当设计计算机监控系统的时候，一个目标对监控目标有一个完整的了解，同时能秘密的保护监控器本身。达到此目的的一种方法是基于hypervisor的，或者更普遍的是基于虚拟机（VM）外部的监视。但是，很多挑战限制了这种机制的使用，其中最重要的是语义鸿沟问题。在过去的十年中，已经进行了大量的研究来弥合语义鸿沟和开发各种out-of-VM监控技术和应用程序。通过追踪out-of-VM安全解决方案的演变，本文将对已提出的克服语义鸿沟的不同方法进行检查和分类，以及它们如何被用于开发各种安全应用程序。特别是，我们回顾过去解决不同限制的方式，例如实用性，灵活性，覆盖范围和自动化，同时弥补语义差距;他们如何开发不同的监测系统;以及监测系统如何已被应用和部署。除了系统化所有提出的技术之外，我们还讨论了这些技术遗留下的研究问题，并阐明了hypervisor-based监控的未来方向。

1. INTRODUCTION

计算机系统监控是维护系统的基本安全机制。入侵检测，访问控制（例如，DAC，MAC和RBAC），沙盒，内联参考监视器，防火墙和防病毒均涉及安全监控。一个理想的监测系统应该有一个监测目标的完整视图和（偷偷地）保护监控系统本身的能力。虽然有许多方式这样做，但是这不是一个简单的任务。 在过去的几十年里，大量的研究被开展来寻求更好更安全的发展方式这种监视器。

到目前为止，一个既有宏伟视野又有强大保护的有前景的针对系统的监视器的战略，需要利用计算机系统被设计成层次结构的事实。传统上，从上到下，有应用层、操作系统（OS）层和硬件层。通常，每一层都实现它上面的层的接口，它使用底层的预定义的接口层执行自己的功能。每个层与上面的层很好地隔离。层次越低，对系统的控制就越多，抽象就越少。

作为一个运行在硬件和OS层之间的层，hypervisor是20世纪60年代首次提出[ Popek和Gordbg 1974 ]。hypervisor也称为虚拟机监视器（VMM）-使计算环境同时在单个物理计算机中运行多个独立的OSes，从而更有效地利用可用的计算能力、存储空间和网络带宽。在我们的计算栈中引入hypervisor层的根本原因是单个服务器的容量如此之大，以至于大多数工作负载几乎不可能有效使用它；因此，虚拟化成为提高资源利用率同时简化和自动化计算单元管理的最好方式。虽然这个计算模型最初被设计为划分大型机上运行的不同程序上时间共享资源，其现在支撑着今天的云计算和数据中心。

除了将我们的计算范式从多任务处理推向多操作系统，hypervisor还推动了传统的虚拟机内的系统监控到虚拟机外，基于hypervisor的监视。这是因为guest操作系统运行在VMM提供的虚拟资源上，其为灵活性和控制提供了新的机会，因为VMM本质上是一个软件，软件层和软件更容易修改，迁移和监视。通过在VMM层提取和重构客户操作系统状态，虚拟机外监视器变为可能：授权监测系统来控制，隔离，干预，检查，治愈并从外部管理虚拟机。Garfinkel和Rosenblum关于这个话题的开创性论文，介绍了第一个hypervisor-based的监控系统“，称这种来自于为了分析运行在其中的软件而检测虚拟机的方法为虚拟机内省（VMI）。

但是，所有虚拟机解决方案都必须解决由于位于客户操作系统下一层而造成的语义鸿沟问题。具体而言，存在语义鸿沟是因为在hypervisor层，我们只能访问一个虚拟机硬件级别的原始数据状态 - 即它的CPU寄存器和物理内存，尽管我们也可以访问特定类型hypervisor的指令级执行状态（例如，二进制基于翻译的VM）。 但是，我们想要的是关于来宾操作系统状态的语义信息，例如被访问的变量，变量类型和来宾操作系统内核事件。 因此，我们必须弥合语义鸿沟以获得有意义的guest OS状态信息。在过去的十年里，有很多为了解决这个问题提出的方法，在不同的约束条件下运行，并且从不同的角度出发，从纯手工到完全自动化。

VM之外的监控提供了许多优于传统的VM内监控的优势，因为他们在更高的特权级别运行，并且与他们监视的客户操作系统内的攻击隔离，也因为它们在客户操作系统下面一层，并且可以介入所有客户操作系统事件。 因此，虚拟机外监视器已被广泛使用在许多安全应用程序中，从只读内省到可写的重构和修复，从被动入侵检测到进行主动预防，从防御恶意应用到恶意操作系统，恶意软件分析到存储器

取证等等。

鉴于如此大量的虚拟机监控研究，迫切需要系统化这个领域的知识。因此，在这篇文章中，我们希望跟随技术脚印并追踪虚拟机监测的发展，重新审视已完成的工作，讨论我们的位置，并阐明发展的方向。 我们在第2部分讨论为什么需要虚拟机外监视和技术的背景。然后我们在第3节中描述所有的虚拟机外监视面临的语义鸿沟挑战。接下来，我们在第4部分总结了这个挑战是如何在不同约束条件下解决的，在第5部分对所有的out-of-VM应用程序进行分类，并且在第6部分介绍如何部署它们。在第7节，我们讨论了剩余的研究问题和未来out-of-VM监测的趋势，相关工作回顾在第8节。最后，我们在第9节中总结。

1. BACKGROUND

在本节中，我们将讨论为什么我们需要VM监控的动机。我们首先回顾了in-VM监测，并在第2.1节讨论了它的优点和缺点。然后在第2.2节我们介绍了out-of-VM监控系统，并讨论了它的优缺点。最后，我们在第2.3节中讨论了这项工作的范围。

2.1 In-VM–Based Monitoring

一般的，一个安全的监控系统可以被定义为:

M(S, P) → {True, False},

其中M表示安全强制机制，S表示当前系统状态，P表示预定义策略。如果当前状态S满足安全性策略P，那么它处于安全状态（TRUE），如果M是在线机制，则可以允许继续执行。否则，它是不安全的（false）；一个攻击被检测到，并且M可以停止执行（用于预防）或报告有攻击实例。例如，在防病毒系统中，S可以表示当前的内存和磁盘状态，并且P表示病毒的签名；如果M标识有任何正在运行的进程或可疑文件有一个在P中定义的签名，反病毒会引发警报。在基于系统调用的入侵检测系统中，S可以表示当前的系统调用，P可以表示S的正确状态机；如果M确认S偏离了P，则可以引发入侵警报。除此之外，各种各样的其他安全工具（例如，安全参考监视器）都属于安全性监控类别。

在任何监控系统中的一个关键步骤是如何从预定义的传感器收集状态信息S，例如嵌入在运行的OS或进程中的状态信息，然后用一个定义良好的安全策略P来监视它们，在out-of-VM monitoring出现之前，大多数监控系统是基于in-VM的，其作为OS应用程序正常运行，或者在OSes中作为可加载内核模块运行。

优点：由于虚拟机监控驻留在操作系统，它使状态采集方便、快捷。具体来说，in-VM实现有以下优势：

——丰富的接口：虚拟机内部监视器可以从很多接口中提取操作系统和进程的状态。他们可以使用关键的内核变量，导出注册表或PROC文件或日志文件（磁盘）和系统调用或操作系统或第三方供应商提供的函数库调用。在更高的抽象层次上，他们也可以使用现有的虚拟机的安全工具来提取状态。监视机制M可以简单地拦截系统调用或库调用，并检查它们的参数、返回值或序列。他们也可以从存储器或磁盘中提取运行进程的已知签名（例如，代码哈希），并验证其完整性。它们还可以监控细粒度的控制流传输并检查其完整性。

——速度快：VM监控执行得也很快。与out-of-VM解决方案比较，in-VM的状态采集、安全检查和执法都是本地执行。例如，in-VM监控程序中，虚拟机内部的状态可以直接访问，in-VM执行可以立即执行而不用陷入hypervisor。

缺点：虽然in-VM系统已经非常成功，他们有一个共同的弱点，他们可以被攻击，因为他们执行在与他们所保护系统相同的权限级别（除非特别注意保护监控，例如，通过hypervisor使用特殊的内存保护执行）。恶意软件，如内核rootkits，或更一般的入侵或攻击，通常可以篡改M(S, P)中的所有组件，比如收集状态信息的传感器和执行安全策略的监控工具，更具体地，他们可以：

——生成假状态S：为了误导监视系统，攻击者可以修改日志、注册表、PROC文件或任何其他具有虚假数据（或甚至负责生成数据的代码）的状态信息，只要系统能够继续运行（例如，没有崩溃）。

——篡改安全策略P。攻击者还可以修改安全策略P，如果它是已知的。例如，攻击者可以修改签名数据库以规避他们的攻击。

——篡改执行机制M。有大量的方法可以被用在这里。例如，如果安全机制是基于系统调用挂钩，攻击者可以修改系统调用表绕过安全检查；如果安全机制是使用一个内核模块或个人监测进程部署的，攻击者可以简单地删除或关闭监控模块或进程。

2.2 Out-of-VM–Based Monitoring

由于in-VM系统的缺点，研究人员提出了将监控移动在VM之外。特别是，基于hypervisor的或所谓的基于VMM，系统使用hypervisor层来保护监控系统。理论上，由于hypervisor运行在比监控系统更低的级别，所以它们也被隔离并且更安全。在过去的十年中，基于hypervisor的监控一直是广泛研究的主题，如前一节中简要提到的。

一个虚拟机运行或直接在主机的硬件（裸机）或运行另一个主机操作系统，并提供了一个类似主机的硬件的软件控制层

根据虚拟机管理程序所在的位置，虚拟机管理程序可分为两种类型：

1. 类型1(裸金属)hypervisors，它直接运行在主机的硬件上，以控制硬件和监视来宾操作系统。这些管理程序的典型示例。包括Xen, VMware ESX和微软Hyper-V。
2. 类型2(寄居)hypervisors，它运行在传统操作系统中。换句话说,一个托管管理程序在主机操作系统上添加了一个独立的软件层，而guest操作系统则成为了硬件之上的第三个软件层。典型的2型hypervisor包括KVM、VMware工作站、VirtualBox和QEMU。

虽然之前的1型和2型hypervisor分类已经被广泛接受，但有时它在同一类型的hypervisor之间(例如KVM vs. QEMU)之间的差别不大。因此，基于虚拟化如何设计(硬件vs软件)以及客户操作系统及其应用程序代码，我们可以在本文中使用另一种类型的虚拟机监控程序:

1. 本地管理程序直接推动客户代码在使用硬件虚拟化的硬件上本地执行(Adams和Agesen 2006)。
2. 使用软件虚拟化(Adams和Agesen 2006)翻译每个客户指令来模拟执行的 仿真管理程序。

本地管理程序的例子包括Xen、KVM、VMware ESX、Microsoft HyperV，仿真管理程序包括QEMU、Bochs和VMware-工作站和VirtualBox的早期版本(注意，最近的VMware-工作站和VirtualBox能够自动执行来宾操作系统代码)。由于没有涉及二进制代码的翻译，所以本地管理程序运行的速度要比模拟管理程序快得多。

有了虚拟机监控程序，系统开发人员就有了一个附加的控制层。它们使用多路复用资源(例如，以类似于调度进程的方式调度vm)，并使用软件迁移和控制vm。在本地管理程序中，系统开发人员可以在某些硬件事件(例如，中断、页面错误异常)中获得控制。对于仿真管理程序，系统开发人员可以随时得到控制，因为所有的指令都可以被方便地捕获。

优点：由于管理程序提供的巨大机会，我们可以推动安全监控进入系统管理程序(即从in-VM到out- vm)。vm监视器在操作系统内部运行，而在操作系统外部运行的vm监视器则位于系统管理程序层。尽管这两种类型的安全监视器可以执行大多数相同的功能(例如，识别恶意软件，检测入侵)，移动监视功能到vm之外有巨大的好处。特别是，我们可以:

——强隔离(防篡改):因为客户操作系统在一个单独的级别上运行。在管理程序之上，当控制在两者之间传递时，有一个全局开关。系统管理程序因此在安全监视器和在来宾操作系统中的攻击之间提供了强大的隔离。除非管理程序有漏洞，否则它会使M和P篡改恢复，因为它们位于来宾操作系统之下。尽管攻击者可能仍然会生成错误的数据，但如果out-of- VM解决方案直接从原始数据中提取S，他们也可以在很大程度上抵御错误的数据生成攻击。

——透明部署:在hypervisor层部署安全监视器，我们不需要在客户操作系统中使用帐户，也不需要在操作系统中安装软件。相反，任何事情都可以透明地发生在虚拟机监控程序层，而不需要中断服务(例如，许多只读的内省技术可以在运行时透明地部署)。

——完整视图:在VM监视器上的另一个优点是，虚拟机监控程序可以完全访问操作系统运行的VM的所有内存、寄存器和磁盘状态。我们可以观察每个应用程序的状态，以及内核状态，包括被攻击者隐藏的那些隐形的状态，这通常是通过in-vm方法实现的挑战。

——high cost savings:out -of- vm还为系统开发人员提供了对虚拟资源的无限制访问。例如，他们可以创建一个沙箱环境，让真正的恶意软件执行并观察其行为，然后简单地丢弃恶意破坏的VM。它们还可以保存来宾操作系统状态的快照，这可以在不影响运行VM的性能的情况下进行分析。这些是in-vm系统经常缺少的特性。

——更少的脆弱性:在虚拟机系统中，通常需要信任整个客户操作系统内核，这往往会有一个庞大的代码库。但是，out- of-vm通常只需要信任底层的管理程序，它有一个较小的代码库。例如，Xen管理程序的代码行数少于Linux内核的1 / 12;这个较小的攻击面会导致更少的漏洞。

缺点：尽管在vm监控中，out-of-vm监控提供了许多优点，但它也有局限性。具体地说,

——没有接口:对于out-of-vm监视器，没有客户操作系统接口。因此，所有out-of-vm解决方案都面临一个必须解决的挑战，以执行有效的监视;他们必须消除在客户操作系统之外移动监视器所引起的语义鸿沟。在第三节讨论了语义鸿沟问题的细节。

——速度慢:此外，out-of-vm监控必须执行额外的地址转换(它所观察到的是物理内存地址，它必须在这些地址和来宾的虚拟地址之间进行转换)，并且要陷入到监控程序进行安全检查和监视。因此，与虚拟机监控相比，它通常要慢一些，尽管最近有一些努力提高了世界切换的性能。

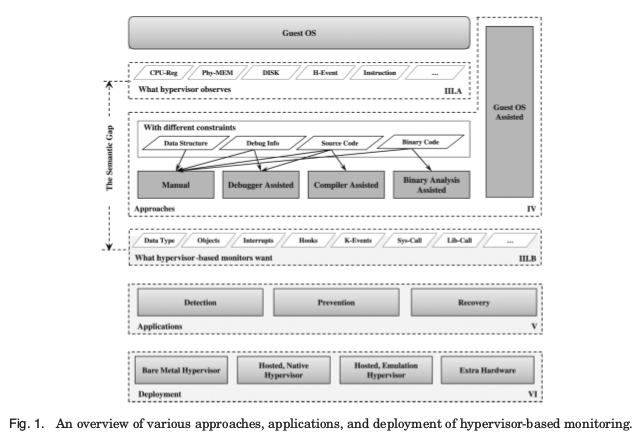
2.3 scope

在本文中，我们关注的是在OS层下面的out-of-vm的解决方案，但是仍然监视机器的内部内容。因此，我们不考虑在网络设备上运行的out-of-vm解决方案(例如，防火墙、网关和交换机)。

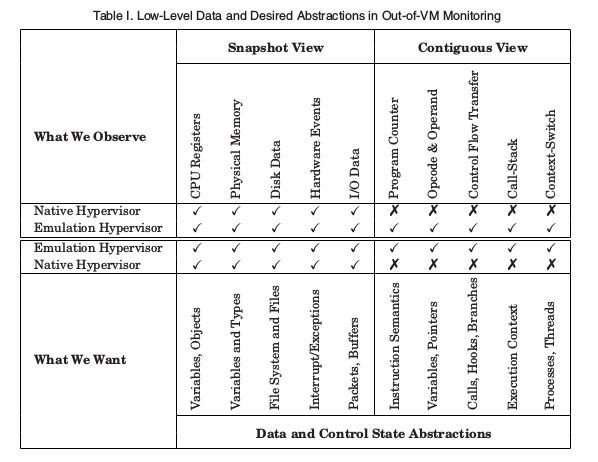
严格地说，我们在技术上也应该排除那些经常使用额外硬件(例如，PCI设备)用于监视的硬件解决方案，因为硬件通常不属于hypervisor层（被定义为软件）。但是，我们仍然包括它们，因为使用纯硬件的方法也可以实现out-of-vm监视。这些硬件解决方案不仅面临语义鸿沟的挑战，如果系统中存在一个管理程序，它们甚至会更加困难;他们必须绕过管理程序层，以内省客户操作系统。因此，对于所有类型的out-of-vm监视技术的完整视图，我们包括这些额外的硬件方法。

1. 语义鸿沟问题

in-VM系统的主要优点是能够直接访问所有类型的os接口。然而，当使用hypervisor时，访问操作系统内部所有丰富的语义抽象就会丢失。尽管虚拟机监控程序对它们监视的vm的整个状态有一个全局的视图，但不幸的是，这个大视图只包含一个没有上下文的0和0。因此，我们可以观察到的(第3.1节)和我们想要的(第3.2节)之间存在语义上的差异，如图1所示，我们必须将其连接起来以提供有效的监视服务。在本节中，我们将更详细地介绍语义鸿沟问题。



3.1 what we Observe



由于hypervisor对所有计算资源进行虚拟化，并向来宾操作系统提供抽象，因此从理论上讲，它们可以查看来宾操作系统状态的任何方面。但是，如表I所示，不同类型的hypervisor对不同的逻辑资源进行虚拟化;因此，他们有不同的观点。

本地管理程序直接推动客户代码在硬件上运行。因此，他们不能连续监视客户操作系统的执行(除非他们使用单步执行)。相反，它们从中断或其他硬件事件中截取控制。因此，它们有一个VM状态的快照视图，当某些硬件事件发生时，它通常被获取，并允许hypervisor重新获得控制。该类型的管理程序可以观察以下内容:

——CPU寄存器:当管理程序获得控制权时，所有的CPU寄存器都可以读取，因为它运行在最高权限级别。

——来宾操作系统内存:整个客户操作系统的内存状态也可以被观察到。但是，hypervisor只能访问物理地址，这些地址必须在访问每个特定内存单元时被转换成虚拟地址。

——硬盘内容:与内存镜像类似，客户操作系统的磁盘映像的内容(如果未加密)也可以在hypervisor中看到。

——硬件事件:所有硬件级别的事件，包括计时器、中断和异常，都可以被观察到。

——I/O流量:hypervisor还负责监控所有I/O流量，包括网络流量、磁盘I/O和击键。

注意，在技术上，本地管理程序可以获得来宾操作系统的连续视图(例如，通过单步执行)，但是这样做的速度太慢;它需要将大多数事件捕获到系统管理程序中，需要不断地切换。因此，大多数时候，本地管理程序只能观察一些特殊的vmm级别的指令(例如，Intel VT指令)和特殊的内核事件，如in-terrupts和异常(例如，如果在硬件中没有扩展或嵌套的页表，则会出现页面错误)。

仿真程序管理程序除了可以看到由本地管理程序所观察到的快照视图之外，还可以对guest-OS执行有一个连续的视图。特别是，他们可以观察到下列情况:

——程序计数器:他们可以知道哪些指令被执行，以及他们的反汇编代码。

——指令操作码和操作数:对于每个执行指令，可以观察其操作码和操作数。

——控制流传输:可以观察所有控制流传输(例如，调用/jmp/ret，条件分支)，以及它们的源地址和目标地址(如果适用)。

——调用栈:如果存在堆栈帧指针，则可以遍历堆栈，或者可以透明地检测指令来构建调用堆栈信息。

——上下文切换:也可以观察到每个特定的进程或线程执行上下文。

3.2. What We Want

由于hypervisor只能访问低级的二进制数据，因此我们必须将数据转换为更高级别的抽象，以提供有用的监视服务。通常，这些抽象可以分为两种类型:数据状态抽象和控制状态抽象。

数据状态抽象。数据状态包括当前内核对象的状态或被监视进程内的对象(主要是内存状态)以及当前的CPU执行状态。更具体地说，我们感兴趣的是:

——变量、对象和虚拟地址空间:考虑到客户操作系统的物理内存，我们希望了解内核的位置或监视的进程变量(或对象)、变量或对象的虚拟地址，以及如何在物理内存中定位虚拟地址。

——数据结构类型及其连接:我们还希望了解对象类型、数据结构和它们的点对关系(这样我们可以遍历它们并验证它们的完整性)。

——文件系统和文件:给定磁盘映像，我们对正在使用的文件系统的类型和文件所在的位置感兴趣。

——中断、异常和其他内核事件:对于一个观察到的硬件事件，我们希望获得更多关于它的详细信息;对于一个中断或异常，我们想知道它是哪个特定的中断或异常。此外，如果可能的话，我们还希望识别其他内核事件，比如创建或销毁某个内核对象时。

——数据包和缓冲区:从观察到的I/O数据，我们希望区分网络数据包和DMA缓冲区。更重要的是，如果可能的话，我们希望确定包的内容和上下文。

控制状态抽象。虽然数据状态提供了运行的vm的快照视图，但它的粒度取决于管理程序控制的时间和频率。在本地管理程序中，hypervisor只有在某些硬件事件发生时才会重新获得控制，而在仿真管理程序中，我们可以任意时刻控制(取决于我们的兴趣)。正如第3.1节所提到的，使用本地管理程序获得一个连续的视图虽然可能，但对于大多数应用程序来说效率太低。因此，我们主要关注控制状态抽象的仿真管理程序。更具体地说，从细粒度到粗粒度，我们感兴趣的是:

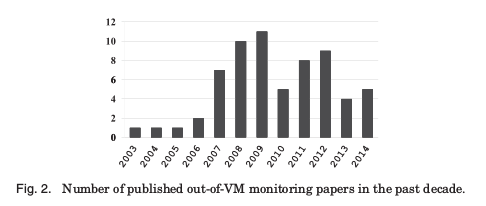
——指令、控制路径和堆栈调用:了解VM正在执行的指令、它所属的控制路径，以及调用上下文可以帮助out- VM监视器精确地理解来宾操作系统的当前状态。这些细粒度的控制状态通常可以通过仿真管理程序或单步的本地管理程序来观察。

——函数调用、系统调用、库调用和钩子:由于指令级监视通常会显著降低VM的执行速度，所以我们可以以粗粒度的粒度(例如，在函数调用执行级别，或者在某些系统调用、库调用或值得监控的钩子)中监视。

——进程、线程和执行上下文:当有上下文切换时，我们想知道哪个进程(线程)被切换到，因为控制流通常是线程特定的。对于给定的执行指令，我们还想知道是哪个进程或线程执行这个指令，执行是在用户空间还是内核空间。

1. 方法

在本节中，我们总结了为弥合语义鸿沟而提出的一般方法，以及这些方法所针对的具体应用程序。我们总共分析了从2003年到2014年在几个高度选择性的安全和系统场所发表的64篇论文(项目)。请注意，我们的目的并不是详尽地检查所有的文件，因为在其他场所发表的其他论文太多了。图2中显示了每年发布的关于out- vm监控主题的论文数量的分布情况。



总的来说，有五种主要的方法来弥补语义差距:手工方法(第4.1节)、调试器辅助方法(第4.2节)、编译器辅助方法(第4.3节)、二进制分析辅助方法(第4.4节)和用户辅助方法(第4.5节)，如图1所示，并在表2中报告。前四种方法是根据在构建一个out- vm监视器时实现人员所面临的约束来分类的。在高级别上，这些约束是基于是否访问客户操作系统数据结构、调试信息、源代码或二进制代码。此外，还可以选择完全避免语义上的差异，代价是牺牲VMI的安全优势。这种方法，即用户辅助的方法，修改客户操作系统内核，或者将一个程序放在内部，以便将信息传递给out- vm监视器。

所有这些方法在困难(当开发人员实现它们)和实用性方面都有所不同。例如，尽管一些手工方法使用了关于特定操作系统的详细信息，但是它们可能更容易实现。不同的方法在自动化程度上也有不同。根据定义，手工方法几乎不涉及自动化，而其他方法可能会使用不同数量的自动化来弥补差距，以及它们如何将实现从一个移植到另一个。在下面的部分中，我们将更详细地研究每种方法，并在第4.6节中对它们进行比较。

4.1手工方法

最直接的方法是在hypervisor层手动重构来宾操作系统抽象，只要我们有客人OS数据结构信息。每个单独的数据结构的布局、偏移量和字段类型，以及客户操作系统的数据结构之间的连接(例如，点到关系)，可以通过手工分析文档、调试信息、源代码或二进制代码来确定。然后，通过访问来宾操作系统的内存，hypervisor程序员可以使用对内核结构的详细理解来提取关于来宾操作系统状态的信息，这些信息可以用于各种内省目的。例如，Linux内核任务结构(即:(task\_struct)是在一个包含所有正在运行的进程的双链表中组织的。通过遍历这个列表，如果没有直接的内核对象操作(DKOM) [Petroni和Hicks 2007]在客户操作系统内部进行攻击，我们可以检索系统中的所有正在运行的进程。此方法还可用于检索有关存储在系统上的文件的磁盘映像的信息。只要能够访问磁盘映像，就可以使用文件系统结构的知识(例如，ext2)来遍历和访问文件。

虽然手工确定数据结构可能很繁琐，但这并不是一个困难的技术问题，因为人能够执行语义解释。

本质上，一个人的工作是解决语义鸿沟问题而不是依赖其他工具。然而，这种方法的最大限制是其极低的可伸缩性。为了支持大量不同的OSes，需要大量的努力来手工构建每个内核的数据结构知识，并重写相应的监控程序。

例子：令人惊讶的是，正如表II所报告的，许多out- vm监视器(64个中的33个)实际上采用了这种手动方法。例如，第一个手动方法，COPILOT [Petroni et al. 2004]，检索Linux内核文本和系统调用表，然后使用外部PCI设备验证它们的完整性(客户操作系统不可见)。关于OS内核数据结构的知识，如代码和系统调用表所在的位置，是手动重构的。特别针对2.4和2.6系列的Linux内核。它利用了Linux内核内存没有分页的事实，并且内核虚拟地址是线性映射的。某些Linux内核文本和数据结构(包括页表)位于虚拟内存中的特定不变位置，并被线性映射，从而允许检索页表，从而使数据结构的位置难以确定。一旦确定了位置，它就会基于内核数据结构知识重新构造客户操作系统语义，这些知识也是人工提取的。提取抽象的一些例子是内核task\_struct的组织和允许访问其他数据结构(比如mm\_struct)的方法。

COPILOT 方法之后，动态间谍软件分析(称为E KKYS3) [Egele et al. 2007]和P - ANORAMA [Yin et al. 2007]也手动重构了客户操作系统的抽象(例如，进程、文件、浏览器或内核对象)，以方便恶意软件分析。VM WATCHER [Jiang et al. 2007]采用客户观景技术来推断来宾操作系统的状态，并通过人工检索的内核数据结构知识来指导浇注。XEN ACCESS [Payne et al. 2007]是一个用于监控在Xen管理程序上运行的来宾操作系统的库。它为访问来宾操作系统的状态提供了一个框架，并减少了客户自省的工作量。

然而，该库本身是基于被监控内核的手动检索数据结构知识而开发的。另一种弥补语义鸿沟的手工方法包括O vers HADOW [Chen et al. 2008]， L YCOSID [Jones et al. 2008]， VMS COPE [Jiang and Wang 2007]， H OOK F INDER [Yin et al. 2008]等。在表II的第三列中列出了所有这些手动方法的列表。

4.2. Debugger-Assisted Approaches

由于数据结构知识通常在调试符号中可用，如果程序或操作系统内核是通过调试选项编译的，我们就可以在调试器的帮助下弥补语义上的差异。这种方法也非常简单。具体地说，如果内核是通过调试选项编译的，那么我们可以从用于分析内核转储或实时内存的现成调试器中获取调试信息。从这些信息中，我们可以进一步派生出guest OS抽象。注意，有时候我们可以直接从软件供应商那里获取调试符号(例如，微软确实向公众发布了Windows内核符号)，但是在这种情况下，我们仍然需要基于检索到的数据结构知识来开发内省例程，这本质上是一种手工方法。与手动方法类似，使用调试器使这种方法更加具体。此外，来宾操作系统内核还必须使用调试符号重新编译，这极大地限制了这种方法的实用性。

例子：没有多少系统使用调试器辅助的方法，并且总共只有5个这样的项目，如表2所示。具体来说，第一个调试器辅助的方法，LIVEWIRE [Garfinkel和Rosenblum 2003]，它是VMI概念的先驱，利用一个修改过的内核崩溃转储分析工具(本质上是一个调试器)来解释由调试信息编译的内核生成的内存转储的原始二进制数据。这种方法的优点是，调试器工具可以直接返回guest OS抽象，而不需要开发任何额外的代码。例如，使用crash工具[Anderson 2003]， LIVEWIRE使用内置命令来检索运行进程的列表(通过调用ps命令)、打开文件(使用文件命令)、实时内核对象(使用kmem命令)，以及每个进程的内核调用堆栈(使用bt命令)。使用从这些命令获得的语义信息，LIVEWIRE会确定来宾操作系统是否被破坏。

另外，NTROVIRT [Joshi et al. 2005]利用调试符号设置了“断点”来检查和执行系统管理程序层中的漏洞特定谓词。hook MAP [Wang et al. 2008]通过查询系统来解析内核符号。用于内核模块的内核文本和nm实用程序的映射文件。作为一个rootkit profiler, P O K E R [Riley et al. 2009]也使用调试标志编译Linux内核，并使用自定义调试器(即:，gdb)遍历内核对象并提供内核对象映射。这些对象映射用于促进对内核rootkit行为的理解。M OSS [Prakash et al. 2013]利用Windows调试符号对内核数据结构的字段进行了突变，用重复值导向的语义字段fuzzing来估计内核数据结构操纵攻击的严重程度。

4.3. Compiler-Assisted Approaches

在编译器辅助的方法中，使用(或修改)操作系统的源代码来自动构造一个数据结构图(或者用感兴趣的数据通知hypervisor)。该方法利用编译器的力量来推断在操作系统运行时，控制可能流向何处，利用OS源中的类型定义和上下文信息来确定通用指针的类型。

编译器还有助于自动化查找抽象的过程，或者如果原始源代码不包含这些抽象，那么内核就会被修改(或自动检测)来生成抽象，但是这种方法依赖于访问源代码。这种方法还需要对任何需要VMI解决方案的新内核重复分析。

例子。总共有8个项目使用了编译器辅助方法。这种类型的第一个方法，SBCFI [Petroni和Hicks 2007]，首先通过提取全局变量并从内核源代码创建一个内核变量类型的图来获得内核抽象的位置，然后使用这些数据生成代码，以遍历这些变量所能访问的指针。BCFI的目标是检查内核控制流的完整性(CFI)。为此，它从内核源代码生成控制流图。通过对最初生成的流图进行内核快照的定期检查，可以确定内核是否已经被破坏。通过验证内核使用的静态和动态指针，可以确定内核的CFI [Abadi et al. 2005]。静态指针包括存储在系统调用表中的函数指针或程序中使用的跳转表。动态函数指针包括在C中像void\*这样的通用指针和在select语句或结合中使用的间接函数调用，这些在执行过程中被确定。为了解决动态指针，SBCFI需要源注释来识别某些指针类型，包括通用指针。

KOP [Carbone et al. 2009]认识到S BCFI的局限性，并提出了利用内部过程点分析来计算所有可能的通用指针类型，解决了与模式匹配的类型歧义，并确定了具有内核内存池知识的动态数组的边界。MAS进一步改进了KOP，提供了一种新的内存遍历算法，该算法支持错误校正(也就是:，切断无效指针)并停止错误传播。

其他的编译辅助方法包括GIBRALTAR [Baliga et al. 2008]，它利用CIL [Necula et al. 2002]来获取数据结构定义，以及SIGGRAPH [Lin et al. 2011]，它使用gcc来派生内核数据结构不变量。L IVE DM [Rhee et al. 2010]利用内核源代码来区分不同的堆数据结构，然后构建内核堆图来帮助理解内核恶意软件行为。OS CK [Hofmann et al. 2011]从内核源代码中提取内存管理数据结构，然后使用一个灵活的线性扫描方法验证内核堆的类型安全属性。G ATEWAY用特殊的填充来重新编译内核源代码，这样就可以应用二进制改写器，并且可以在管理程序层中观察和执行控制流传输。

4.4. Binary Analysis–Assisted Approaches

在只有一个操作系统的编译的二进制文件可用的情况下，使用二进制分析辅助方法。它不需要访问任何特殊版本的操作系统、任何调试信息或任何定义的内核数据结构。相反，该方法分析编译的(被删除的)OS内核二进制代码，并重构客户操作系统抽象。

这种方法非常复杂，因为它不从任何其他来源获得抽象或定义。必须通过对寄存器值执行动态二进制分析、访问内存和执行指令，或者对原始代码和数据执行静态二进制分析来确定所有内容。然而，这使得VMI的最终结果更加实用，因为这可以由任何人来执行，并且不依赖于特定版本的操作系统。

对二进制代码的分析可以离线和在线进行。对于脱机分析，抽象是通过观察一个操作系统的可信版本在检查内核状态时执行的方式来开发的。然后，从这些观察所得的信息中，可以训练一个工具来帮助非vm监视器。另一方面，在线分析可以通过实时系统动态地进行。具体地说，它观察数据如何被访问，然后通过在线数据流分析和数据重定向重构抽象。

例子。有趣的是，六个项目使用了二元分析辅助方法。特别是，第一个二进制代码分析方法V IRTUOSO [Dolan-Gavitt et .2011a]表明，可以从执行本机检查命令和操作系统内核中提取指令，然后将这些跟踪转换为内省程序。使用V IRTUOSO时涉及三个不同的阶段:培训、分析和运行时环境生成。培训包括对一个内省程序(例如ps)进行多次跟踪。分析阶段包括查看跟踪中的指令，以识别诸如malloc之类的不需要的指令。然后使用所需的指令来生成可执行文件。生成的程序可以在hypervisor层的来宾操作系统之外独立运行，以执行内省。

当V IRTUOSO涉及到离线训练阶段，生成覆盖问题的跟踪时，V MST [Fu and Lin 2012, 2013a]展示了一种在线的二进制代码重用方法，而无需经过培训，使本机检查程序能够自动成为内省程序。对于vmst，不需要进行任何培训，因为内省兴趣的数据会被本机检查过程自动识别和访问。(Fu和Lin 2013b)本质上使用了V MST的相同的二进制分析技术，但将其扩展到客户OS可写的内省。B缺乏S HEEP [Bianchi et al. 2012]避免了复杂的二进制代码分析，而是使用一种新的内存比较方法来识别二进制数据中的内核rootkit攻击点。TZB [dolan gavitt et al. 2013]表明，我们还可以使用挖掘方法来识别操作系统内核和二进制执行中的应用程序中的hook点。结合从virtuoso的基于训练的方法和V MST的在线内核数据重定向方法，H YBRID B RIDGE [Saberi et al. 2014]通过训练memoization和解耦执行提高了V MST的性能。

4.5. Guest-Assisted Approaches

与其他方法不同的是，这种方法与其他方法不同，它避免了语义上的差异，代价是可能牺牲了在VM之外工作的安全优势(除非采用特定的特殊照顾)。在这种方法中，程序被放置在来宾操作系统中，以提供关于系统的监视器信息。它还可能涉及到将钩子添加到客户端，或者简单地运行它内部的监视系统。由于程序或钩子在操作系统内部运行，因此它可以完全访问在VM外部移动监视器时通常会丢失的所有抽象。然而，这也可能使它容易受到与主机监视器相同的危险，因此，如果系统受到损害，并没有特别注意保护内部组件，那么它向基于hypervisor的监视器提供的所有信息就会变得不可靠。

与其他方法相比，这种方法更简单;它避免了语义鸿沟问题，它可以轻松地处理任何可以运行在客户端程序的操作系统，并且它可以访问所有需要的客户系统信息。然而，使用这种方法所引起的安全问题与最初导致vm外监控的问题是一样的。因此，实现通常会寻求使用hypervisor来为它们的in-VM监控代码提供额外的保护。

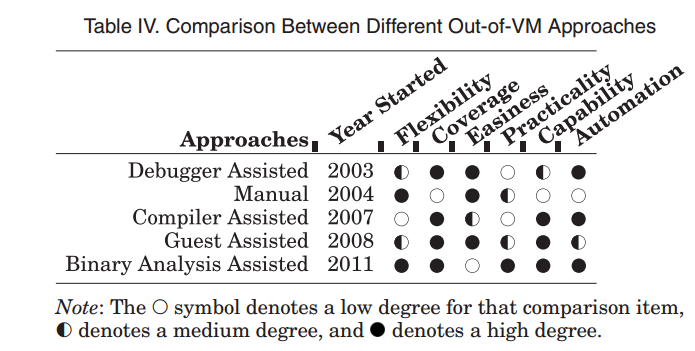
例子：有12个项目使用了guest辅助方法。先驱工作L ARES [Payne et al. 2008]，在客户虚拟机中插入钩子，并通过支持主动监视的目标，使用hypervisor来保护其客户端组件。与被动监视不同，主动监视需要内核事件的相互作用。因此，它要求在来宾操作系统内部执行监控代码，这就是为什么它本质上导致了在guest VM中插入某些钩子的解决方案。钩子用于触发事件，这些事件可以通知hypervisor或将执行重定向到外部VM。更具体地说，LARES设计包含三个组件:一个guest组件、一个安全VM和一个管理程序。hypervisor可以通过内存隔离保护来宾VM组件，并充当来宾VM和安全VM之间的通信组件。安全VM用于分析事件，并采取必要的行动来防止攻击。

在另一项工作中，安全的In - vm monitoring (S LCL) [Sharif et al. 2009]将监控代码放置在来宾操作系统中，并使用系统管理程序的功能来隔离在客户操作系统中由监视器所占用的区域。H OOK S AFE [Wang et al. 2009]插入了一个在客户端短时间内的内核模块来设置钩子indirection层，然后在管理程序层执行hook保护。H UKO [Xiong et al. 2011]将一个受信任的驱动程序插入到来宾操作系统中，以通知系统管理程序关于内核内存的分配和回收，并且还将每个内核页面的所有者主体标记为hypervisor，以强制执行强制访问控制。进程移植(GDXJ) [Gu等人，2011]利用了客户端流程的执行上下文，并用检查或管理工具替换其代码来执行内省和恢复。Process outgrafting (SWJX) [Srinivasan et al. 2011]将一个内核模块插入到一个guest VM中，并将监视过程的系统调用执行重定向到guest VM进行行为分析。S YRINGE [Carbone et al. 2012]使用了一种新的客户辅助功能调用注入和局部引导技术来验证客户代码的执行。而不是将数据访问(如V MST)、H ypershell(Fu et al . 2014]和W CLM(吴et al . 2014年)重定向的执行系统调用从安全虚拟机到来宾VM,协助客人VM辅助过程,缩小语义鸿沟。

更广泛地说，需要内核源代码修改(或插装)来主动向管理程序报告状态更改的方法也是客户辅助的。例如，OSVM [Oliveira和Wu 2009]修改了客户内核源代码，为内核对象和对象添加Biba完整性级别，然后在管理程序层中协作执行Biba完整性模型。KRUISER [Tian et al. 2012]修改了内核源代码，尤其是堆管理数据结构来添加canaries，然后在hypervisor层验证canaries以检测堆溢出。SENTRY [Srivastava和Giffin 2012]修改了内核源代码，以划分内核数据结构布局，这样它就可以检测并防止对由hypervisor保护的关键内核数据结构进行恶意修改。

* 1. 总结和比较

正如前面所讨论的，不同的方法有不同的约束、优点和缺点。下面我们将更详细地比较它们。我们提供表三中每一个指标的定义，并将它们与表四进行比较。具体来说，我们比较了每种方法的灵活度(方法的灵活性;更少的限制,更灵活),覆盖(多少抽象可以重建,多少来宾操作系统状态可以观察到),从容(是多么容易实现这个方法),实用性(可用和可采用的这是真实世界的应用程序),和自动化(多少可以自动完成,而不是用手)。我们根据他们第一次出现的年份订购这些方法。



调试器辅助。调试器辅助方法使用调试工具帮助提供关于来宾操作系统的语义信息。由于它依赖于对调试信息的访问，所以它只有中等的灵活性，但是从调试信息中获得的大量数据也提供了很高的覆盖率。对于这种方法(高度的易用性)来说，不支持vm的工具相对容易编写，因为它可以使用现有的OS调试工具(高自动化)。例如，LIVEWIRE直接重用可用的内置命令来检索内核语义信息。通过调试信息，还可以使用监视工具来分析获取系统信息后的VM快照。然而，获取调试信息可能涉及在调试模式中重新编译源代码，或者在源代码不可用的情况下运行操作系统的特殊调试版本。期望用户运行一个操作系统的调试版本并不总是可行的，这样的操作系统常常需要从内核源代码中重新编译。由于这些缺点，这种方法的实用性较低。

手工。无论有什么资源可用或约束存在，都可以采用手动方法，因此它具有很高的灵活性。它可以分析源代码或调试信息，甚至可以反向设计二进制代码来确定感兴趣的内核数据结构的形式和位置。但是，在这种情况下，这种方法的覆盖率很低，因为每个数据结构都必须手动识别。或者，数据结构的可用定义可能已经存在。非vm工具必须手动地从头开发，从而导致自动化程度低。然而，尽管它很乏味，但是创建这些工具并不困难(高度的轻松)。这种方法具有适度的实用性，因为它可以在大多数情况下由任何人进行，但是它的低扩展性可能会妨碍它的实际使用。

编译器辅助。编译器辅助的方法需要访问操作系统的源代码，这意味着它具有低灵活性和低实用性。由于它试图通过使用源上的编译器来跟踪潜在的程序流和抽象的原因，从而实现自动化，所以它具有很高的自动化程度。一些数据结构可以通过跟踪源中的静态信息自动生成，而有些数据结构也依赖于运行时的某些值。除了静态数据的完整视图之外，一些实现还能够跟踪动态数据，因此这种方法具有很高的覆盖率。然而，编译器传递并不容易实现，动态数据也很难确定，因此它具有适度的轻松性。

客人协助。在来宾操作系统中通常会安装一个辅助程序，或者修改现有的系统，将有用的信息导出到outof-VM监视器中。由于该方法避免了语义鸿沟问题，因此具有较高的易用性。它具有适度的灵活性，因为它需要客户操作系统的授权来安装辅助软件或重新编译内核。这些要求导致了适度的实用性。如果需要，它可以覆盖几乎所有东西(高覆盖率)，因为它可以访问所有来宾操作系统抽象。自动化不像纯vm解决方案(中等自动化)那么高，因为它需要在来宾操作系统中安装软件或重新编译客户操作系统内核。

二进制分析辅助。当只有二进制代码可用时，使用二进制分析辅助方法，因此具有较高的灵活性。它试图通过静态分析或动态分析来确定抽象，通过观察编译代码运行时的行为。因此，它可以用来有效地内省闭源OSes。二进制代码分析可以在实时系统中实时执行，也可以用于训练工具以供以后使用。这种方法很难实现(低的易用性)，但是它具有很高的实用性，因为它可以执行许多outof-VM监视功能。能够从二进制代码中忠实地推理出抽象，甚至重用某些遗留二进制代码，它还具有高覆盖率和高自动化程度。

1. APPLICATIONS

5.1检测

任何监控系统的第一步是收集状态，以允许M做出决策(回想一下(1)中的定义)。观察到的状态可以是内核级或用户级，可能是良性的或恶意的。任何观察到的更改都可以分为代码修改或数据修改。

在观察系统中的良性或恶意行为后，我们可以利用这些观察所得的信息来侦测攻击。基于攻击发生的地方，我们将它们分为内核级或用户级攻击。此外，基于攻击是否篡改代码或数据，我们将它们归类为代码攻击或数据攻击。此外，虚拟机监视器也可以纯粹用于行为分析(例如，恶意软件分析)和法医调查(在攻击发生后)。在下面的段落中，我们将回顾这些特定的应用程序。

内核级别。内核级攻击具有管理特权，可以篡改操作系统内核和应用程序进程。内核rootkit是此类攻击的示例。检测内核异常通常需要构建一个捕获良性内核行为的模型。任何来自良性行为的偏差都可以被标记为攻击。一些系统关注于观察良性的内核数据结构。特别是SBCFI [Petroni and Hicks 2007]， KOP [Carbone et al. 2009]，和MAS [Cui et al. 2012]静态分析内核源代码，构建良性内核数据结构图。OSCK [Hofmann et al. 2011]也首先使用静态分析来提取良性内核的堆管理数据结构。LIVEDM [Rhee et al. 2010]使用动态分析来构建内核堆图。DGSTG [Dolan-Gavitt et al. 2009]应用fuzzing技术来获得内核数据结构的值不变特征，SIGGRAPH [Lin et al. 2011]利用从源代码提取的良好快照和内核数据结构来获得图形签名。

另一种选择是观察由内核恶意软件(误用检测)引起的恶意行为。HOOKFINDER [Yin et al. 2008]观察到hook行为如何受到内核恶意软件的影响。KTRACER [Lanzi et al. 2009]从内核rootkit中提取恶意行为; POKER [Riley et al. 2009]和RKPROFILER [Xuan et al. 2009]也动态观察了rootkit在OS内核中的执行情况。这些信息通常有助于获得有效的签名以捕获攻击。

用户级别。还有许多用户级的攻击，比如病毒、蠕虫或间谍软件，它们会在用户级别上损害系统。要确定一个进程是否是恶意的，监视器必须分析其行为。虽然in-VM解决方案可能会使用像strace这样的东西来跟踪in-VM系统调用行为，但out-of-vm解决方案也可以实现这样的功能。例如，VMSCOPE [Jiang和Wang 2007]在虚拟机监控程序层解释系统调用事件，而过程outgrafting (SWJX) [Srinivasan et al. 2011]使用双vm来跟踪良性进程的系统调用行为。EKKYS [Egele et al. 2007]通过分析浏览器插件来检测间谍软件。PANORAMA [Yin et al. 2007]分析了整个系统的信息流来分析恶意软件造成的恶意行为。

代码修改。代码修改检测涉及到对内核代码更改的识别。通过对内核文本进行哈希处理，可以进行周期性的重新计算，以确定内核文本是否有变更的livewire [Garfinkel和Rosenblum 2003]和COPILOT [Petroni et al. 2004]使用哈希来检测任何未经授权的代码修改。SBCFI [Petroni and Hicks 2007]， PATAGONIX [Litty et al. 2008]， HIMA [Azab et al. 2009]， RKPROFILER [Xuan et al. 2009]， POKER [Hofmann et al. 2011]， are [Hofmann et al. 2011]， are [Moon et al. 2012]， KI-MON [Lee et al. 2013]也有这种强大的代码修改检测。相比之下，一些系统在代码修改中提供了非常弱的检测，比如ANTFARM [Jones et al. 2006]，它只是一个框架，因此用户必须扩展它以获得足够的检测。

数据修改。数据修改包括对关键内核对象(如函数指针和系统调用表)的更改。数据修改可以通过基于签名的方法来检测。还可以遍历内核内存，以发现包含内核状态信息的内核数据结构，例如，遍历task\_struct获取进程列表或模块列表，以获得加载的内核模块列表。PFWA、VMWATCHER、KOP、DGSTG、POKER、RKPROFILER、LIVEDM、SIGGRAPH、VIRTUOSO、VMST、MAS、BLACKSHEEP、SENTRY和KI-MON都是检测内核数据修改的系统示例。其中，PFWA、KOP、POKER、RKPROFILER、LIVEDM、MAS、KI-MON都有强大的数据检测功能，因为它们甚至可以检测动态内核对象的修改。

除了直接修改函数指针或系统调用表外，还有另一种类型的攻击，可以通过利用缓冲区溢出之类的内存错误来触发。为了检测这种类型的攻击，KRUISER [Tian et al. 2012]对内核源代码进行了重新编译，并在hypervisor层中检测是否存在任何堆溢出。

取证与分析。法医学包括对一个系统如何在事后遭到攻击的调查分析。任何法医调查的第一步是收集被感染系统的状态。基于签名的扫描(例如，DGSTG和SIGGRAPH)或其他系统，如TRAILOFBYTES [Krishnan et al. 2010]， VIRTUOSO [Dolan-Gavitt et al. 2011a]， SHELLOS [Snow et al. 2011]，以及VMST [Fu and Lin 2012]都可以用于对受害者VM进行法医分析。

虚拟机监视器也被用于恶意软件分析。例如，ANUBIS使用VMM来观察恶意软件执行的系统调用和api，它还支持其他高级程序分析，比如切片和污染分析。ETHER为分析恶意软件提供了一个高保真的环境。EKKYS、全景图和V2Ecan也会在这样的场景中使用，因此恶意软件的profiler，如KTRACER、POKER、RKPROFILER和MOSS也可以使用。

5.2 预防

任何安全机制的最终目标都是防止攻击。这通常需要主动监控和拦截攻击的能力。类似于我们在5.1节中对检测应用程序的分类，我们还将现有的预防应用程序分为内核级和用户级保护，以及代码和数据保护。

内核级别。除了稍后描述的内核代码和数据保护之外，还有几个项目的目标是将内核模块与内核的其余部分分开，因为内核模块通常是操作系统内核的脆弱点。特别是，内核模块的隔离常常为内核模块分配特殊的内存地址空间。验证是为了确保内核模块不执行对内核区域的任何未授权访问。这有助于保护内核不受恶意模块的影响。HIMA [Azab et al. 2009]和HUKO [Xiong et al. 2011]使用硬件辅助的分页技术从内核中分离出内核扩展。GATEWAY [Srivastava和Giffin 2011]以类似的方式隔离设备驱动程序，同时在驱动程序代码中对拦截进行间接调用。

用户级别。与内核模块隔离类似，进程隔离是通过在一个隔离的地址空间中运行一个进程来实现的，该进程受一个hypervisor的保护，不受未经授权的修改。实现包括SLCL [Sharif et al. 2009]和HIMA [Azab et al. 2009]。虽然在SIM中使用这种技术来保护监视过程，但是可以将这个想法扩展到监视和隔离普通流程，如HIMA中所示。

代码的保护。代码保护包括保护内核文本，使内核文本内存区域不可更改。代码保护下的另一项技术是加密来宾操作系统内存，以防止未经授权的访问。例如，OVERSHADOW [Chen et al. 2008]对客户的VM页面进行了加密，这样就不可能对VM进行篡改了。在另一种方法中，SECVISOR [Seshadri et al. 2007]使用硬件辅助的内存保护，并验证代码的完整性。PATAGONIX [Litty等人，2008]在允许执行之前验证代码的真实性，HIMA也是如此。这些代码保护非常强大。

数据保护。数据保护旨在保护内存中的数据结构、系统调用表和内核对象，包括静态和动态数据。保护动态数据包括遍历内核堆以发现动态数据对象。HOOKSAFE [Wang et al. 2009]将所有的内核钩子移动到一个单独的地址空间，并防止内存访问写入该区域。虽然HOOKSAFE完全保护内核钩子，但它不保护其他内核数据。这样就有了适度的完整性，如表2所示。NICKLE [Riley et al. 2008]， LARES [Payne et al. 2008]， OSVM [Oliveira and Wu 2009]， SENTRY [Srivastava and Giffin 2012]都使用内存访问限制来保护内核数据。OVERSHADOW [Chen et al. 2008]对页面进行加密以击败任何试图查看关键数据的恶意进程。因此，它提供了强大的保护，因为攻击者修改数据非常具有挑战性。

5.3恢复

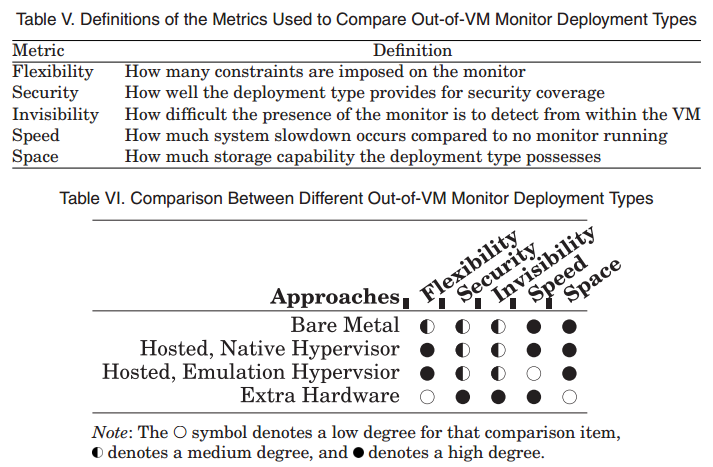
当有问题时，系统自动从攻击中恢复的能力有助于减少人类干预的需要。一旦我们发现了攻击，如果有可能直接修复它，系统可以保持在线状态，而不会中断服务。这对于需要高可用性的服务尤其有用。

然而，如表二所示，即使在可修复的攻击中，攻击恢复的注意也大大减少。

VICI [Fraser等人2008]对基于vmm的客户操作系统恢复进行了初步尝试。它执行代码修复，在检测到修改过的内核代码后，将内核文本恢复到原来的状态。它还可以使用回滚机制，该机制包括定期在运行时对VM进行快照。这些快照可以用于在发生攻击时将VM恢复到已知的良好状态。PMPDFGJ [Paleari et al. 2010]会自动生成恶意软件补救程序，使用模拟虚拟机进行恶意攻击，然后执行它们将被感染的机器恢复到良好状态。GDXJ [Gu et al. 2011]能够将一个清理过程注入到客户操作系统中，因此具有一些恢复功能。EXTERIOR (Fu和Lin 2013b)通过跨机器检查修复有限数量的内核对象，比如系统调用表。我们必须注意到，除了回滚，现有的任何工作都无法实现高度的恢复，因为肯定有一些攻击是无法修复的，并且需要重新安装或从较早的快照恢复系统。

1. 部署

在实现一个out-of-vm的监视器时，另一个重要的考虑是它将在哪里部署。决定部署哪一层时考虑的因素包括灵活性、安全性、可见性、速度和空间限制。关于这些指标的详细定义见表V。每种部署类型都有优点和缺点，我们已经将不同的部署方法归类到表VI中。注意，所有的hypervisor部署都有很高的空间，因为它们没有使用有限的内存和适度的不可见性，这是由于系统管理程序检测技术(如时间分析)所造成的。下面，我们将更详细地讨论它们。



裸金属。一种部署方法使用裸金属管理器，如Xen。这种部署提供了适度的灵活性和改进的性能(高速)，因为客户机代码直接在硬件上运行。它还允许监视特权指令和硬件事件。与其他部署方法(中等安全性)相比，这种部署缺少安全覆盖的程度，但是，它只在某些内核事件发生时(例如，某些中断)看到客户操作系统的快照视图。典型的实现包括SECVISOR [Seshadri et al.2007]， ETHER [Dinaburg et al. 2008]， XENACCESS [Payne et al.2007]， MAVMM [Nguyen et al. 2009]， HUKO [Xiong et al. 2011]等。

主机,本机虚拟机监控程序。另一种部署方法是使用托管的本地管理程序，比如KVM。这种类型的hypervisor提供了使用主机操作系统抽象来实现hypervisor服务的高度灵活性。它还允许对客户代码(高速)进行全速执行，尽管它仍然与裸金属管理程序(中等安全性)共享相同的快照视图。使用这类管理程序的实现包括SLCL [Sharif et al. 2009]、HOOKSAFE [Wang et al. 2009]、GATEWAY [Srivastava and Giffin 2011]、OSCK [Hofmann et al. 2011]、SHELLOS [Snow et al. 2011]和SWJX [Srinivasan et al. 2011]。

主机,仿真程序。与前面的两种方法不同的是，部署了一个托管的模拟管理程序，它提供了监控服务的完整软件控制能力。这种类型的部署的典型管理程序是QEMU或VMware工作站和VirtualBox的早期版本。仿真管理程序允许拦截所有的指令，这提供了高灵活性和良好的安全覆盖。然而，由于在软件层上模拟指令的开销，这种技术会受到性能下降(低速)的影响。这种减速也导致了适度的安全性，因为它增加了定时攻击的可能性。使用这种部署方法的实现包括LIVEWIRE [Garfinkel和Rosenblum 2003]、VMWATCHER [Jiang et al. 2007]、HOOKMAP [Wang et al. 2008]和VMST [Fu and Lin 2012]。

此外，从表2中可以看出，有些系统对任何特定类型的管理程序都是不可知的，并且可以部署到其中的任何一个。这对于基于记忆的监测系统来说尤其如此，例如SBCFI [Petroni和Hicks 2007]， GIBRALTAR [Baliga et al. 2008]， KOP [Carbone et al. 2009]， DGSTG [Dolan-Gavitt et al. 2009]， SIGGRAPH [Lin et al. 2011]， VIRTUOSO [Dolan-Gavitt et al. 2011a]和MAS [Cui et al. 2012]。

额外的硬件。更安全的方法是在外部硬件上部署out-of-vm的监视。通过使用单独的设备部署监视服务，攻击的风险大大降低。主要缺点是这种部署方法的灵活性较低(灵活性较低)。许多外部设备可能也有有限的内存，这限制了监视器(低空间)的功能。在这种类型的部署中涉及的外部硬件包括PCI卡，它已经被用于COPILOT [Petroni et al. 2004]、PFWA [Petroni et al. 2006]和HYPERCHECK [Wang et al. 2010]。由于与其他部署相比，额外的硬件可能并不总是有足够的空间，但是它可以提供相对较高的安全性、高的可视性和高速。此外，最近还设计了特殊的额外硬件组件，用于与系统总线集成，以使用总线监视技术(如Lee et al.[2013]和Moon et al.[2012])来验证内核完整性。

1. 讨论和未来的方向

虽然已经有大量的研究在vm外监测，但仍有很多的机会进行未来的研究。在本节中，我们将讨论剩余的研究问题，并阐明未来的方向。

**7 . 1 . I m p rov i n g Hy p e r v i s o r S e c u r i t y**

在执行out-of-vm监视时，重要的是在设计工具来派生客户操作系统抽象时确定要信任什么，因为内省依赖于重构数据的完整性来理解客户操作系统的状态。Jain等人[2014]最近强调，许多自省技术会对客户操作系统做出假设，这样就可以允许被攻击的或恶意的客户操作系统来绕过检测。这些假设包括在训练过程中操作系统是良性的。

在执行out-of-vm监视时，重要的是在设计工具来派生客户操作系统抽象时确定要信任什么，因为内省依赖于重构数据的完整性来理解客户操作系统的状态。Jain等人[2014]最近强调，许多自省技术会对客户操作系统做出假设，这样就可以允许被攻击的或恶意的客户操作系统来绕过检测。这些假设包括在训练过程中操作系统是良性的，可以通过自动化过程(非手动方法)来学习抽象，攻击的时间足够长，可以检测到(如果只是周期性地检查系统状态)，并且在部署中使用白名单模块是可行的。例如，如前所述，本地管理程序的快照视图可能会错过非常短暂的攻击。根据所采取的方法，假设的有效性各不相同。对于未来的out-of-vm解决方案来说，清楚地说明它们的安全性假设和预期用例来更好地定义它们的功能是非常重要的。

另一方面，虚拟机监视器的安全性依赖hypervisor的安全性。正如第2.2节中所讨论的，与in-vm解决方案相比，out-of-vm的攻击向量要少得多，但是最近有一些攻击目标是破坏虚拟机程序，并更隐蔽地执行恶意功能。著名的攻击包括Blue Pill [Laurie and Singer 2008]， smm的rootkit [Embleton et al. 2008]，基于pci的rootkit [Heasman 2006]， SubVirt [King et al. 2006]， SubXen [Wojtczuk 2008]，和活性VMI [Fu et al. 2013]。

为了克服对hypervisor的威胁，提出了几个不同角度的解决方案。他们要么将out-of-vm的监控层压入逻辑隔离的硬件，提高系统管理程序的安全性，减少管理程序的代码库和攻击面，或者强制操作系统验证自己。更具体地说:

——向下推一层:由于管理程序层可以确保在它上面一层的客户操作系统的执行，我们可以通过将监控层推入到硬件中来保护hypervisor的执行。最近的努力包括HYPERSENTRY [Azab et al. 2010]、CLOUDVISOR [Zhang et al. 2011]、HYPERCOFFER [Xia et al. 2013]和MGUARD [Liu et al. 2013]。HYPERSENTRY利用x86系统的系统管理模式(SMM)来监控硬件状态。SMM可以访问主机内存和CPU寄存器来检查运行系统的完整性。CLOUDVISOR在商品管理程序的底层引入了一个很小的安全监视器，它使用嵌套虚拟化来保护VMMs和vm。为了保护来宾VM的隐私和完整性，HYPERCOFFER通过使用内存加密和完整性检查在来宾VM和hypervisor之间引入了一个VM- shim机制。使用drop-in内存控制器，MGUARD监视内存流量，以防止对hypervisor代码进行任何非法修改。

——改进hypervisor代码:另一个解决方案是改进管理程序代码。正式的方法和类型验证已经被应用于证明没有某些弱点，如SEL4 [Klein et al. 2009]和VERVE [Yang and Hawblitzel 2010]。此外，由于要消除系统管理程序代码中的软件漏洞，HYPERSAFE [Wang和Jiang 2010]对管理程序代码进行测试，然后运行时验证其完整性。最近，XMHF [Vasudevan et al. 2013]提出了一个可扩展的模块化管理程序框架，该框架可以验证hypervisor的内存完整性，并确保在较低权限级别上运行的软件不会对hypervisor内存进行修改。

对管理程序的特权:第三个解决方案是最小化系统管理程序的可信计算基础(TCB)。NOVA [Steinberg和Kauer 2010]实现了一个瘦的金属管理程序，它将虚拟化支持移动到用户级，从而导致至少一个比现有系统更小的TCB大小。NOHYPE [Keller et al. 2010;Szefer et al. 2011)通过严格划分硬件资源和重写来宾操作系统内核，消除了hypervisor层。HYPERLOCK [Wang et al. 2012]创建了一个单独的地址空间来限制hypervisor的执行。DEHYPE [Wu et al. 2013]直接将大部分管理程序代码与客户操作系统内核不进行任何合作。

Paraverification:与半虚拟化概念相关的另一种解决方案，即使面对潜在的恶意或折衷的操作系统，也要考虑验证。这个解决方案需要对操作系统进行细微的修改，因为它必须被设计成向hypervisor报告其活动。即使OS不能被信任，如果它被迫报告自己的行为，hypervisor可能会检查不一致。在这个领域还需要做一些工作来确定这种方法的潜力[Jain et al. 2014]。

7.2提供高保真的hypevisor

与虚拟机攻击和虚拟机防御的“猫和鼠标”游戏类似，通过检测虚拟机监控程序的执行情况，攻击者可以改变他们的行为，以避开0ut-of-vm的监视器。例如，通过执行指令执行的计时分析，可以很容易地检测到仿真管理程序(因为它的速度很慢)[Garfinkel和Rosenblum 2003;江et al . 2007]。另外，通过观察内存和CPU寄存器值的异常变化，恶意软件可以检测到某些hypervisor的存在[Dinaburg et al. 2008]。

为了对抗这种监控意识的攻击，一种可行的方法是开发有效的检测技术(例如，行为差异[Balzarotti et al. 2010])或秘密地对抗恶意软件(例如，Vasudevan和Yerraballi[2006])，而另一个方向是提供高保真的管理程序。Ether[Dinaburg et al. 2008]已经迈出了消除恶意软件可检测的副作用的第一步，导致了高度透明的监控系统。POKEEMU [Martignoni et al. 2012]利用路径提升和符号执行交叉vlidaate实现仿真虚拟机监控程序，从而提高了它们的保真度。TXINTRO [Liu et al. 2014]使用硬件事务性存储器(HTM)对关键内核数据结构进行动态监控。使用HTM，它可以实现客户机vm的并发、及时和一致的内省。尽管已经取得了一些进展，但是仍然需要付出巨大的努力才能达到拥有高保真度和高度透明的管理程序的阶段。

7.3完整内存监控

尽管hypervisor可以访问客户操作系统的所有物理内存，但遗憾的是它所观察到的内存是不完整的。具体地说，如果某个页面未被使用，且操作系统正在耗尽内存，它通常会将页面从物理内存掉到磁盘，从而导致管理程序不可见的页面。然而，Linux内核不交换内核内存，而Windows内核是这样做的。要对整个来宾操作系统(包括内核和它的运行应用程序)有一个完整的视图，我们必须解决内存问题。

尽管客户OS辅助的方法，LARES [Payne et al. 2008]可以通过注入一个代理来访问虚拟地址来解决这个问题(它们将会被来宾操作系统自动交换)，我们没有意识到任何解决这个问题的纯粹的out-of-vm解决方案。直观地说，它需要遍历客户机操作系统页面交换数据结构以及从磁盘内省(在第7.4节中讨论)的额外进展。如果今后在这方面有更多的研究努力，也就不足为奇了。

7.4 完整磁盘监控

有趣的是，据我们所知，目前所有的out- VM监视器都主要关注内存，除了XENACCESS，它提供了一个用于访问VM磁盘映像的库，而VMWATCHER也检查磁盘文件，在访问VM磁盘数据(包括交换内存)时，很少有光线。然而，磁盘数据确实包含了关于操作系统的许多有价值的信息，因为现在几乎所有的in-VM反病毒软件实际上都扫描磁盘和内存来查找病毒。

有人可能会说，要对客户操作系统的磁盘数据进行内省，这是微不足道的，因为我们可以直接将其挂载到第二台计算机上，并使用本地磁盘扫描工具来分析其状态。不幸的是，如果磁盘有一个已知的文件系统，如XENACCESS和VMWATCHER中所显示的，并且如果文件没有被加密，那么这只会起作用。如果客人的操作系统使用一个私人未知的文件系统,或者是文件加密等底层文件系统使用全磁盘加密,这往往是一种在今天的云上常见的公开vms的做法,这就带来了重大挑战,因此需要更多的研究来解决这个问题。

* 1. 可比较的性能指标

不幸的是，在我们的研究中，我们发现，许多现有的解决方案，对vm的内省，都有无法与其他解决方案相媲美的性能评估。通常，解决方案使用各种计算、基准或常见任务来确定其性能开销，但是没有使用标准的测试集。此外，除非使用相同的硬件和底层软件(例如，hypervisor或OS)，否则相同测试的结果就不那么有用了。除了时间开销之外，还需要能够比较代码大小以确定解决方案添加了多少额外的攻击表面，而不是所有的工作都提供这些信息。因此，对于未来的工作来说，创建某种形式的标准测试指标可能是有用的，这可以让研究人员直接比较解决方案。

7.6不只是只读的内省

几乎所有过去对vm的监视的研究都是只读的，主要是避免对客户操作系统进行更改。这是合理的，因为除非VMM清楚地知道哪些客户地址可以安全地修改，并且在何时可以安全的写入，否则它就有可能破坏内核状态，甚至破坏内核。

由于跨越语义鸿沟的巨大进步，最近的实现几乎包含了关于来宾操作系统的语义信息，因此我们有机会超越传统的只读自省。考虑到虚拟机监控程序层的强大功能，可以比虚拟机安全系统更高级的客户操作系统修改。正如外部所演示的，可以执行一些操作，比如配置内核参数、更改IP路由表、甚至杀死恶意进程。

可写VMI最吸引人的地方是它能够立即响应和防止攻击，而不需要客户操作系统或客户操作系统应用程序的任何合作。它不需要来自客户操作系统的根特权来执行根级别的操作，比如killing一个rootkit创建的隐藏进程，或者运行rmmod来对抗隐藏的恶意内核模块。考虑到有很多的VMI实现(例如，Dolan-Gavitt等[2011a]、Fu和Lin[2012]、Garfinkel和Rosenblum[2003]、Jiang et al.[2007]和Jones等[2008])，可以无缝地合并可写的VMI。鉴于可写VMI的激励和语义鸿沟的缩小，我们期望在这个方向上有更多的研究。

7.7

目前，大多数的out-osf-vm监视器都对来宾操作系统内核的行为感兴趣，它仅仅是监视器之上的一层。作为这个概念的一个进展，一个自然的问题是为什么不监视两个层，以观察在来宾操作系统中运行的应用程序级别的行为?事实上，这是可能的。最近的一项研究，TZB [Dolan-Gavitt等人，2013]，已经证明通过高度自动化的程序分析，通过识别Web浏览器的hook点，我们能够通过主动监视(例如Payne等[2008])在hypervisor层观察Web浏览器的行为。我们相信未来将会有更多的努力来推动对客户操作系统应用程序的监控，比如Web浏览器。

另外，目前的out-of-vm监控主要集中在传统的平台上，移动平台上有迫切的需求。早期的一个尝试是DROIDSCOPE [Yan and Yin 2012]，它监控Android内核和在Dalvik虚拟机中运行的应用程序。考虑到更多的移动设备和虚拟化支持，以及在移动操作系统内核中更隐蔽的恶意软件(Zhou和Jiang 2012)，更多的虚拟机监视器将被开发为内省移动设备也就不足为奇了。例如，最近的一项工作，tzr - rkp [Azab et al. 2014]，已经演示了使用虚拟机监控程序来为Android平台提供内核保护工具。

1. 相关工作

最近，已经有几项工作致力于将VMI的知识系统化，特别是从取证和安全云计算的角度。特别是，库恩和泰勒[2011]回顾了法医学的特定目的的内省。他们还讨论了VMI设计和实现的各种方式，并概述了VMI的应用，而不是不同的实现技术。

Denz和Taylor[2013]从云安全的角度总结了内省。他们首先关注云中的vm的威胁模型，包括当前的检测和预防技术。然后他们讨论了安全的管理程序，各种强化管理程序的尝试，以及云的恢复力。最后，他们试图对攻击表面的大小和性能开销进行比较，但是他们只能比较开源系统的代码大小，并且一些系统缺少性能开销数据。

Jain等[2014]也是一个类似的工作，它也关注VMI中的语义鸿沟问题。然而，他们的论文侧重于信任、安全问题和攻击，而我们的工作则对现有方法进行了更系统的研究、分类和总结，以及对潜在应用程序和部署策略的概述，我们认为这对试图为该领域做出贡献的研究人员是有用的。

1. 结论

从第一天出现以来，out-of-vm监控一直是一个吸引人的替代方案。在过去的十年中，已经进行了大量的研究，以扩展out-of-vm监视器的安全应用程序，使其更加实用和高效。在这篇文章中，我们通过检查和分类不同的方法以克服语义鸿沟，并开发各种安全应用程序，从而使我们在虚拟机监控领域的知识系统化。具体来说，我们回顾了现有的方法如何跨越语义鸿沟，同时处理不同的约束，如灵活性、覆盖率、实用性和自动化;他们如何开发不同的监测系统;以及如何应用和部署监测系统、out-of-VM监控的未来,我们相信,有几个有趣的途径探索,如保护管理程序本身,拥有完整的内存和磁盘监视和发展更多的自省能力包括可写的VMI kerneland用户级程序,以及为移动平台扩展out-of-VM显示器。