的子集，它通过简化 C 语言的部分能力，在避免难以建模语义的同时，支持指针运算等系统软件的常用特性，具备较好的表达能力。*LAsm* 是 CompCert x86 汇编模型[33]的超集，对 x86 硬件平台中 VMM 需要使用的寄存器、内存和相应的汇编指令进行了建模，是最底层的机器模型。

框架内置的 CompCertX 编译器将已证明的 *Clight* 程序编译为 *LAsm* 程序，并生成二者的精化证明(见图 4(b))，因此不需要信任编译过程的正确性，也可保证规约与编译结果的一致性。

基于抽象层[27]的验证框架如图 5 所示，在数据精化的基础上，将软件实现分解为多个子模块 $M_1, M_2, \cdots$，并要求每一个子模块的规约 L 可以作为验证下一个模块的抽象机器模型。这使得规约必须满足上下文相关的精化属性：

$$\forall P.[\![P]\!]L_2 \leqslant_R [\![P \oplus M_1]\!]L_1 \tag{1}$$

即对于任意上下文程序 P，其在模块 M_1 所构成的规约 L_2 上执行的行为与其链接 M_1 后在 L_1 上的行为一致。后文将式(1)简化为：$L_2 \sqsubseteq [\![M]\!]L_1$。

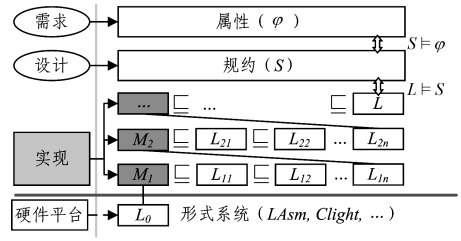


图 5 基于抽象层的软件验证

Fig. 5 Software verification based on abstraction layers

验证工作首先搭建 VMM 的最底抽象层，然后逐步引入代码实现，构建新的规约，当全部抽象层的验证完成后，将其插入到原有操作系统的证明中，获得最顶抽象层(L_{Tsyscall})与最底层(L_{Mboot})间的上下文精化证明。即：

$$L_{\text{Tsyscall}} \sqsubseteq [\![M_{\text{VMM-SC}} \oplus M_{\text{kernel}}]\!]L_{\text{Mboot}} \tag{2}$$

接下来，通过 CompCertX 编译器生成 $M_{\text{VMM-SC}}$ 中 *Clight* 代码所对应的 *LAsm* 汇编。最终使用汇编器和链接器，把生成的汇编代码与用户态 VMM-RT 连同其他用户进程打包，生成可执行 VMM 镜像。

3.1　抽象层的形式定义

抽象层被定义为一个三元组(L_1,M,L_2),其中 L 称为抽象层接口,定义为:

$$L::=\emptyset\mid i\mapsto\sigma\mid i\mapsto v\mid L_1\oplus L_2$$

即 L 是一个命名的有限映射$(i\mapsto_)$,它的成员包含原语规约(σ)与数据(v),且与唯一的标识符 i 绑定。M 是抽象层所引入的代码实现,定义为:

$$M::=\emptyset\mid i\mapsto\kappa\mid i\mapsto v\mid M_1\oplus M_2$$

其中,κ 表示一段 *Clight* 或 *LAsm* 程序。(L_1,M,L_2)成立的条件是当且仅当证明了模块 M 的所有函数均符合其规约,即 $L_2\sqsubseteq[\![M]\!]L_1$。

原语规约 σ 是$(val^*\times m\times A)\times(val\times m\times A)$上的谓词。$\sigma(args,m,a,rv,m',a')$成立,当且仅当原语以 $args$ 为参数、m 和 a 分别为初始内存和抽象状态时,返回结果 rv,并以 m'和 a'作为终止状态。

当 κ 为 *Clight* 函数时,它被定义为一个三元组$(targs,lvars,S)$,其中 $targs$ 为函数运行时接收参数的临时变量,$lvars$ 为栈上分配的局部变量,S 为 C 语句。当 κ 为 *LAsm* 程序时,它被定义为汇编指令(I)的序列。二者语义的定义分别如式(3)和式(4)所示:

$$\frac{f\mapsto\kappa\in M\quad\Gamma,L,M\vdash f:(args;m,a)\Downarrow(rv;m',a')}{[\![\kappa]\!](args,m,a,rv,m',a')}\tag{3}$$

$$\frac{i\mapsto\kappa\in M\quad\Gamma(i)=b\quad M[b][0]=\kappa\quad\Gamma,L,M\vdash(\rho,m,a)\to^{\kappa}(\rho',m',a')}{[\![\kappa]\!](\rho,m,a,\rho',m',a')}\tag{4}$$

其中,$\Downarrow$ 为 *Clight* 函数的大步操作语义[34],$\to^{\kappa}$ 为 *LAsm* 对于程序 κ 的小步语义[27]的多步执行,ρ 为机器寄存器组。

3.2　扩展页表

扩展页表(Extended Page Table,EPT)用于管理客户系统的内存空间映射。如图 6 所示,对于开启分页机制的客户系统而言,当其访问内存中的数据时,硬件虚拟化 MMU 会通过客户页目录指针($gCR3$)、页表(gPG)和扩展页表,一次性地将客户虚拟地址(GVA)转换为宿主物理地址(HPA)。EPT 在内存中为一个由四级页表组成的树形结构(见图 6),其中每一级表项包含了下一级页表的首地址、访问权限和内存属性。通过 GPA 的分段索引,可以从 EPT 结构中找到该地址所对应的物理页面。

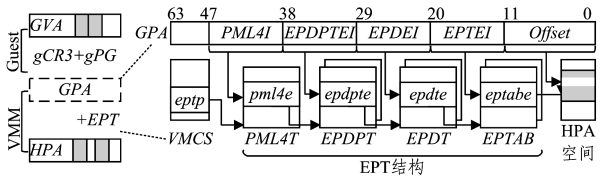


图 6　EPT 结构及地址空间转换

Fig. 6　EPT structure and space address translation

EPT 被建模为:

$$EPTPool::=\mathrm{ZMap.t}\ EPML4T$$

$$EPML4T::=\mathrm{ZMap.}t(\mathrm{EValid}(e;EPDPT)\mid\mathrm{EUndef})$$

$$EPDPT::=\mathrm{ZMap.}t(\mathrm{EValid}(e;EPDPT)\mid\mathrm{EUndef})$$

$$EPDPT::=\mathrm{ZMap.}t(\mathrm{EValid}(e;EPTAB)\mid\mathrm{EUndef})$$

$$EPDAB::=\mathrm{ZMap.}t(\mathrm{EValid}(hpg:\mathbb{Z},p:Eperm)\mid\mathrm{EUndef})$$

其中,ZMap 用于表示整数($\mathbb{Z}$)到数据 $x\in T$ 的偏射,使用操作符 $map[key\mapsto value]$和 $map[key]$从其中存/取数据。通过 $id:\mathbb{Z}$可以从 $EPTPool$ 中获得对应虚拟机的 EPT 结构,然后逐层解析各级表项,最终获得 HPA 所映射的物理页帧($hpg\in\mathbb{Z}$)及其访问权限($p\in Eperm$)。$EUndef$ 用于表示还未映射的表项。

对 EPT 的底层操作包括:各级表项的设置、客户系统的物理内存访问原语和对页面缓存的刷新操作。

3.3　虚拟机控制结构

虚拟机控制结构(VMCS)决定硬件的虚拟化行为,它被建模为:

$$VMCSPool::=\mathrm{ZMap.}t\ VMCS$$

$$VMCS::=\mathrm{VMCSValid}(revid:\mathbb{Z},abrtid:\mathbb{Z},data:\mathrm{ZMap.}t\ val)$$

其中,$revid$ 为结构版本号,需与处理器硬件虚拟化版本相兼容;$abrtid$ 存储 *vm exit* 错误时的终止码;$data$ 是 VMCS 的主体,包含虚拟机运行过程中 *vm entry/exit* 操作相关的数据和控制信息,以及暂存的宿主和客户状态。

由于虚拟化硬件实现对于 VMCS 存在组织结构差异,为了避免直接使用内存寻址所造成的移植性问题,VMCS 字段的访问需要使用特殊指令:*vmread/vmwrite*。其形式规约如式(5)和式(6)所示:

$$\frac{a.in\_kern=\mathrm{true}\quad a.pg=\mathrm{true}\quad a.in\_host=\mathrm{true}\quad d=a.vmcspool[\mathrm{vmid}()].data\quad None\neq vmcs\_Z\_to\_encoding(enc)\quad \mathrm{Vint}\ v=data[enc]}{\mathrm{Some}\ v=vmread(enc,a)}\tag{5}$$

$$\frac{a.in\_kern=\mathrm{true}\quad a.pg=\mathrm{true}\quad a.in\_host=\mathrm{true}\quad None\neq vmcs\_Z\_to\_encoding(enc)\quad vmcs=a.vmcspool[\mathrm{vmid}()]\quad vmcs'=vmcs[data\leftarrow vmcs.data[enc\mapsto v]]\quad a'=a[vmcspool\leftarrow a.vmcspool[\mathrm{vmid}()]\mapsto vmcs']]}{\mathrm{Some}\ a'=vmwrite(enc,v,a)}\tag{6}$$

其中,$a.in\_kern$,$a.pg$ 和 $a.in\_host$ 分别表示原语执行上下文的系统特权级、分页状态和宿主态。此外,机器原语 *vmptrld* 用于切换当前处理器的 VMCS 结构。

3.4　VMM-SC 运行时状态

除 VMCS 外,VMM-SC 的运行还依赖于诸如客户通用寄存器、虚拟机运行状态等额外信息,VMExtra 结构用于管理这些数据。其抽象类型的形式化定义为:

$$VMExtraPool::=\mathrm{ZMap.}t\ \mathrm{VMExtra}$$

$$VMExtra::=($$

$g\_rax:\mathbb{Z},g\_rbx:\mathbb{Z},\cdots,g\_rip:\mathbb{Z},$　　◁通用寄存器

$g\_cr2:\mathbb{Z},g\_dr0:\mathbb{Z},\cdots,g\_dr6:\mathbb{Z},$　　◁控制/调试寄存器

$h\_ebx:\mathbb{Z},h\_ebp:\mathbb{Z},\cdots,h\_edi:\mathbb{Z},$　　◁宿主寄存器

$entey\_tsc:\mathbb{Z},txit\_tsc:\mathbb{Z},\cdots,last\_tsc:\mathbb{Z},$　　◁虚拟时钟

$launched:\mathbb{B},stopped:\mathbb{B},failed:\mathbb{B},$　　◁虚拟机状态

$pid:\mathbb{Z},vdisk:\mathbb{Z},irqcnt:\mathrm{list}$

$\mathbb{Z},real\_tsc:\mathbb{B},\cdots).$　　◁其他

由于 VMExtra 与虚拟化硬件无关,因此原语 vmx_readz 和 vmx_writez 使用索引而非编码从 VMExtra 中存取数据。

此外,在最底层机器模型中,原语 *host_out* 和 *host_in* 分别对应 *vmentry* 和 *exit* 操作,并以此设置 *a.in_host* 的值。为了确保客户系统的执行不会产生危害,在 *host_out* 的规约中额外进行虚拟机安全状态检查,即在验证过程中若调用者不能保证虚拟机处于安全执行状态,则无法进入客户模式。

3.5 虚拟机安全执行状态

虚拟化系统的安全性可分解为两个方面:1)VMM 行为正确性及控制信息的正确配置;2)客户系统运行时的资源隔离。

考虑虚拟机运行过程中的任意一条活动序列,如图 7 所示,由于虚拟机运行过程中,VMM 虚拟机控制和状态信息的改变在有限的范围内,因此本文使用虚拟机安全执行状态的方式对客户系统上下文中的任意代码进行建模。

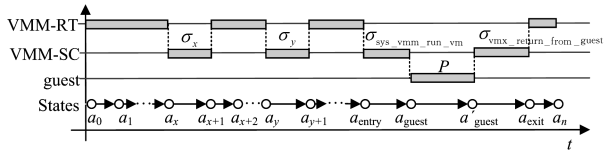


图 7 虚拟化系统的运行时序及状态迁移

Fig. 7 Operation sequences and state transitions of virtualization system

安全执行状态被定义为 VMM 在运行过程中可能处于的且不会破坏客户系统隔离属性的状态。对于这些状态的修改必须通过 VMM-SC 进行监管。在第 4 节中,安全执行状态将被逐步细化为对 EPT 和 VMCS 等结构上的操作的不变量。

4 VMM 的功能正确性验证

本节介绍如何通过抽象层验证 VMM-SC 中各模块功能的正确性。

4.1 虚拟机内存管理

虚拟机内存管理模块主要实现两项功能:1)*ept_init*,初始化 EPT,预先分配各级表项,并将最后一级表项设置为空;2)*sys_vmm_mmap*,绑定虚拟机客户物理页面(GPA)与对应 VMM-RT 进程的用户虚拟页面(HVA)。

ept_init 的规约如式(7)所示,其使用了一系列辅助递归(fix-point)函数,逐一分配并初始化各级 EPT 表项。辅助递归函数 *cal_epml4* 初始化 PML4 表中的第 n 项,并将更新后的 PML4 表传递给自身(见式(8)),直到所有 PML4 表项被初始化时结束递归过程(见式(9))。

$$\frac{a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=false\quad epml4t'=cal\_epml4(RPML4,ZMap.init(EUndef))\quad a'=a[eptpool\leftarrow a.eptpool[\text{vmid}()]\vdash\!\!\rightarrow epml4t']]}{a'=ept\_init(a)} \tag{7}$$

$$\frac{n>0\quad pml4e=cal\_epdpt(RPDPT,ZMap.init(EUndef))\quad epml4t'=cal\_epml4(n',epml4t[n\vdash\!\!\rightarrow pml4e])}{epml4t'=cal\_epml4(n,epml4t)} \tag{8}$$

$$\frac{n=0}{epml4t=cal\_epml4(n,epml4t)} \tag{9}$$

sys_vmm_mmap 的规约如式(10)和式(11)所示。作为系统调用,该原语首先通过 *uarg*♯()从用户上下文中获取参数,然后进行安全检查:1)参数所给地址是否页对齐;2)HVA 是否在进程合法内存空间中;3)调用者是否为合法 VMM-RT 进程。接着,通过 *pt_read* 获得 *hva* 对应页面的 *hpa*。若页面存在于 VMM-RT 进程中,则建立 *hpa* 与 *gpa* 的映射关系(见式(10)),否则首先通过 *pt_resv* 在当前页表中分配一个新的页面,获取其 HPA 后再与 *gpa* 建立映射关系(见式(11))。最后,刷新缓存,使页面映射生效。

$$\frac{\begin{array}{c}a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ gpa=uarg2(a)\quad hva=uarg3(a)\quad perm=uarg4(a)\\ gpa\ mod\ pagesize=0\quad hva\ mod\ pagesize=0\\ USER\_LOW\leqslant hva\leqslant USER\_HIGH\\ vmx=vmxpool[\text{vmid}()]\quad curid()=vmx.pid\\ Some\quad hpa=pt\_read(hva,a)\quad 0\leqslant hpa\leqslant RPADDR\\ Z.land(hpa,PT\_PERM\_P)=1\quad a_0=ept\_mmap(gpa,hpa,perm)\\ a_1=ept\_inv\_map(a_0)\quad a'=uerrno(E\_SUCC,a_1)\end{array}}{a'=sys\_vmm\_mmap(a)} \tag{10}$$

$$\frac{\begin{array}{c}\cdots\\ Z.land(hpa,PT\_PERM\_P)=0\\ a_0=pt\_resv(hva,PT\_PERM\_PTU,a)\\ Some\ hpa=pt\_read(hva,a_0)\quad Z.land(hpa,PT\_PERM\_P)=1\\ a_1=ept\_mmap(gpa,hpa,perm,a_0)\quad a_2=ept\_inv\_map(a_1)\\ a'=uerrno(E\_SUCC,a_2)\end{array}}{a'=sys\_vmm\_mmap(a)} \tag{11}$$

虚拟机内存管理的验证通过 4 个抽象层完成:VEPTIntro 建立内存 EPT 页表和基本访问函数与抽象模型间的精化关系;2)VEPTOp,VEPTInit 和 TSyscall 分别验证 *ept_init*,*ept_mmap* 等中间原语和 *sys_vmm_mmap* 实现与规约间的精化关系。此外,需要证明内存管理模块所引入的规约符合定义 1—定义 3 中的不变量。

定义 1(EPT 结构正确性) 对于任意合法 GPA,均能在 EPT 中找到与之对应的 EPTAB 表项,且表项可以为某一合法 HPA 或者未定义:

$$\begin{aligned}&INV\_VALID\_EPT\_STRUCTURE(a)::=\\ &a[init]=\text{true}\wedge\forall gpa.pml4=PML4I(gpa)\wedge pdpt=EPDPTI(gpa)\wedge pdir=EPDIRI(gpa)\wedge ptab=EPTABI(gpa)\wedge 0\leqslant pml4\leqslant RPML4\wedge 0\leqslant pdpt\leqslant RPDPT\wedge 0\leqslant pdir\leqslant RPDIR\wedge 0\leqslant ptab\leqslant RPTAB\\ &\Rightarrow\exists epml4t,epdpt,epdt,eptab,entry.epml4t=a.eptpool[\text{vmid}()]\wedge\text{EValid}\ epdpt=epml4t[pml4]\wedge\text{EValid}\ epdt=epdpt[pdpt]\wedge\text{EValid}\ eptab=epdt[pdir]\wedge entry=eptab[ptab]\wedge(entry=\text{EUndef}\vee\exists hpg.perm.entry=\text{EValid}(hpg,perm))\end{aligned}$$

定义 2(EPT 正确性) 在虚拟机 EPT 的所有合法映射中,物理页面均来自于对应 VMM-RT 进程的页表,即:

$$\begin{aligned}&INV\_VALID\_EPT(a)::=\\ &\forall gpa,hpa,perm.a.init=\text{true}\wedge\text{Some}(hpa,perm)=ept\_gpa\_to\_hpa(gpa,a)\Rightarrow\exists hva.pt\_read(hva,a)=\end{aligned}$$

$$\text{Some } hpa \wedge Z.\text{land}(hpa, PT\_PERM\_P) = 1$$

定义 3(EPT 缓存一致性)　虚拟机 EPT 映射与地址转换缓存保持一致,即:

$$INV\_VALID\_EPT\_CACHE(a) ::= \forall id.\, a.init = \text{true} \rightarrow a.ept\_stale[id] = \text{false}$$

由于虚拟机仅能从 VMM-RT 进程的 HVA 空间分配新的物理页面,使得虚拟机间的地址空间隔离问题被简化为进程间的地址空间隔离问题,为虚拟机管理提供了更为清晰的内存边界;并且通过 VMM-RT 进程最大化可用内存的限制,解决了恶意虚拟机通过不断分配内存资源来阻塞其他客户系统执行的问题。

4.2　虚拟 CPU

除了提供对 VMCS 和 VMExtra 结构的访问原语外,虚拟 CPU 还提供对二者的初始化及客户/宿主切换的服务。

4.2.1　VMCS 初始化

为了保证虚拟机执行的安全性,VMM-SC 的执行遵循最简安全配置原则,即:

1)在不影响宿主安全性的前提下,尽量使用硬件机制完成客户系统的执行,提高虚拟机的效率;

2)所有可能对客户系统以外的资源产生影响的操作,均要求产生 *vm exit*;

3)当开启某项硬件虚拟化功能需要较复杂的配置,且配置失误可能导致发生安全威胁时,关闭该功能,转而由用户态 VMM-RT 实现;

4)若 VMM 需要,则可以在确定的时间周期内夺回处理器控制权。

VMCS 初始化操作原语的规约如式(12)所示。遵循上述原则,对 VMCS 内的虚拟机控制信息、客户系统上下文和宿主上下文等近 60 个字段逐一进行配置。

$$\frac{\begin{array}{c} a.in\_kern = \text{true} \quad a.pg = \text{true} \quad a.in\_host = \text{true} \quad a.init = \text{false} \\ vmcs = a.vmcspool[\text{vmid}()] \quad d = vmcs.data \\ d_1 = d[PIN\_BASED\_CTLS \mapsto CPIN\_BASED\_CTLS] \\ \cdots \\ a' = a[vmcspool \leftarrow a.vmcspool[\text{vmid}() \mapsto vmcs[data \leftarrow d_n]]] \end{array}}{\text{Some}\quad a' = vmcs\_set\_defaults(a)} \tag{12}$$

VMCS 初始化完成后,建立了虚拟机安全执行状态,要求后续引入的代码遵守定义 4 的 the 不变量。

定义 4(虚拟机控制信息安全)　在 VMM 运行过程中,虚拟机控制信息、客户/宿主状态符合最简安全配置原则,且关键信息不被修改。

$$\begin{array}{l} INV\_SAFE\_VMCS\_CONTROL(a) ::= \\ \quad a.init = \text{true} \\ \quad \Rightarrow let\ data := a.vmcspool[\text{vmid}()].data\ \text{in} \\ \qquad (\text{let Vint } v_1 := data[PIN\_BASED\_CTLS] \text{ in} \\ \qquad Z.land(v_1, CPIN\_BASED\_CTLS\_SETS) = 1 \wedge Z.lor(v_1, CPIN\_BASED\_CTLS\_UNSETS) = 0) \wedge \\ \qquad (\text{let Vint } v_2 := data[PRI\_PROC\_BASED\_CTLS] \text{ in } Z.land(v_2, CPRI\_PROC\_BASED\_CTLS\_SETS) = 1 \wedge Z.lor(v_2, CPRI\_PROC\_BASED\_CTLS\_UNSETS) = 0) \wedge \cdots \end{array}$$

4.2.2　宿主/客户切换

VMM 客户/宿主切换过程基于系统调用 *sys_vmm_run_vm* 完成,其执行流程如图 8 所示。

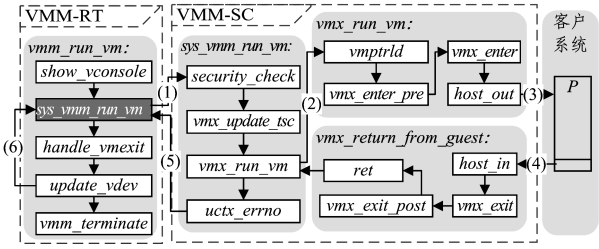


图 8　VMM 主循环与客户/宿主上下文切换处理

Fig. 8　Main loop and guest/host context switch of VMM

首先进行安全检查和虚拟时钟的更新(见图 8 中(1));完成后,转入 *vmx_run_vm* 的执行(见图 8 中(2)),该函数首先进行 VMCS 的切换,然后通过 *vmx_enter_pre* 从 VMExtra 结构中更新客户系统下一次运行时的指令地址(RIP),通过 *vmx_enter* 准备处理器上下文数据(见式(13));最后使用 *host_out* 切换至客户系统(见图 8 中(3))。*vmentry* 操作完成后,客户系统根据 VMCS 中的配置继续执行,直至下一次 *vmexit* 产生时,跳转到 VMCS 中预先设置的返回点,即 *LAsm* 函数:*vmx_return_from_guest*(见图 8 中(4))。该函数首先使用 *host_in* 恢复抽象层状态,然后通过 *vmx_exit*(见式(14))和 *vmx_exit_post* 切换处理器上下文。由于栈顶在上下文恢复后,返回地址与原语 *vmx_run_vm* 被调用时的地址相同,因此通过 *ret* 指令,返回并完成客户/宿主状态切换的全过程。

$$\frac{\begin{array}{c} a.in\_kern = \text{true} \quad a.pg = \text{true} \quad a.in\_host = \text{true} \quad a.init = \text{true} \\ 0 \leqslant \text{vmid}() < NUM\_VMS \quad vmx = a.vmxpool[\text{vmid}()] \\ \rho' = \rho[EAX \leftarrow vmx.g\_rax][EBX \leftarrow vmx.g\_rbx] \cdots \\ vmx' = vmx[h\_ebx \leftarrow \rho[EBX]][h\_ebp \leftarrow \rho[EBP]] \cdots \\ vmx'' = vmx'[enter\_tsc \leftarrow rdtscp()] \quad \rho'' = \rho'[PC \leftarrow host\_out] \\ a' = a[vmxpool \leftarrow a.vmxpool[\text{vmid}() \mapsto vmx'']] \end{array}}{\text{Some}(\rho'', a') = vmx\_enter(\rho, a)} \tag{13}$$

$$\frac{\begin{array}{c} a.in\_kern = \text{true} \quad a.pg = \text{true} \quad a.in\_host = \text{true} \quad a.init = \text{true} \\ 0 \leqslant \text{vmid}() < NUM\_VMS \quad vmx = a.vmxpool[\text{vmid}()] \\ vmx' = vmx[g\_rax \leftarrow \rho[EAX]][g\_rbx \leftarrow \rho[EBX]] \cdots \\ \rho' = \rho[EBX \mapsto vmx.h\_ebx][EBP \mapsto vmx.h\_ebp] \cdots \\ vmx'' = vmx'[exit\_tsc \leftarrow rdtscp()] \quad \rho'' = \rho'[PC \leftarrow RA] \\ a' = a[vmxpool \leftarrow a.vmxpool[\text{vmid}() \mapsto vmx'']] \end{array}}{\text{Some}(\rho'', a') = vmx\_exit(\rho, a)} \tag{14}$$

系统调用执行完成后,不论成功与否,都将返回到 VMM 主循环中(见图 8 中(5)),以便由 VMM-RT 中的 *vm exit* 处理模块(*handle_vmexit* 部分)读取并处理虚拟机请求,继而在更新虚拟设备后,开启下一个虚拟机执行周期(见图 8 中(6))。

4.3　其他模块

在 VMM-RT 处理 *vm exit* 的过程中,需要通过访问 EPT,VMCS 和 VMExtra 等数据来控制虚拟机的后续行为,如模拟敏感指令的执行,进行中断注入,访问设备等。接下来,讨论 VMM-SC 如何为这些操作提供安全的访问接口。

4.3.1 虚拟中断管理

虚拟中断管理为 VMM-RT 提供 3 项功能：1)可中断性检查，判断客户系统是否可以进行合法的中断注入，当客户系统屏蔽中断(见式(15))、执行特殊指令(如 *mov SS* 等，见式(16))或者遇到已注入但还未响应的中断时，VMM 不应产生新的中断注入；2)中断计数，记录中断源当前已被触发的中断请求次数，并通过系统调用 *sys_vmm_get_irqcnt* 将计数值返回给 VMM-RT 进程中的虚拟中断控制器；3)中断注入，当客户可响应中断时，通过 *sys_vmm_inject_event* 在检查调用者身份、注入向量范围和类型的合法性后执行中断注入(见式(17))，否则开启一个新的中断窗口，等待客户系统的中断响应时机。

$$\frac{\begin{array}{c}a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ 0\leqslant \text{vmid}()<NUM\_VMS\quad data=a.vmcspool[\text{vmid}()].data\\ e=GUEST\_RFLAGS\quad \text{None}\neq vmcs\_Z\_to\_encoding(e)\\ \text{Vint } v=data[e]\quad b=Z.land(Int.unsigned(v),BIT\_IF)\\ rv=Z.eqb(b,0)\end{array}}{\text{Some}(rv,a')=vmx\_check\_iflag(a)} \tag{15}$$

$$\frac{\begin{array}{c}a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ 0\leqslant \text{vmid}()<NUM\_VMS\quad data=a.vmcspool[\text{vmid}()].data\\ e=\text{INTERRUPTIBILITY}\quad None\neq vmcs\_Z\_to\_encoding(e)\\ \text{Vint } v=data[e]\\ b=Z.land(Int.unsigned(v),BITS\_STI\_MOVSS)\quad rv=Z.gtb(b,0)\end{array}}{\text{Some}(rv,a')=vmx\_check\_intwin(a)} \tag{16}$$

$$\frac{\begin{array}{c}a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ 0\leqslant \text{vmid}()<NUM\_VMS\quad data=a.vmcspool[\text{vmid}()].data\\ e_{info}=ENTRY\_INTR\_INFO\quad \text{None}\neq vmcs\_Z\_to\_encoding()\\ v_1=Z.lor(type,BIT\_VALID)\quad v_2=Z.lor(v_1,vector)\quad ev=0\\ type=HW\_INTR\quad data'=data[e_{info}\mapsto \text{Vint } v_2]\\ a'=a.vmcspool[\text{vmid}()\mapsto a.vmcspool[\text{vmid}()][data\leftarrow data']]\end{array}}{\text{Some}(a')=vmx\_inject\_event(type,vector,err,ev,a)} \tag{17}$$

4.3.2 虚拟时钟

时钟中断为客户系统的分时执行和调度提供了最基本的计时单位。客户系统可感知的时间有 3 个来源：1)*gTSC*，客户时间戳计数器；2)*vPIT*，为客户系统产生周期性的时钟中断的虚拟定时器；3)*vRTC*，现实世界时钟关联的虚拟实时时钟。其中，*gTSC* 通过虚拟化硬件接口 *VMCS.data*[TSC_OFFSET]实现，其余二者均以虚拟设备的形式在 VMM-RT 中实现。

为了屏蔽由时间引起的信息泄漏，并保持 3 个时钟源的一致性，VMM-SC 采用虚拟戳计数器(*vTSC*)作为最基本的计时单位(计算公式见式(18))，更新所有的虚拟时钟源：

$$vTSC=exit\_tsc_n-\sum_{i=1,2,\cdots,n}(entry\_tsc_i-exit\_tsc_{i-1}) \tag{18}$$

其中，$exit/entry\_tsc_i$ 分别为虚拟机产生 *vm exit* 和 *entry* 的物理时间戳计数值且 $exit\_tsc_0=0$。

原语 *vmx_update_tsc*(见式(19))完成对 *vTSC* 的更新，它在宿主/客户切换时被调用(见图 8)，并同时更新 *gTSC*。系统通过调用 *sys_vmm_get_vtsc* 在式(20)的基础上进行安全检查，为虚拟时钟源的更新提供依据。

$$\frac{\begin{array}{c}a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ 0\leqslant \text{vmid}()<NUM\_VMS\quad vmx=a.vmxpool[\text{vmid}()]\\ data=a.vmcspool[\text{vmid}()].data\\ e=vmx[entry\_tsc]\quad x=vmx[exit\_tsc]\quad t=vmx[last\_tsc]\\ t'=x-e+t\quad tsc\_offset=rdtscp()+C_{\text{entry}}-t'\\ vmx'=vmx[last\_tsc\leftarrow t']\quad a'=a.vmxpool[\text{vmid}()\mapsto vmx']\\ toffset\_lo=Z64\_lo\_int(tsc\_offset)\\ toffset\_hi=Z64\_hi\_int(tsc\_offset)\\ e_{lo}=TSC\_OFFSET\quad \text{None}\neq vmcs\_Z\_to\_encoding(e_{lo})\\ e_{hi}=TSC\_OFFSET+1\quad \text{None}\neq vmcs\_Z\_to\_encoding(e_{hi})\\ data'=data[e_{lo}\mapsto \text{Vint } toffset\_lo][e_{hi}\mapsto \text{Vint } toffset\_hi]\\ a''=a'[vmcspool\leftarrow a.vmcspool[vmid()\mapsto data']]\end{array}}{\text{Some}(a'')=vmx\_update\_tsc(a)} \tag{19}$$

$$\frac{\begin{array}{c}a.in\_kern=\text{true}\quad a.pg=\text{true}\quad a.in\_host=\text{true}\quad a.init=\text{true}\\ 0\leqslant \text{vmid}()<NUM\_VMS\quad vmx=a.vmxpool[\text{vmid}()]\\ e=vmx.entry\_tsc\quad x=vmx.exit\_tsc\quad t=vmx.last\_tsc\\ vtsc=x-e+t\end{array}}{\text{Some } vtsc=vmx\_get\_vtsc(a)} \tag{20}$$

由于 *vTSC* 在宿主运行时暂停，因此客户时钟会逐步落后于真实世界的时间。此情况一方面屏蔽了客户系统对宿主和其他虚拟机执行状况的探测，另一方面也对依赖真实时间运行的应用带来了不便。为了解决此问题，针对可信度较高的客户系统，Hypercall：*hyp_set_real_tsc*，用于停止 VMM 对 *VMCS.data*[*TSC_OFFSET*]的更新，使 *gTSC* 与物理 *TSC* 保持同步。

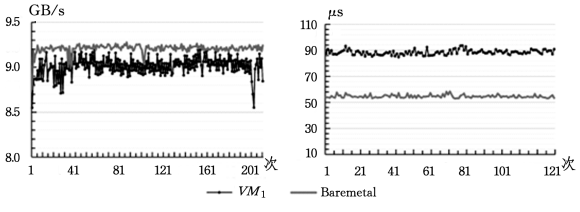
5 实验与性能评估

为了评估 VMM 在实际运行过程中的性能，并测试其是否能满足安全关键系统的隔离要求，本文利用验证后抽取的 VMM 镜像，引导并支持一组带有实验程序的 Linux 客户系统(Ubuntu 16.04.3 LTS，内核版本 4.10.0)运行。实验运行于 Intel Core i5-7600(3.5 GHz，4 核心，6 MB L3 缓存)处理器，8GB 内存和 120GB Toshiba SSD 固态存储器平台。为了确保数据的一致性，禁止 Turbo Boost 和 C-State 状态。此外，在需要获得准确结果的性能测试中，VMM 开启 *real_tsc* 状态(见 4.3.2 节)，每个物理核心上最多运行一个虚拟机。

5.1　运行安全性

本节通过两组实验分别测试 seVMM 是否能满足虚拟机对内存操作和处理器指令执行的安全隔离要求。

实验 1　测试 seVMM 对于内存操作的隔离性。实验由分别运行于两个处理器核心上的虚拟机(VM_1 和 VM_2)组成。首先在 VM_1 内启动并持续执行 lmbench[35] 中的内存带宽测试程序 *bw_mem*,然后在 VM_2 中使用内存压力测试工具 memtester 对物理内存(特别是内核和设备寄存器映射区域)进行写入,观察两个虚拟机的执行情况。VM_1 隔离性测试结果与基准的对比结果如图 9 所示。



(a)内存带宽　　(b)fork 执行时延

图 9　VM_1 隔离性测试结果与基准的对比结果

Fig. 9　Comparison results of isolation test performed on VM_1 to baremetal references

由于 memtester 对 VM_2 物理内存的写入破坏了客户系统的内核数据,因此在执行一段时间后 VM_2 崩溃,但 VM_1 保持正常运行。图 9(a)中的序列"VM_1"涵盖了 memtester 执行前后及 VM_2 崩溃前后 VM_1 中内存拷贝性能测试的结果。与没有执行 memtester 的基准序列"Baremetal"相比,其带宽基本保持恒定,VM_2 无法通过内存写入来影响 VM_1 的正常运行。

实验 2　测试 seVMM 的性能隔离。实验继续沿用位于两个物理核心的虚拟机 VM_1 和 VM_2。在 VM_1 中启动并持续运行 lmbench 进程创建 fork 测试项,然后在 VM_2 中使用 shell 脚本":(){ :|:& };:"对客户系统进行 fork 炸弹拒绝服务攻击,观察两个虚拟机的执行情况。

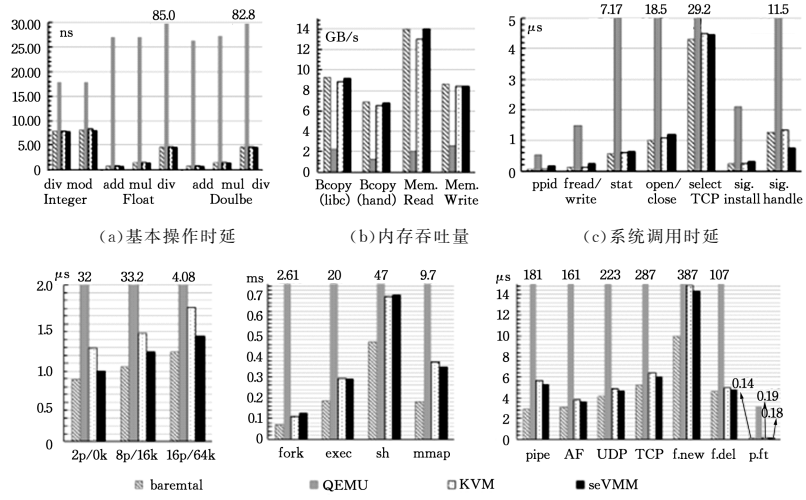
VM_2 在执行脚本一段时间后由于资源耗尽而崩溃,而 VM_1 则继续正常运行。图 9(b)中的序列"VM_1"记录了 VM_2 在执行脚本前后和崩溃前后,fork 测试项的性能数据。与在物理机上运行且未进行 fork 攻击的基准序列"Baremetal"相比,其时延基本保持稳定,针对 VM_2 的 fork 攻击并未对 VM_1 产生明显影响。

在功能方面,主流虚拟化方案(如 Xen,KVM 等)同样以虚拟机隔离性作为 VMM 设计的安全目标之一。但对于系统是否能满足安全属性的要求,则无法提供确凿的证据。本文通过约 39 kloc 的 Coq 代码,对运行在内核态中的 VMM-SC 完成了形式化验证。对所有证明目标进行了机器检查并链接形成最终定理(见式(2))。因此,安全性和正确性具备了较高的可信度。

5.2　性能评估

本节分别测试 VMM 的基准性能和综合性能。

实验 3　通过比较分项基准性能测试程序 lmbench[35] 在物理机(baremetal)、QEMU[36]、KVM[37] 和抽取的 VMM 镜像(seVMM)上的运行指标,分析和评估本文虚拟化方法在执行普通指令、特权指令、内存访问、外设访问等方面对客户系统所产生的性能负载。lmbench 性能测试结果如图 10 所示。



(a)基本操作时延　(b)内存吞吐量　(c)系统调用时延

(d)上下文切换时延　(e)进程执行时延　(f)其他

图 10　lmbench 性能测试结果

Fig. 10　Performance metric results of lmbench

由图 10 可以看出,由于 seVMM 和 KVM 均采用硬件辅助虚拟化技术,且使用扩展页表(EPT)进行内存寻址,因此其在执行普通指令和进行内存访问的过程中,性能与物理机非常接近。在内存的读取和拷贝过程中,seVMM 的性能稍好于 KVM,其原因是 seVMM 较小的热点内存占用提高了缓存命中率。而 QEMU 使用的二进制转换机制在复杂指令执行,特别是浮点运算中,产生了较大的性能差距。

在执行特权指令时,由于伴随着 *vm exit* 的产生,并需要 VMM 协助处理,因此 seVMM 和 KVM 的性能都略低于物理机基准。此外,由于 KVM 对部分客户系统所产生的系统调用进行了优化,因此其性能稍好于 seVMM。

在执行更为综合的任务时,KVM 的性能与 seVMM 较为

接近，在 14 个测试项中，有 8 个标准化时延增量的差额小于 10%；在剩余的 6 个测试项中，仅有系统调用密集型的 fork 测试 seVMM 的性能低于 KVM。

实验 4 通过比较宏基准测试程序 DaCaPo[38] 在各虚拟化平台上客户系统中运行时的性能指标，考虑较长运行周期内，VMM 对虚拟机性能的综合影响，以评估 VMM 对复杂、综合任务执行的可行性。

DaCaPo 测试集包含了系统并发能力(AVRORA)、图片渲染(batik，SunFlow)、开发环境(Eclipse)、代码分析编译与解释(fop，Jython，pmd 和 xalan)、文本查找与索引(Luindex，Lusearch)、数据库(h2)和服务器(Tomcat，Tradebeans 和 Tradesoap)等 14 个性能测试项。测试结果如图 11 所示。

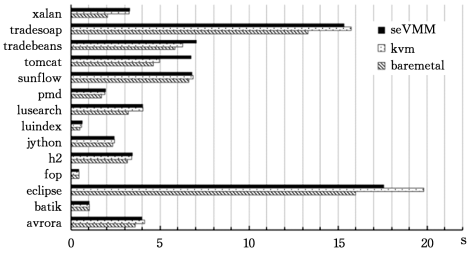


图 11 DaCaPo 性能测试结果

Fig. 11 Performance metric results of DaCaPo

在多数情况下，由于 KVM 所运行的宿主内核 Linux 比 seVMM 更为复杂，且需要执行大量后台任务，因此处理时延略高于 seVMM，但二者标准化时延增量的平均值不超过基准的 19%，本文所设计的 VMM 对于大多数任务的执行具备一定的实用性。

结束语 本文描述了一种面向安全关键系统的虚拟机监视器设计与验证过程。以安全为边界，将 VMM 划分为两个部分：在内核中运行与虚拟机安全直接相关的核心组件 VMM-SC；以用户进程为载体，实现虚拟机处理逻辑的运行时 VMM-RT。验证采用分层精化的方法，通过构建抽象层，逐步引入 VMM 的模块，并建立其与规约间的前推模拟关系，最终获得内核规约与其实现间的上下文相关精化关系证明。实验结果表明，安全划分和形式化验证没有对 VMM 的运行性能产生明显影响。

未来工作包括：优化 VMM 对虚拟机请求的处理算法，以进一步提升虚拟机的执行效率；通过增加新的虚拟设备和支持更多的处理器架构，使得 VMM 满足更多应用领域的需求。

参 考 文 献

[1] SMITH J，NAIR R. Virtual machines：versatile platforms for systems and processes[M]. San Francisco，California，USA：Elsevier，2005.

[2] SHU R，WANG P，GORSKI S A，et al. A Study of Security Isolation Techniques[J]. ACM Computing Surveys，2016，49(3)：1-37.

[3] CHONG S，GUTTMAN J，DATTA A，et al. Report on the NSF Workshop on Formal Methods for Security[R]. College Park，MD，USA：National Science Foundation，2016.

[4] SCHMITTNER C，MA Z. Towards a Framework for Alignment Between Automotive Safety and Security Standards[C]// Computer Safety，Reliability，and Security：SAFECOMP 2015 Workshops. Cham，Netherlands：Springer International Publishing，2015：133-143.

[5] BECKER H，CRESPO J M，GALOWICZ J，et al. Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor[C]// International Symposium on Formal Methods. Limassol，Cyprus：Springer，Cham，2016：69-84.

[6] MATICHUK D，MURRAY T，ANDRONICK J，et al. Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification[C]// Proceedings of the 37th International Conference on Software Engineering-Volume 1. Piscataway，NJ，USA：IEEE Press，2015：722-732.

[7] WU C，WANG Z，JIANG X. Taming hosted hypervisors with (mostly) deprivileged execution[C]// Proceedings of the Network and Distributed System Security Symposium (NDSS). San Diego，CA，USA：Internet Society，2013：1-15.

[8] PAN W，ZHANG Y，YU M，et al. Improving Virtualization Security by Splitting Hypervisor into Smaller Components[C]// Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy. Berlin，Heidelberg：Springer-Verlag，2012：298-313.

[9] SANÁN D，BUTTERFIELD A，HINCHEY M. Separation Kernel Verification：The Xtratum Case Study[C]// Working Conference on Verified Software：Theories，Tools，and Experiments. Vienna，Austria：Springer，Cham，2014：133-149.

[10] PENNEMAN N，KUDINSKAS D，RAWSTHORNE A，et al. Formal virtualization requirements for the ARM architecture [J]. Journal of Systems Architecture，2013，59(3)：144-154.

[11] BARTHE G，BETARTE G，CAMPO J D，et al. Formally Verified Implementation of an Idealized Model of Virtualization[C]// 19th International Conference on Types for Proofs and Programs (TYPES 2013). Dagstuhl，Germany：Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik，2014：45-63.

[12] NEMATI H，GUANCIALE R，DAM M. Trustworthy Virtualization of the ARMv7 Memory Subsystem[C]// International Conference on Current Trends in Theory and Practice of Informatics. Berlin：Springer，2015：578-589.

[13] BLANCHARD A，KOSMATOV N，LEMERRE M，et al. A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C[C]// International Workshop on Formal Methods for Industrial Critical Systems. Berlin：Springer，2015：15-30.

[14] BAUMANN C，SCHWARZ O，DAM M. Compositional Verification of Security Properties for Embedded Execution Platforms [C]// 6th International Workshop on Security Proofs for Embedded Systems(PROOFS 2017). Taipei，Taiwan，China，EPiC open access，2017：1-16.

[15] NORDHOLZ J. Design and provability of a statically configurable hypervisor[D]. Berlin：Technische Universität Berlin，2017.

[16] LEINENBACH D,SANTEN T. Verifying the Microsoft Hyper-V Hypervisor with VCC[C]// Proceedings of the 2Nd World Congress on Formal Methods. Berlin,Heidelberg:Springer-Verlag,2009:806-809.

[17] ALKASSAR E,HILLEBRAND M A,PAUL W J,et al. Automated Verification of a Small Hypervisor[C]// Proceedings of the Third International Conference on Verified Software:Theories, Tools, Experiments. Berlin, Heidelberg: Springer-Verlag, 2010:40-54.

[18] KLEIN G,ANDRONICK J,ELPHINSTONE K,et al. Comprehensive Formal Verification of an OS Microkernel[J]. ACM Transactions on Computer Systems,2014,32(1):2:1-2:70.

[19] VASUDEVAN A,CHAKI S,JIA L,et al. Design,Implementation and Verification of an eXtensible and Modular Hypervisor Framework[C]// 2013 IEEE Symposium on Security and Privacy. Berkeley,CA,USA:IEEE Press,2013:430-444.

[20] BOLIGNANO P. Formal models and verification of memory management in a hypervisor[D]. Breizh:Université Rennes 1, 2017.

[21] BARTHE G,BETARTE G,CAMPO J D,et al. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization[C]// 2012 IEEE 25th Computer Security Foundations Symposium. Cambridge,MA,USA:IEEE Press,2012:186-197.

[22] BOLIGNANO P,JENSEN T,SILES V. Modeling and Abstraction of Memory Management in a Hypervisor[C]// International Conference on Fundamental Approaches to Software Engineering. Berlin,Germany:Springer,2016:214-230.

[23] KOVALEV M. TLB virtualization in the context of hypervisor verification[D]. Saarbrucken:Saarland University,2013.

[24] MOGOSANU L,CARABAS M,CONDURACHE C,et al. Evaluating Architecture-Dependent Linux Performance[C]// 2015 20th International Conference on Control Systems and Computer Science. New York:IEEE Press,2015:499-505.

[25] ABDELRAHEM O,BAHAA-ELDIN A M,TAHA A. Virtualization security:A survey[C]// 2016 11th International Conference on Computer Engineering Systems (ICCES). Cairo, Egypt: IEEE Press,2016:32-40.

[26] FENG X,SHAO Z,GUO Y,et al. Combining Domain-Specific and Foundational Logics to Verify Complete Software Systems [C]// Proceedings of the 2Nd International Conference on Verified Software:Theories,Tools,Experiments. Berlin,Heidelberg: Springer-Verlag,2008:54-69.

[27] GU R,KOENIG J,RAMANANANDRO T,et al. Deep Specifications and Certified Abstraction Layers[C]// Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York,NY,USA: ACM,2015:595-608.

[28] BERTOT Y,HUET G,CASTÉRAN P,et al. Interactive Theorem Proving and Program Development:Coq'Art:The Calculus of Inductive Constructions[M]. Berlin,Germany:Springer Berlin Heidelberg,2013:1-472.

[29] HEISER G. The Role of Virtualization in Embedded Systems [C]// Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems. New York, NY, USA: ACM, 2008:11-16.

[30] GE Q,YAROM Y,COCK D,et al. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware[J]. Journal of Cryptographic Engineering,2018,8(1): 1-27.

[31] KIM J H,LEE S H,JIN H W. Supporting Virtualization Standard for Network Devices in RTEMS Real-time Operating System[J]. ACM SIGBED Review,2016,13(1):35-40.

[32] KLEIN G. Operating system verification—An overview [J]. Sadhana,2009,34(1):27-69.

[33] LEROY X,BLAZY S,KÄSTNER D,et al. CompCert-A Formally Verified Optimizing Compiler[C]// ERTS 2016:Embedded Real Time Software and Systems,8th European Congress. Toulouse,France:SEE,2016.

[34] LEROY X,BLAZY S. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations[J]. Journal of Automated Reasoning,2008,41(1):1-31.

[35] STAELIN C. Lmbench:an extensible micro-benchmark suite [J]. Software: Practice and Experience, 2005, 35(11): 1079-1105.

[36] KLEINERT B,WEI S,SCHÄFER F,et al. Adaptive Synchronization Interface for Hardware-Software Co-Simulation Based on SystemC and QEMU[C]// Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques. ICST, Brussels,Belgium,Belgium:ICST,2016:28-36.

[37] DESHPANDE S M,AINAPURE B. An Intelligent Virtual Machine Monitoring System Using KVM for Reliable And Secure Environment in Cloud[C]// 2016 IEEE International Conference on Advances in Electronics, Communication and Computer Technology (ICAECCT). New York: IEEE Press, 2016: 314-319.

[38] PU J,SONG Z,TILEVICH E. Understanding the Energy,Performance,and Programming Effort Trade-Offs of Android Persistence Frameworks[C]// 2016 IEEE 24th International Symposium on Modeling,Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). London, United Kingdom:IEEE Press,2016:433-438.