基于自适应机制的虚拟机进程实时监视方法

崔超远[1] 李勇钢[1),2)] 乌 云[3)] 孙丙宇[1)]

[1)](中国科学院合肥智能机械研究所 合肥 230031)

[2)](中国科学技术大学 合肥 230026)

[3)](中国科学院合肥物质科学研究院应用技术研究所 合肥 230088)

摘 要 系统虚拟化作为支撑云计算的关键技术,其安全问题显得尤为重要.在虚拟机外部对虚拟机内部的进程进行监视,可以使监视工具与不受信任的虚拟机隔离,能够有效地监视虚拟机行为.当前的虚拟机外部监视技术大多是基于周期或某一时间点上的内存快照技术展开的.但是,由于进程执行具有随机性、突发性的特点,而基于周期性或时间点的内存快照类方法监视过程不连续,容易造成漏检率较高的问题.为此,提出了一种基于自适应机制的虚拟机进程实时监视技术 RTMonitor.RTMonitor 利用虚拟机管理器(Virtual Machine Manager,VMM)在虚拟机外部实时捕获内核栈基址的切换,以此确定进程的切换动作.之后,获取与当前进程相关的硬件信息流,通过解析硬件信息得到当前运行进程的高层语义.针对进程执行的随机性和突发性特点,RTMonitor 通过缓存进程内容并创建多任务的方法提高执行速率,采用自适应调度机制实现存储节点数量、执行任务数量与进程执行情形的动态匹配.与基于时间点的内存快照类方法相比,RTMonitor 将进程监视扩展到了整个时间轴上,使得监视过程具有连续性的特点.最后,通过对进程的捕捉率、捕捉延时及隐藏性检测实验,证明了系统的可行性和有效性.

关键词 虚拟机;实时监视;进程检测;rootkit;隐藏性监测;自适应机制

中图法分类号 TP309 **DOI 号** 10.11897/SP.J.1016.2019.00896

A Real-Time Monitoring Method of Virtual Machine Process Based on Self-Adaptive Mechanism

CUI Chao-Yuan[1] LI Yong-Gang[1),2)] WU Yun[3)] SUN Bing-Yu[1)]

[1)](*Institute of Intelligent Machines, Chinese Academy of Sciences, Hefei* 230031)

[2)](*University of Science and Technology of China, Hefei* 230026)

[3)](*Institute of Applied Technology, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei* 230088)

Abstract Virtualization is a key technology to support cloud computing, and its security is particularly important. Monitoring the processes outside of the virtual machine can isolate the monitoring tools from untrusted virtual machines making the monitoring to virtual machine more effectively. The current virtual machine external monitoring technology is mostly based on the memory snapshot technology running in cycles or at a point in time. However, due to the characteristics of random and sudden of a process, if the method based on periodic or point-of-time memory snapshot to monitor processes is not continuous, it will lead to a higher leakage rate. For example, if a malware has not been started or has been completed during the scan time or scan interval, the security tool will not be able to detect it activity. Therefore, a real-time technology to monitor virtual machine process based on adaptive mechanism, RTMonitor, is proposed.

收稿日期:2017-02-07;在线出版日期:2017-12-11.本课题得国家自然科学基金(31371340)和国家重点研发计划项目(2016YFB0502600)资助.**崔超远**,男,1972 年生,博士,研究员,中国计算机学会(CCF)会员,主要研究领域为系统虚拟化、信息通信安全、云计算.E-mail:cycui@iim.ac.cn.**李勇钢**,男,1988 年生,博士研究生,主要研究领域为操作系统、信息安全和虚拟化.**乌 云**(通信作者),女,1974 年生,博士,副研究员,主要研究领域为人工智能.E-mail:wuyun@rntek.cas.cn.**孙丙宇**,男,1974 年生,博士,研究员,主要研究领域为模式识别.

RTMonitor captures the kernel stack switch in real-time outside the virtual machine by the help of virtual machine manager (called VMM). The captured result is treated as a determination of the process switching action. Then, RTMonitor obtains the hardware information flow related to the current process, and gets the system-level execution semantics of the process by analyzing the hardware information. To solve the problem of the randomness and burstiness of process execution, RTMonitor monitors the process continuously. Besides, it improves its execution speed by caching process content and creating multi-task. RTMonitor adopts adaptive scheduling algorithm that automatically adjusts the number of every task and storage nodes to dynamically match with process execution in all senses. To compensate for the speed difference among all tasks RTMonitor establishes a caching mechanism caching process context and memory, and the cache size shrinks and expands automatically according to process execution scene. When the process switching frequency increases, it first dynamically increases the number of storage nodes used to cache the process data that can not be processed immediately. Then RTMonitor increases the number of tasks to improve data processing speed. When the process switching frequency decreases, RTMonitor reduces the number of storage nodes and the number of tasks to reduce performance overhead. Besides, instead of the logical relationship, such as the doubly linked list structure, between processes, RTMonitor monitors process action though the physical resource updates. As a result, it can detect the rootkits that hides kernel objects by destroying the logical relationship.

The feasibility and effectiveness of RTMonitor are proved by experiments of capture rate, capture delay and concealment. RTMonitor can monitor processes running in milliseconds. The capture rate is more than 95% when processes run less than 10 ms, and it close to 100% when processes run more than 10 ms. At the same time, RTMonitor controls the capture delay 2—15 ms. It introduces 21% and 17.6% performance overhead for CPU intensive application in security VM (called SVM) and target VM (called TVM) respectively. In addition, RTMonitor can detect the processes hidden by all kinds of rootkits. Compared with traditional security tools, RTMonitor will not be bypassed by rootkits, ensuring the completeness of detection.

Keywords　virtual machine; real-time monitoring; process detection; rootkit; hidden detection; self-adaptive mechanism

《计算机学报》

1　引　言

近年来，虚拟化技术展现出了巨大的市场应用前景，根据 IDC(International Data Corporation)最新的统计结果显示，超过 50%的互联网数据都会经过虚拟化平台. 受商业利益驱使，计算机病毒开发者开发出了诸多针对虚拟机的恶意软件. 为了绕过杀毒软件的检测，恶意软件通常与 rootkit[1-2] 相结合，使其本身很难被检测到.

传统的安全工具大都是基于主机的，即安全工具自身部署在目标操作系统中，导致两者之间缺乏足够的隔离性. 一旦目标操作系统被感染，安全工具就会处在一个不受信任的环境中，存在被绕过或者被欺骗的可能. 为增强安全工具与目标操作系统之间的隔离性，最直接的解决方案是将安全工具部署在目标操作系统外部. 这样，安全工具与目标操作系统运行在不同的环境中，使得它们各自所处的空间之间存在天然的隔离. 但是，严格的隔离性也带来了一项严峻的挑战——语义鸿沟问题[3].

语义鸿沟是指不同的语义层级之间所存在的差异，如二进制的内存底层信息和高级语言之间的语义差异. 目前解决语义鸿沟问题最常用的技术是虚拟机自省技术(Virtual Machine Introspection, VMI)[4-5]. 虚拟机自省可以解决语义鸿沟问题，是因为它能够在虚拟机外部完成语义重构. 语义重构的三要素分别是底层状态数据、高层语义信息和语义知识. 其中，语义知识是桥接底层状态数据和高层语

义信息的桥梁[6]. VMI 技术基于这一点,利用语义知识解释底层状态数据,完成由"低级语义"到"高级语义"的转换.

基于 VMI 的入侵检测系统虽然可以在虚拟机外部获取到虚拟机内部的语义视图,但是获取过程要么是基于时间点的,要么是基于周期扫描的. 基于时间点的检测方法类似于单次内存快照,将虚拟机内部状态定格在某一时刻进行分析[7];基于周期扫描的检测方法,每隔一段时间获取一次内存,两次扫描之间存在较长的间隔. 由于恶意软件的启动时间点是随机的,如果恶意软件在扫描时间点或扫描过程中尚未启动或已经完成执行,那么安全工具将无法检测到恶意行为. 例如,文献[8]中提出了一种利用安全工具的扫描间隙向外发送关键文件的技术,可以规避周期性检测. 综上,需建立一套虚拟机内部状态实时监视系统.

进程的执行具有随机性和突发性的特点. 随机性表现为进程的执行时间点具有不确定性,突发性是指短时间内可能爆发大量执行进程. 这些特点对实时监视系统的时效性提出了极高的要求. 进程实时监视系统对进程的捕捉主要包括:监视进程切换、捕获进程内容和解析进程语义三个步骤. 由于短生命周期进程的运行时间极短,甚至可能小于一个最小 CPU 时间片,因此会在被调度执行后很快消亡,它所占用的物理资源也会很快被回收. 若某一随机时间点上大量短生命周期进程同时创建执行,进程的消亡与调度执行频繁发生,导致硬件中(如寄存器、内核栈及内存等)的二进制字节信息刷新速度激增,即进程内核栈与内存内容刷新速度或置换频率加快. 如果单次扫描时间过长,则可能对进程切换产生漏检,导致进程内容捕获失败,降低进程捕捉率;如果语义解析时间过长,则会增大捕捉延时,降低实时性能.

针对上述问题,该系统需要提高硬件信息流的捕捉速度和语义解析速度. 本文提出一种针对虚拟机进程的实时监视方法 RTMonitor. RTMonitor 采用动作分离策略,将进程切换捕捉、内存捕捉、语义解析分离成实现上相互独立、逻辑上相互关联的执行实体. 进程内存捕捉及语义解析实体通过多任务并发执行方式实现,二者根据进程执行情形的变化自动扩张和收缩. 为补偿各执行实体间的速度差异,减少漏检现象,RTMonitor 建立了能够自动收缩和扩张的缓存机制,缓存进程上下文与内存. 进程切换过程中会伴随着进程上下文的切换,RTMonitor 首先通过快速连续地监视虚拟机进程的上下文切换实现对虚拟机进程切换的实时捕捉;然后通过内存映射捕获进程的内存原始数据;最后根据语义知识库将原始内存数据还原成高层语义. 与传统的内存快照技术相比,RTMonitor 的监视过程具有连续性特点,解决了基于时间点和基于周期的方法所产生的检测时间点选取困难和检测间隔过长的问题.

2 相关工作

基于 VMI 的入侵检测系统因其良好的抗干扰性而得到广泛关注. 近年来,许多研究者对该类系统展开了深入的研究工作. 按照在获取虚拟机内部状态时是否需要介入虚拟机内部,可以将虚拟机监视系统分为两类. 一类是需要介入虚拟机内部的系统. 例如,Sebastian 等人[9]提出的 X-TIER,首先向目标虚拟机(TVM)中注入内核模块,然后通过该模块读取 TVM 的数据结构以获取虚拟机的状态信息,最后以超级调用(Hypercall)的形式将获取的信息实时传递给虚拟机管理器(VMM). 与 X-TIER 类似,SYRINGE[10]也需要向 TVM 内部注入内容以获取内部状态信息. 它采用 Function-call Injection 技术,使其能够在虚拟机外部对虚拟机内部的函数进行调用. 同时采用 Localized Shepherding 技术监控数据获取代码的控制流的完整性. SYRINGE 将上下文(injection context)信息注入到 TVM 的代理进程中,并在注入位置处设置页表级断点. 当进程执行到断点处发生中断后会捕获高层语义信息并通过 Function-call Injection 将捕获数据传递给 TVM 外部的监控程序. Virtuoso[11]和 VMST[12]从控制逻辑角度着手获取 TVM 状态信息. Virtuoso 首先需要进行训练,将虚拟机内部程序在 TVM 内运行多次,提取相关指令及指令执行路径;之后生成自省代码所需的路径,最后将其融合翻译为可在虚拟机外执行语义重构的代码. VMST 针对 virtuoso 训练工作繁琐且自动化程度低的问题,将代码执行与数据访问进行分离后利用内核重定向技术,将处于 TVM 内部的代码与虚拟机内存中的数据相结合生成自省程序.

另一类是系统无需介入虚拟机内部. VMwatcher[13]首先获取到虚拟机的内存,然后把 TVM 内核数据(如文件、目录、进程)结构定义和函数语义(如文件系统驱动程序)用作模板来解释从 VMM 获取的底层虚拟机状态,进行语义视图重构,得到虚拟机状态信息. 与 VMwatcher 不同,RTKDSM[14]

是一种实时系统.它将整个系统分为两部分:自省代理和监视代理.前者置于安全虚拟机中,后者置于 hypervisor[15]中.监视代理通过第三方工具 volatility① 定位特定数据结构在内存中的位置;自省代理利用 xenaccess② 进行内存映射,并利用内核知识完成语义的重构.通过对特定数据结构所在的页进行只读设置,使得发生在该数据结构中的任何写操作都会产生页错误,从而被 hypervisor 捕捉到,完成该数据结构的实时变化监视. TxIntro[16]也是一种实时系统,它采用 Intel 提出的 RTM 技术,可以实时跟踪特定数结构的变化.此外,TxIntro 利用 RTM 技术的强原子性操作原理解决了 VMI 过程中存在的状态一致性问题.

上述几项研究都使用了 VMI 技术在虚拟机外部对 TVM 内部状态进行监视.但是,除 RTKDSM 和 TxIntro 之外的其它几种方法,都是基于时间点展开的,无法对虚拟机进行持续性监视.如果恶意软件采用 Hyperprobe[17]中提到的反向侦测技术,将直接导致基于时间点的检测方法失效.而 RTKDSM 在进行完全的实时监控时,对操作系统产生的性能开销会随着所监视数据结构数量的增多而急剧增加,因此往往需要设置较长的扫描周期.此外,RTKDSM 的监视对象具有局限性,仅针对其 PFN 列表内的内存监视有效,无法对随机生成的进程及其内存进行有效监视.与 RTKDSM 类似,TxIntro 也对监视对象进行了设置,它只对事先设置好的"读集"内的数据结构具有监视效果,对随机生成进程的数据结构无效.

3　系统设计

3.1　安全模型

本文中虚拟化平台上运行着 SVM(安全虚拟机)和 TVM 两类虚拟机.RTMonitor 的设计实现基于以下两点假设展开:

(1)虚拟化平台,即 hypervisor 是安全的;

(2)hypervisor 为虚拟机之间提供严格的隔离,被感染的 TVM 并不会影响 SVM.

3.2　总体设计

3.2.1　设计目标

RTMonitor 旨在利用虚拟化技术在虚拟机外部实现对 TVM 的实时监视,在确保监视过程连续性和实时性的同时,不对 SVM 和 TVM 引入过多的性能开销 .因此设计目标包括以下四个方面:

(1)实现对进程捕捉的高捕捉率. 基于此点要求,RTMonitor 需要及时地发现进程的切换动作,然后快速捕捉到底层内存,因此须快速地对进程的物理资源进行连续地监视和获取;

(2)尽量减小进程捕捉延时. 为减小捕捉延时,除快速捕捉进程内存原始数据外,还需要提高语义解析速度;

(3)引入较小的性能开销. 在确保进程高捕捉率、低延时的前提下,要尽量减小 RTMonitor 对系统造成的计算开销和存储开销;

(4)发现隐藏进程. 对 TVM 内部的隐藏进程,RTMonitor 要能够及时地发现,以确保进程捕捉的完备性.

3.2.2　RTMonitor 总体架构

RTMonitor 的总体设计方案如图 1 所示,主要由进程切换监视模块 WatchStack、调度模块 sched、内存映射模块 MapMem、语义解析模块 SemParse、环形栈基址缓冲池(包括缓冲池调整模块)和双向循环链表(包括链表调整模块)六部分组成.运行过程中,WatchStack 监视内核栈切换动作,并将新切换的内核栈存入环形缓冲池;与此同时,sched 从缓冲池中读出内容,传递给空等态的 MapMem. MapMem 将逐次进行内存的定位、映射、读取及存储四步操作;最后 SemParse 根据 MapMem 存入双向循环链表的原始内存进行语义解析.

实现 RTMonitor 需要解决三个关键问题:一是实时地捕获进程的切换,第二个是及时地提取进程的底层内存数据,第三个是快速地恢复进程语义.前两个是影响进程捕捉率的主要因素,第三个主要影响进程捕捉延时.

在 RTMonitor 的运行过程中,首先需要 WatchStack 连续快速地捕捉到进程的切换动作. RTMonitor 采用"最小即最优"[18]的原则设计了 WatchStack,以提高其运行速度,这解决了第一个关键问题.然后,MapMem 需要及时地提取出进程的原始内存数据,以防内存置换或刷新导致进程漏检.由于 WatchStack 的执行速度较快,所以 RTMonitor 建立缓冲池暂存 WatchStack 捕捉的栈基址.同时 MapMem 采用多任务执行策略,并发地从缓冲池中读取栈基址,完成内存映射,这解决了第二个关键问题.最后,SemParse 执行语义解析,将原始

① http://www.volatilityfoundation.org/.
② https://sourceforge.net/projects/xenaccess/.

计算机学报

内存数据恢复成高层语义. 与 MapMem 类似,SemParse 也采用多任务并发执行策略,以加快执行速度,这解决了第三个关键问题.

此外,OS 中的进程执行具有随机性和突发性特点,给实时系统带来挑战. 进程可能会在极短的时间内爆发性执行,造成进程数量激增,进程切换频率增大;也有可能会在相当长的一段时间内,进程执行数量极少,进程切换频率极低. 如果任务数量和缓存节点数量保持不变,那么为保证进程捕捉率,需要将二者的数量设置为较大的值,以防止进程激增时导致严重的漏检. 这使大量任务和缓存节点多数时候处于空闲状态,导致计算资源、存储资源的浪费,并增加了实时系统的负载,也会影响整个虚拟化平台的性能. 因此,RTMonitor 采用自适应机制控制任务数量和缓存节点数量随进程执行情形的变化而动态变化.

当缓冲池或双向循环链表被填充满时,触发节点扩张动作;当空闲节点过多时,则会触发节点收缩动作,扩张和收缩的方向相反. 任务(MapMem 和 SemParse)数量的扩张和收缩受自身数量与待解析的缓存节点数量两方面因素影响. 当待解析的缓存节点数量较多而任务自身数量较少时,则会触发任务数量扩张;反之,如果缓存节点数量较少而任务自身数量较多时,则会触发任务数量收缩.

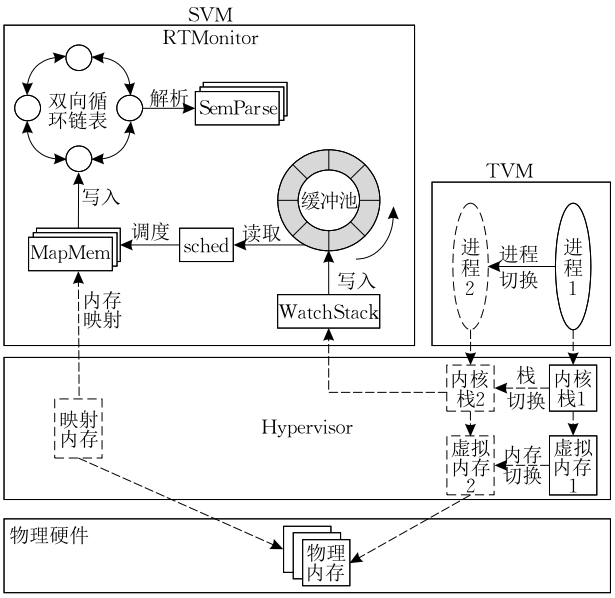


图 1 RTMonitor 总体架构

4 系统实现

4.1 进程资源

任何进程的运行都需要依赖操作系统的资源,RTMonitor 通过对 Linux 进程的物理资源切换监视实现进程的实时监视. 在 Linux 中,内核会调用 alloc_thread_info 为每个进入就绪态的进程分配若干个完整的内存页(8 KB 或 16 KB)作为内核态栈. 内核态栈采用自顶向下的方式增长,esp 寄存器指向栈底,数据结构 thread_info 存储在以栈基址为起始地址的一段内存中. thread_info 中的第一个成员变量指向进程描述符 task_struct. 与进程相关的状态信息都存储在该数据结构中.

当进程处于运行态时,它会占用 CPU 时间片,在单个时间片内该进程独占 CPU 资源,如寄存器、缓存等. 直到进程结束或时间片耗尽,将资源释放. 根据进程对系统资源的占用时间长短,本文将进程进行如下分类:

分类 1. 运行时间 $t<6$ ms 的进程记作“瞬态进程”;

分类 2. 运行时间 $6\text{ ms}\leq t\leq 100$ ms 的进程记作“常态进程”;

分类 3. 运行时间 $t>100$ ms 的进程记作“常驻进程”.

4.2 捕获当前运行进程

4.2.1 WatchStack 工作流程

WatchStack 作为 RTMonitor 动作的第一个环节,它的执行速度是影响进程捕捉率的重要因素. 如果 WatchStack 对 TVM 的进程切换监视产生遗漏,将直接导致进程漏检. WatchStack 的运行过程是连续的,即完成一次监视之后并不停歇,而是继续进行下一次监视. 进程运行时间与 WatchStack 的扫描时间系如图 2 所示. 图中黑色线段代表进程运行的最短时间 T,虚线段表示 WatchStack 的最大运行时间 t;①、②、③代表进程 3 种不同的运行起点,a、b、c 代表 WatchStack 不同的运行时长.

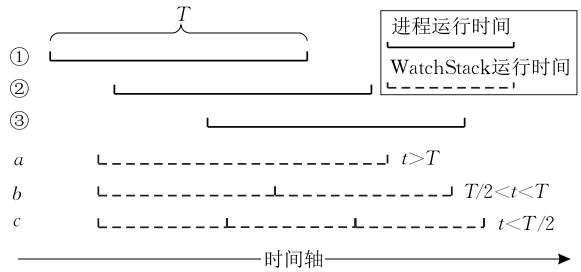


图 2 WatchStack 与进程运行时长关系图

为保证能够捕获到完整的内核栈基址,WatchStack 需要在任意进程运行时间内至少完成一次完整的执行. 从图中可以看出只有 c 能满足该条件,因此需要保证 WatchStack 的单次最长运行时间 t 要

小于进程的最大运行时间的 1/2. RTMonitor 采用“最小即最优”的原则设计 WatchStack，以加快它的运行速度. 遵循“最小即最优”原则，WatchStack 只负责捕获进程切换产生的新栈基址，将后续操作交由其它模块.

在虚拟化平台中，对硬件资源的访问需要 hypervisor 的介入，以超级调用 hypercall[19] 的形式对硬件信息进行读取操作. RTMonitor 利用 36 号超级调用__HYPERVISOR_domctl[20] 捕获 TVM 的虚拟 CPU(VCPU)的上下文信息，并从中提取出 esp 寄存器信息. 然后屏蔽 esp 寄存器后 12 位获取栈基址，即 thread_info 的首地址；最后利用 thread_info→task 获取到占用当前 CPU 时间片的进程. WatchStack 工作流程如图 3 所示，其工作过程如下所述：

Step 1. 调用 HYPERVISOR_domctl 获取 VCPU 上下文信息；

Step 2. 获取栈基址，并与预存值进行异或操作，若返回值为 0，则直接抛弃，进入下一轮监视动作，若非 0，则将该栈基址存入环形内存缓冲池中.

Step 3. 设置内存缓冲池中的数据节点状态为 1，更新预存值为当前栈基址，进行下一轮监视. 内数据节点状态 0 表示当前节点为空，状态 1 表示当前节点已完成数据存储且非空.

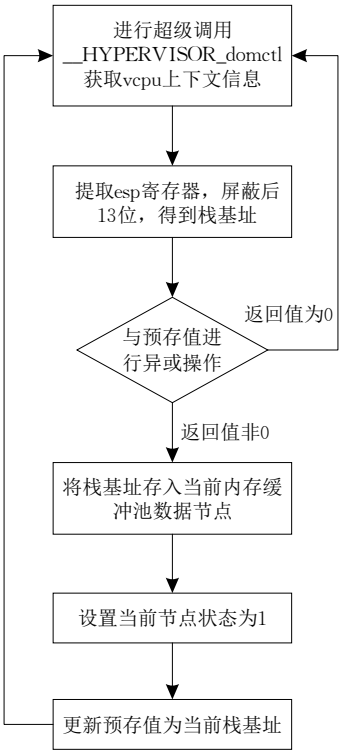


图 3　WatchStack 工作流程图

4.2.2　环形栈基址缓冲池

环形栈基址缓冲池用于缓存 WatchStack 捕获的内核栈基址，建立数据结构 stack_struct 描述缓冲池数据节点的状态，如定义 1 所示. 第 1～2 行定义了数据节点的两种状态(与第 4 行相对应)，0 代表节点为空，1 代表节点填充完毕. 第 5 行用于记录栈基址，第 6 行指向下一节点的状态描述符.

定义 1.　缓冲池节点状态描述符 struct *stack_struct*.

```
1. #define FULL      1
2. #define EMPTY     0
3. struct stack_struct {
4. int state;
5. addr_type base_addr;
6. struct stack_struct *next;};
```

环形栈基址缓冲池节点数量初始值设置为 α，之后由缓冲池调度模块动态地调整. 调度模块将实时统计缓冲池节点总量. 当 WatchStack 与 sched 的工作节点重合且下一节点非空时，对缓冲池节点数量进行扩张，单次扩张节点数量设置为 β. 当 sched 与 WatchStack 所处节点重合且下一节点为空时，调度模块对缓冲池节点数量进行收缩. 节点收缩方向与节点填充方向相反，当节点收缩数量为当前总结点数量的一半或遇到非空结点时停止收缩.

4.3　获取底层内存

4.3.1　MapMem 自适应扩张

MapMem 受 sched 调度，采用多任务轮策略提取 TVM 底层内存. sched 从内存缓冲池节点中读取栈基址并将当前节点状态设置为 0，然后对 MapMem 进行轮询调度，其工作流程如图 4 所示.

图 4 中 j 代表当前缓冲池的数据节点；k 代表缓冲池节点总数；i 代表当前轮询到的内存映射模块；n 代表内存映射模块总数；count 代表 sched 已经轮询的模块个数. sched 根据 WatchStack 的捕捉速率自适应地增加 MapMem 的数量，从而快速地完成内存映射，使内存页的刷新与换出导致的漏检现象减少.

4.3.2　MapMem 自适应收缩

进程的创建执行具有突发性特点，短时间内可能会产生大量的进程，从而生成许多内核栈及 task_struct. 为及时获取新生成的内存内容，MapMem 受 sched 调度，使任务数量自适应增加. 但若是某一时间段内以常态或常驻进程为主的话，则会存在较多空等态的 MapMem，造成存储和计算资源的浪费，

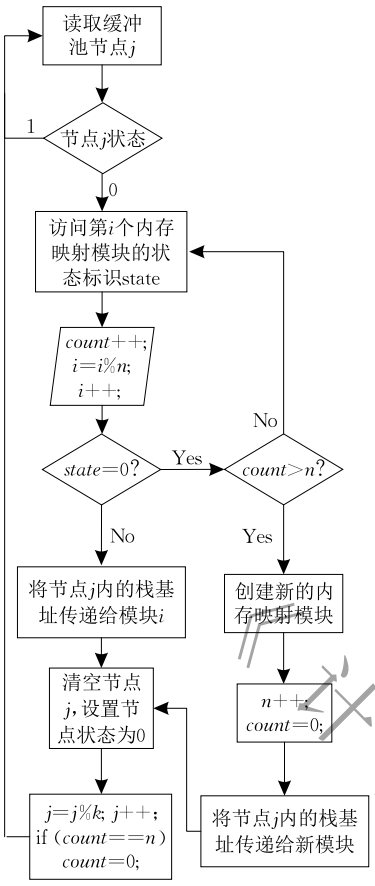


图 4 sched 调度流程图

影响系统实时性. 因此,MapMem 的数量需要自适应收缩以降低性能开销. 为描述 MapMem 的运行状态,建立状态描述符 struct map_struct,如定义 2 所示. 第 1～2 行的 TASK_RUNNING 和 TASK_WAITING 用于描述 MapMem 的运行状态(第 6 行),分别表示运行态和空等态;第 4 行表示将所有 MapMem 组成双向循环链表;第 5 行是 MapMem 的任务 id 号,用于唯一的标识 MapMem;第 7 行 run_time 记录任务最近一次执行内存映射所消耗的时间.

定义 2. MapMem 状态描述符 struct *map_struct*.

1. **#define** *TASK_RUNNING* 1;
2. **#define** *TASK_WAITING* 0;
3. struct *map_struct* {
4. struct list_head *tasks*;
5. int *map_id*;
6. int *state*;
7. long *run_time*;};

当 MapMem 处于空等状态时,它将启动定时器,定时参数为 map_struct→run_time,之后将自己挂起. 挂起期间若收到 sched 的调度信号,则被唤醒,终止定时,改为运行态并进行内存映射操作. 在本次映射完成之后更新 run_time,再次进入空等状态. 若定时结束后未收到 sched 的调度信号,则在定时结束后查看其所在链表中的上一个任务状态 state,若为 1,则进入空等状态,若为 0,则将自身从双向链表中删除,释放资源退出.

4.3.3 MapMem 捕获内存

由于虚拟机之间存在严格的内存隔离,所以所有在虚拟机外部对虚拟机内部的内存读操作都需要进行内存可读映射. MapMem 用于将当前进程的 task_struct 进行可读映射,并将其保存在双向循环链表中,供 SemParse 进行解析. MapMem 以页(4 KB 大小)为单位进行内存的可读映射,所以首先需要得到内存页框首地址(伪物理地址). 借助于从 VCPU 上下文中提取的 cr3 寄存器,通过分页机制下的多级页表转换即可获得该地址. 紧接着,以该地址为参数,通过 ioctl 函数向 SVM 中的设备驱动 privcmd 传递 IOCTL_PRIVCMD_MMAP 命令,将该地址在 TVM 内部所属的整个物理内存页映射到 RTMonitor 的地址空间内.

不同的内核版本中,task_struct 所占的内存大小存在差异,但都处在(4k,6k)的区间内. 在 Linux 中,task_struct 的内存空间由 slab 分配,以 2^k(k≤12)个字节对齐[21],而非页对齐方式. 所以,task_struct 所需跨越的内存页数量与它在页中的起始位置和大小有关. MapMem 在进行映射之前要根据 task_struct 的首地址及从语义知识库中提取的 task_struct 的大小判定所需映射的次数,判定方法如算法 2 所示. 第 1 行定义了内存页大小,本文中采用的是传统的 4 KB 大小的分页方式. 第 2 行计算 task_struct 在内存页中相对页首的偏移量. 第 3 行计算 task_struct 的大小与页大小的差值. 第 4～6 行用于判定 task_struct 在内存中跨越的内存页数量,即需要映射的次数. 第 7 行返回计算结果.

算法 2. 映射页数计算算法.

输入:虚拟地址 *vaddr*,*task_struct* 字节数 η

输出:需要映射的次数 *page_num*

1. **#define** *PAGE_SIZE* $0x1\ll12$
2. offset=vaddr & $0xfff$;
3. value=η−PAGE_SIZE;
4. **IF** ((*PAGE_SIZE*-*offset*)<*value*)
5. *page_num*=3;
6. **ELSE** *page_num*=2;
7. **RETURN** *page_num*;

计算机学报

MapMem 会将映射之后的数据存储在双向循环链表的数据节点中，供语义解析模块解析. 完成映射的 MapMem 设置自身状态为空等态，等待再次被调度.

4.4　语义解析

语义解析是将存储在双向循环链表中的 task_struct 的字节信息转换成高层语义. RTMonitor 首先离线地对内核版本进行解析，提取出内核语义知识，构建完备的语义知识库. 语义知识库中涵盖了当前流行的内核版本的语义知识，如 task_struct、thread_info 和 mm_struct 等数据结构语义. RTMonitor 所需的数据结构信息包括数据结构中成员变量相对于首地址的偏移量及成员变量的数据类型. 前者用于数据定位，数据结构首地址加偏移量即为成员变量的地址；后者用于语义恢复，特定数量的内存根据数据类型即可恢复为高层语义. 此外，知识库中还封装了对库的索引函数和用于语义提取的内核操作函数.

与 MapMem 类似，语义解析模块 SemParse 和双向循环链表也采用自适应机制进行自适应扩张和收缩，其状态描述符如定义 3 所示. SemParse 存在运行、空等和消亡三种状态(1～3 行). 第 5 行用于描述任务的运行状态，第 6 行记录任务的 id 号. 若 SemParse 的当前目标节点非空则其处于运行态；若其目标节点状态为空时则处于等待状态；若其生成了子任务则进入消亡态. SemParse 的状态转移图如图 5 所示.

定义 3.　SemParse 状态描述符 struct *sem_struct*.

1. **#define** *RUNNING*　0
2. **#define** *WAITING*　1
3. **#define** *DYING*　2
4. struct *sem_struct* {
5. int *state*;
6. int *sem_id*;};

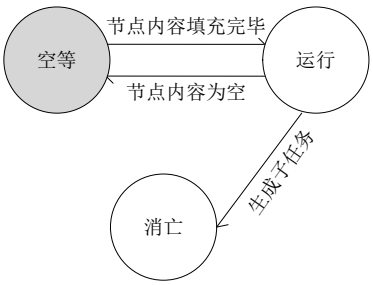


图 5　SemParse 状态转移图

双向循环链表的节点状态如定义 4 所示. 双向循环链表的数据节点含有“空”和“满”两种状态(1～2 行)，用于描述当前节点内容是否已被读取，第 4 行记录了节点状态. 第 5 行的指针指向了 6KB 大小的存储空间，这段内存用于存储 task_struct 的原始数据；第 6 行是数据节点的双向循环链表组织形式.

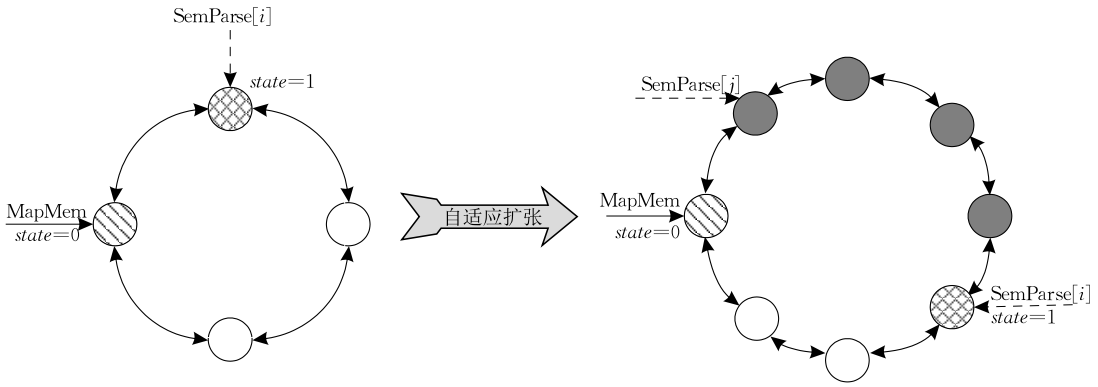
定义 4.　双向循环链表状态描述符 struct *node_struct*.

1. **#define** *EMPTY*　0
2. **#define** *FULL*　1
3. struct *node_struct* {
4. int *state*;
5. void **task_mem*;
6. struct list_head *list*;};

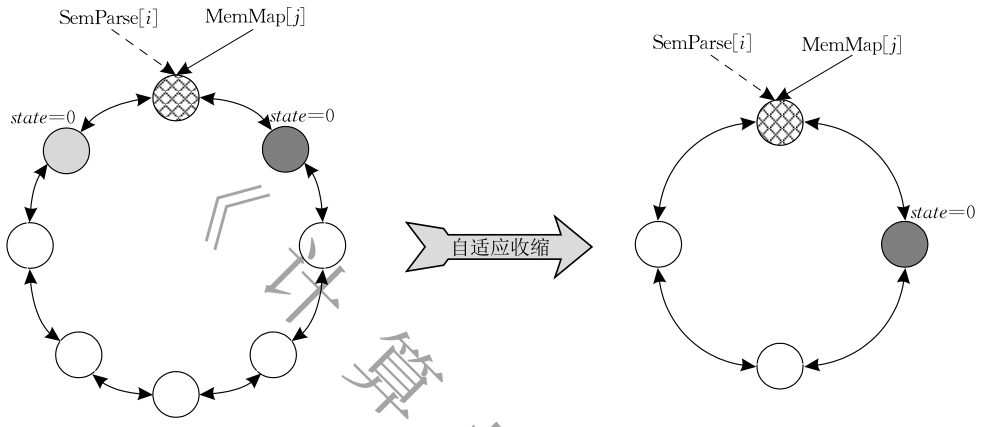
MapMem 与 SemParse 均沿双向循环链表正向滑动，MapMem 将内存映射数据填充到数据节点中，SemParse 从节点中提取内容进行语义解析. 双向链表的节点数量会随着进程执行情形的变化自适应地收缩与扩张. 同时，SemParse 也会跟随节点数量的变化而自适应地扩张和收缩. 双向循环链表和 SemParse 的自适应变化过程如图 6(a)、6(b)所示.

自适应扩张. 图 6(a)中，当 MapMem 执行到某一节点处时，首先检查当前节点和下一节点的状态，若当前节点状态为 0(节点内容已被读取)且下一节点状态为 1(节点内容尚未被读取)，则触发节点扩张动作，节点数量沿正向扩张，图中灰色节点即为扩张的节点. 扩张之后，MapMem 继续沿双向链表正向填充数据. 同时，SemParse[i]创建新的任务 SemParse[j]并将自身切换到消亡态，新任务创建后处于空等态，直到新生成的数据节点被填充完数据进入运行态. SemParse[i]将继续沿双向链表正向进行语义解析，直至解析至状态为 0(已被读取)的数据节点时释放资源并退出. SemParse[j]则会沿着新节点继续工作，若其生成了新任务 SemParse[$j+1$]，它也会将自身切换至消亡态，在解析至状态为 0(已被读取)的数据节点时消亡；若其一直未生成新的任务，则它将一直运行下去.

自适应收缩.　图 6(b)中，若 SemParse 处于运行态，当其前后两个数据节点状态均为 0 时，链表进行收缩，收缩方向与扩张方向相反，沿双向链表的逆向收缩. 数据节点收缩数量为当前数据节点数量的一半或遇到节点状态为 1 的数据节点时停止收缩.



(a) 双向循环链表与SemParse自适应扩张



(b) 双向循环链表与SemParse自适应收缩

图 6 双向循环链表和 SemParse 的扩张与收缩

5 实验及分析

5.1 实验环境

实验环境由物理主机、Xen 虚拟化平台、Linux 安全虚拟机(SVM)系统和 TVM 系统组成. 物理主机为 Dell R710 机架式服务器,配置为 Intel Xeon (R) E5520@2.27 GHz 16 核处理器,16 GB 内存,137 GB 硬盘. Xen 虚拟化平台为 Xen 4.1.2. 操作系统系统环境如表 1 所示.

表 1 系统环境

	CPU	内存	OS	内核
SVM	16 核,2.27 GHz	16 GB	64 位,Ubuntu12.04	3.2.16
TVM1	单核,2.27 GHz	1 GB	64 位,Ubuntu12.04	3.2.0-23-generic
TVM2	单核,2.27 GHz	1 GB	32 位,CentOS6.4	2.6.32
TVM3	单核,2.27 GHz	1 GB	64 位,Ubuntu16.04	4.4.0-53-generic

针对进程的随机性和突发性特点,本文模拟了两种类型的实验过程:3 个小时之内随机挑选 1000 个时间点,在每个时间点上生成 1 个进程,记作 A 情形;随机选取 1 个时间点,以该时间点为起点,快速生成 1000 个进程,记作 B 情形. A 情形模拟 TVM 中进程执行时间点选取的随机性特点. B 情形模拟进程执行突发性的特点,即短时间内爆发大量进程(类似于压力测试). 针对 A、B 两种情形,RTMonitor 对进程进行捕捉,并记录进程捕获率和捕捉延时. A 情形用以验证 RTMonitor 监视过程的连续性,B 情形用以验证它的有效性.

实验过程中,首先测量 RTMonitor 各组件的运行时长范围. 在 TVM1、TVM2 和 TVM3 三台虚拟机中逐次设定 A、B 种执行情形,在 TVM 中使指令"ps -aux"执行 1000 次. 实验中分别测量 WatchStack、sched、MapMem 和 SemParse 的运行时长,并记录范围值,如表 2 所示.

表 2 RTMonitor 组件运行时长 /μs

	TVM1		TVM2		TVM3	
	A	B	A	B	A	B
WatchStack	112～268	129～307	121～276	123～296	131～274	122～311
sched	36～409	43～391	42～466	51～379	49～343	55～486
MapMem	774～1060	896～1123	764～998	804～1203	966～1312	1004～1297
SemParse	1485～1559	1399～1602	1517～1624	1471～1618	1369～1581	1542～1616

5.2　进程的实时监视

5.2.1　CPU 密集型进程

CPU 密集型进程是指大部份时间用来做计算、逻辑判断等 CPU 动作的进程，此时 CPU 负载较大，本文以圆周率 Pi 的计算作为 CPU 密集型进程进行试验. 实验统计后发现，Pi 计算圆周率后 0 位(仅完成一次逻辑判断操作)的时间 t 范围是 $1\,\text{ms}<t<2\,\text{ms}$，计算后 400～500 位的时间 $6\,\text{ms}<t<9\,\text{ms}$，计算后 1000 位的时间范围为 $100\,\text{ms}<t<500\,\text{ms}$. 通过对 Pi 计算位数的调整，可使 Pi 分别成为瞬态、常态和常驻型进程.

针对 A、B 两类情形，RTMonitor 分别对进程进行捕捉，捕捉率如图 7 所示. 由图可知对于瞬态进程，即计算 0 位 Pi(运行时间约为 1～2 ms)，无论是随机性还是突发性实验，其捕捉率都超过了 96%，分别达到了 97.1%和 96.2%. 对于常态进程，即计算 400～500 位以上，捕捉率接近 100%. 对于常驻进程捕捉率稳定在 100%. 对常态进程和常驻进程的捕捉效果体现了 RTMonitor 对进程监视的连续性. 从两类进程的执行角度来看，RTMonitor 的监视过程不存在间歇，即在任何一个常态或常驻进程的执行期间 RTMonitor 都能够依次完成进程的切换监视、内存捕捉和语义解析，使其无法逃避 RTMonitor 的监视.

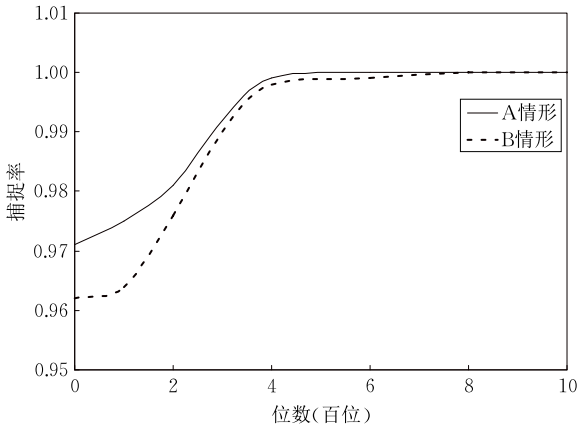


图 7　CPU 密集型进程捕捉率

5.2.2　I/O 密集型进程

I/O 密集型进程是指大部分时间 CPU 都在等待 I/O(硬盘/内存)的读/写的进程. 实验中采用对硬盘文件进行读操作的进程 ReadFile 作为 I/O 密集型进程，在 ReadFile 运行期间 CPU 的大部分时间用于等待 I/O 结束. 通过设置文件大小控制 ReadFile 的执行时间，可使其分别成为瞬态、常态和常驻型进程. 文件大小是 1 KB 时，ReadFile 运行时间 $3\,\text{ms}<t<6\,\text{ms}$；文件大小是 30 KB 时，ReadFile 运行时间 $6\,\text{ms}<t<10\,\text{ms}$；文件大小是 200 KB 时，ReadFile 运行时间 $50\,\text{ms}<t<60\,\text{ms}$. 针对进程的随机性和突发性，采用和 5.2.1 节相同的两种执行情形，分别对进程进行实时捕捉. 捕捉效果如图 8 所示.

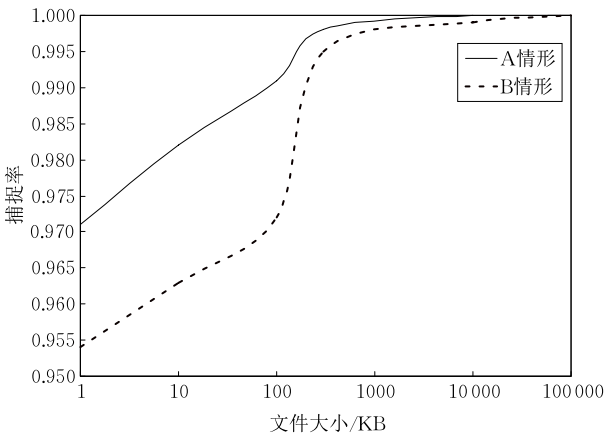


图 8　I/O 密集型进程捕捉率

与 CPU 密集型进程的捕捉效果类似，A 情形的捕捉率略高于 B 情形. 在文件大小是 1 KB 时(只读取一个字节，ReadFile 运行时间约为 4 ms)，两种执行情形的捕捉率分别为 95.4%和 97.1%；在文件大小是 300 KB 时(ReadFile 运行时间约为 10 ms)，两种执行情形的捕捉率都接近 100%.

RTMonitor 对 CPU 密集型的进程的捕捉效果要好于 I/O 密集型进程的捕捉效果. 这是因为在 CPU 密集型的进程执行过程中，CPU 负载较大，占用 CPU 资源的时间较长，因此 RTMonitor 在下一次资源置换或刷新之前有更长的时间捕获到它们，增大了信息流捕捉概率.

造成 RTMonitor 进程漏检现象的缘由来自两方面，一是对内核栈切换的漏检导致的进程完全遗漏；另一个是获取底层内存时，内存已经被刷新或置换至交换区导致的进程完全遗漏，或因内存状态正在发生变化产生的状态一致性问题[22]导致的进程状态部分缺失. RTMonitor 对内核栈切换的单次监视动作耗时约为 120～300 μs 左右. 进程是动态实体，其生命周期范围从几毫秒到几个月[23]，所以在任何进程的执行期间 WatchStack 对内核栈的切换扫描超过两次，符合 4.2.1 节中提到的不漏检条件，因此对内核栈切换的监视并不会出现遗漏. 但是，内核栈监视耗时会导致内存映射操作产生 120～300 μs 的滞后，记为 t_1. 此外，内存映射模块 MapMem 在产生动作之前要接受 sched 的调度，调度时长(记为 t_2)受进程切换速率和进程数量影响. 正常情况下，

sched 直接对已有的 MapMem 执行队列进行任务调度，调度时间约为 50 μs；若短时间内大量进程需要同时执行，进程切换速率随之增大，sched 需要生成新的 MapMem 任务，使调度时长增大为 150～500 μs. MapMem 获取底层内存需要经过 2～3 次内存映射和页拷贝，耗时约为 0.8～1.3 ms，记为 t_3. 由于内存映射过程中存在缓存，因此若某个进程在完成一次调度时还没执行完，紧接着被再次调度，由于缓存中仍然存在映射的页表，所以无需进行新的映射，此时 t_3 接近 0. 记进程内存捕捉时间为 t_0，则 $t_0=t_1+t_2+t_3<2$ ms. 若 t_0 大于进程的执行时间，则会由于对运行时间极短的进程获取内存不够及时而产生进程漏检.

通过对 CPU 密集型和 I/O 密集型进程的捕捉结果表明，RTMonitor 能够实时有效地捕获随机执行的进程，且捕捉率较高. 由于 RTMonitor 在完成单次监视之后，并不停歇，而是立即展开下一轮的监视，因此它的实时与连续程度和自身的单次扫描时间有关，对运行时间大于等于 2 ms 的进程均能捕捉到. 与 X-TIER、SYRINGE、VMwatcher 等单次内存快照技术的监视系统对比，RTMonitor 的监视效果具有持续性和实时性的特点. 与 RTKDSM 和 TxIntro 相比，RTMonitor 的监视对象是 TVM 中实时创建执行的随机进程，而非限定的监视对象.

5.3 捕捉延时

捕捉延时是指进程捕获时间点与进程运行时间点之间的时间差，是实时系统性能的重要指标. 本节采用 5.2 节中的应用程序 Pi(CPU 密集型进程)和 ReadFile(I/O 密集型进程)测试 RTMonitor 的进程捕捉延时，实验结果如图 9 所示. 实验中将 Pi 和 ReadFile 都设置为运行时间约为 5 ms 的瞬态进程，随机选取一个时间点连续执行多次. 图中 X 轴表示应用程序在随机选取的时间点上执行的次数，Y 轴表示在执行 1～1000 次的过程中逐次记录所产生的延时，并根据最新值不断更新最大延时.

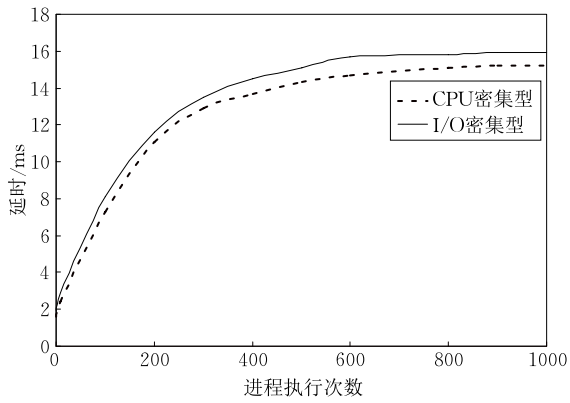


图 9 捕捉延时

结果表明对 CPU 密集型进程的捕捉延时略小于 I/O 密集型进程的捕捉延时. 这是因为相比于 I/O 密集型进程，CPU密集型进程在调度过程中使用CPU资源的时间更久. RTMonitor 通过监视进程资源调度实现对进程的实时捕捉，捕捉的起始点是 CPU 上下文中的寄存器信息，因此频繁使用 CPU 资源的进程，其寄存器的变化更容易被捕获到. 从图 9 中还可以看出，捕捉延时首先随着进程的执行次数逐渐增大，之后逐渐收敛. 曲线单调递增的原因包括两方面，一方面是在极短的时间内 TVM 爆发的进程数量越多，它的负载就越大. 高负载运行的 TVM 会占用较多的物理资源，导致整个虚拟化平台负载增大，影响平台整体执行速度. 另一方面，瞬态进程的爆发式执行会导致 RTMonitor 的负载迅速增大，使并发性任务数量增多，缓存节点数量也随之增多. 多任务并发执行策略虽然能够加快整体运行速度，但是作为多任务中的个体会受资源分配、任务调度等因素影响，导致其执行速度降低. 任务数量越多，所受影响就越大，延时也就越大. 缓存节点数量的增多会使节点的等待读取、解析时间增大，进而影响捕捉延时.

RTMonitor 的延时主要由进程捕捉和语义解析两部分产生. 由 5.2 节分析可知，进程捕捉产生的延时小于 2 ms，当存在映射页缓存时，其值达到最小值，约为 170～350 μs，记为 t_0'. 语义解析带来的延时来自三方面：一是语义解析本身所消耗，包括节点内容读取、索引语义知识库、解析内存及语义输出四部分耗时，其值的大小在一个很小的范围内变化，趋于一个常量，约为 1.6 ms，记作 t_1'；第二个是节点等待时间 t_2，双向循环链表中节点内容的“生产”(填充节点)与“消费”(解析节点内容，即语义解析)两个动作是异步的，尤其是处于第二种执行情形时，“生产”的速度大于“消费”的速度，因此产生了非空等待节点，导致等待延时，该值变化范围较大；第三是节点跳转时间 t_3'，该值相对 t_1' 和 t_2' 可忽略. 当双向循环链表节点内容的“消费”速度大于“生产”速度时，捕捉延时 $T=t_0'+t_1'+t_2'$，此时 $t_2'=0$，出现最小延时 $T<2$ ms. 语义解析的自适应机制可以控制第二种延时的变化范围. 当双向链表自适应扩张时，语义解析模块任务数量随之扩张，使等待被解析的节点数量始终小于等于扩张的节点数量. 从图中可以看出，延时大小逐渐向 14～16 ms 之间收敛，收敛值即为 RTMonitor 的最大延时. 当双向链表自适应扩张时，MapMem

所在的节点等待时间最长，因此该节点将会产生最大延时. 最大节点等待时间 $t'_{2\max}$ 等于该节点之前尚未被解析的最大节点数量 n(n 小于等于单次扩张生成的新节点数量，实验中设置为 8，将在 5.4.2 节中说明)乘以 t'_1，即 $t'_{2\max} \leq n \times t'_1$，即最大延时 $T = t'_0 + t'_1 + t'_{2\max} \leq 2 + (n+1) \times t'_1$.

5.4　栈基址缓冲池与双向循环链表

本节采用 5.2 节中的 Pi，将其设置为运行时间约为 5 ms 的瞬态进程. 在随机的时间点 Pi 连续执行 1000 次，RTMonitor 对其进行捕捉实验.

5.4.1　栈基址缓冲池参数选取

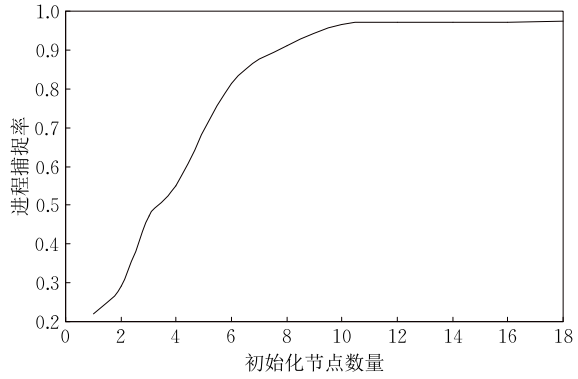
栈基址缓冲池的设定主要影响进程捕获率. 它的节点数量受缓冲池调度模块动态调整，能够与进程执行情形实现动态匹配，以实现使用较少的存储节点取得较高捕捉率的目的. 缓冲池的设定参数包括初始节点数量 α 和单次扩张节点数量 β. 实验以捕捉率作为参数选取的标准，测试不同的参数对进程捕捉率的影响. 实验过程中，首先将 β 固化为 0，即缓冲池不扩张，同时取消收缩机制，使初始化节点数量从 1 开始逐渐递增，观察进程捕捉率，实验结果如图 10(a-1)所示. 从图中可以看出当 α 取值为 5～10 时捕捉率增长最快，当 α 大于 10 时增长变缓. 因此将 α 设置为 10，能够达到以较小的存储开销取得较大捕获率的效果. 当 sched 需要创建新的 MapMem 时，WatchStack 生成栈基址的速度比 sched 读取栈基址的速度快，此时若缓冲池节点数量较少，则会造成栈基址还未被读取就被覆盖的现象，导致进程漏检.

测定单次扩张节点 β 时，α 设置为 10，同时开启收缩扩张机制，观察 β 递增时进程捕捉率的变化，实验结果如图 10(a-2)所示. 从图中可以看出，β 取值为 6 或更大值时捕捉率达到最大，因此将 β 设置为 6. 当缓冲池节点数量不足时，若 β 值设置的过小，会使缓冲池调度模块频繁地产生创建动作，影响执行效率，导致进程捕捉率受到影响.

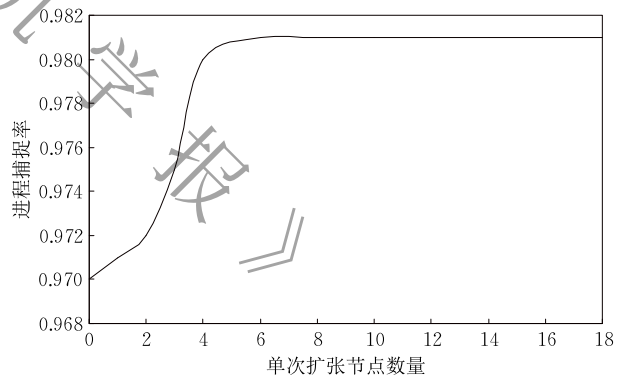
缓冲池开启自适应收缩与扩张后，能够确保缓冲池节点数量与进程执行情形的动态匹配，从而避免缓冲池节点不足造成的栈基址覆盖问题，也解决了存储节点带来的资源浪费问题，因此提高了进程捕捉率.

5.4.2　双向循环链表的参数选取

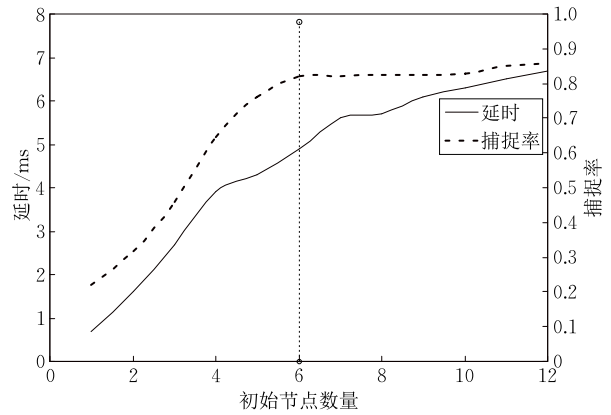
双向循环链表的节点数量会同时影响进程捕捉



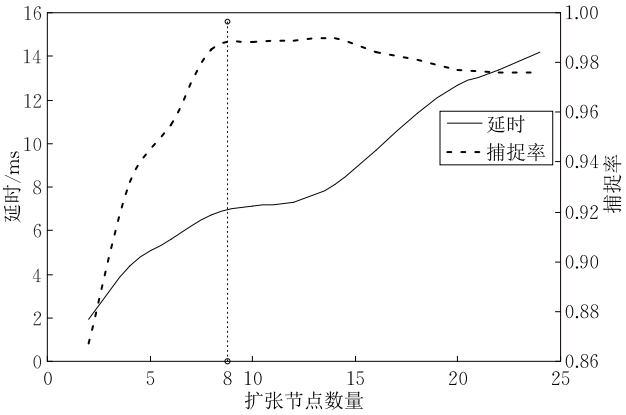
(a-1) α-捕捉率变化曲线



(a-2) β-捕捉率变化曲线



(b-1) 初始数量对RTMonitor性能的影响



(b-2) 扩张节点数量对RTMonitor性能的影响

图 10　栈基址缓冲池与双向循环链表的参数选取

《计算机学报》

率和捕捉延时,因此将捕获率和捕捉延时作为双向链表参数选取的标准.实验中首先将缓冲池的扩张和收缩机制关闭,测量初始化节点数量在递增过程中对进程捕捉率和捕捉延时的影响,如图 10(b-1)所示.

当进程切换频率增大时,若初始节点数量取值较小,则会由于节点内容覆盖导致内存原始数据丢失,从而影响进程捕捉率.初始节点数量取值较大时虽然能够提高进程捕捉率,但是一方面会增大捕捉延时,另一方面也会造成在进程切换频率较低时存在较多空闲节点的现象,导致存储资源浪费影响系统性能.从图中可以发现初始节点数量取值为 6 时进程捕捉率较大,捕捉延时较小,因此初始节点数量设定为 6.

RTMonitor 中双向循环链表的单次扩张的节点数量是影响进程捕捉率和进程捕捉延时的重要因素.实验中将初始化节点数量设置为 6,并开启缓冲池的自适应扩张和收缩机制,测试不同的扩张节点数目对进程捕捉率、捕捉延时的影响,测试结果如图 10(b-2)所示.

图中深灰色线是捕捉延时随扩张节点数量的变化曲线,虚线是捕捉率随扩张节点的变化曲线,左侧纵坐标轴表示延时,右侧纵坐标轴表示捕捉率.从图中可以看出,进程捕捉延时会随着扩张节点数量的增多而逐渐增大,进程捕捉率则呈先增大后减小的趋势.单次扩张的节点数量增大,导致非空节点等待时间变长,因此进程捕捉延时会随之增大.单次扩张的节点数量过少,会使扩张过程频率增大,Sem-Parse 创建和销毁动作频繁发生,影响系统整体性能;单次扩张的节点数量过多,会增大扩张时间,在扩张间隙产生进程漏检现象.所以节点数量过多或过少都会降低进程的捕捉率.根据图中结果,本文中将双向链表单次扩张的节点数量设置为 8,此时进程的捕捉延时较小,捕捉率较高.

双向链表采用自适应扩张和收缩机制后,能够保证存储节点数量与进程切换频率的动态匹配.动态扩张可以减少由于进程切换频率激增导致的存储节点不足的现象;动态收缩可以在进程切换频率较小时减少存储节点数量,减小系统存储开销.

5.5 隐藏进程的捕捉

5.5.1 内核级 rootkit 的隐藏性检测

为确保进程捕获的完备性,RTMonitor 对具有隐藏效果的进程要能够及时地检测到.实验以 adore-ng 和 linuxfu 两种不同类型的内核级 rootkit 为例进行捕捉验证.

Adore-ng 通过修改操作系统的系统函数,更改内核控制流,达到隐藏进程的目的,它对内部检测工具具有良好的隐藏效果.图 11 是对 adore-ng 隐藏进程的检测,并与 TVM 内部指令做结果对比.左侧为 TVM 内部指令所得状态,右侧为 SVM 内的 RTMonitor 所得的 TVM 实时状态.在 adore-ng 产生动作前,在 TVM 内部运行 1 号指令“ps -u”发现 4097 号进程 bash 与 ps 进程之间存在 4266 号进程 top.之后通过 2 号指令“./ava i 4266”将 pid 号为 4266 的 top 进程隐藏.最后,运行 3 号指令“ps -u”再次查看 TVM 运行状态,发现 4097 号进程 bash 与 ps 进程之间的 4266 号进程被隐藏起来了.RTMonitor 在 SVM 内部对 TVM 进行实时监视,发现在 TVM 内的 2 号指令进程 ava 运行前后均能检测到 4266 号进程 top,即对 adore-ng 的隐藏性检测有效.

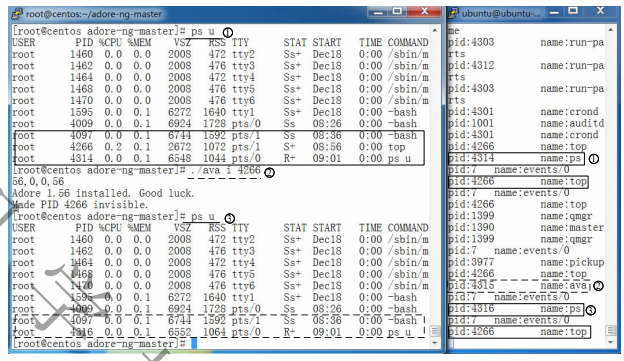


图 11 与内部工具的捕捉结果对比

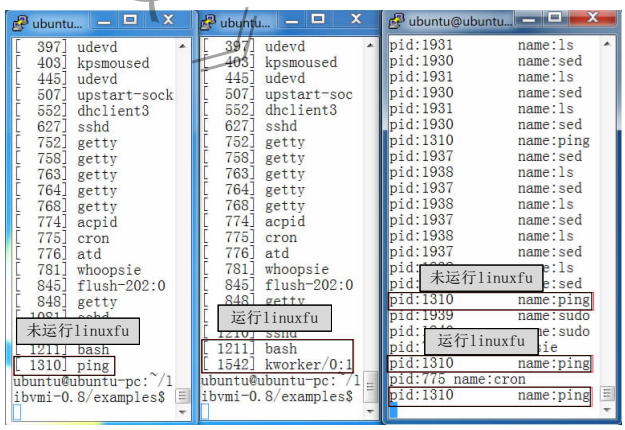


图 12 与 libVMI 的外部捕捉结果对比

Linuxfu 则是对内核对象进行直接操作,通过断开进程描述符中的双向链表(摘链操作)对进程进行隐藏,其针对外部检测工具具有良好的隐藏性.图 12 是 RTMonitor 对 linuxfu 隐藏进程的检测,并与虚拟机外部状态监测工具 libVMI 做对比.左侧窗口为未运行 linuxfu 时,在 SVM 内部运行 libVMI

对 TVM 内部进程的检测结果；中间窗口是运行 linuxfu 后，libVMI 在 SVM 内部对 TVM 进程的检测结果；右侧窗口是 RTMonitor 的实时检测结果. 左侧与中间窗口对比显示，linuxfu 对 libVMI 隐藏了 1310 号进程 ping；右侧窗口显示，无论 linuxfu 运行与否，RTMonitor 都能够监视到 1310 号进程.

除了上述两类典型的 rootkit 之外，RTMonitor 还对多种当下流行的内核级 rootkit 进行了隐藏进程的检测并与多种 rootkit 检测工具进行了对比，检测结果如表 3 所示. 对比发现，RTMonitor 对 rootkit 隐藏进程的检测效果良好，能够确保进程监视的完备性.

表 3　Rootkit 检测结果对比

	kbeast	suterusu	diamorphine	f00lkit	z-rootkit	adore-ng	linuxfu	xingyiquan
RTMonitor	√	√	√	√	√	√	√	√
chkrootkit 0.51	×	√	×	×	×	√	×	×
rkhunter 1.4.2	√	×	×	×	×	×	×	√
Avast 1.3.0	×	√	×	×	×	×	×	×
Avira 7.6.0	×	×	×	√	×	√	×	×
ClamAV 0.99.2	√	×	×	×	×	×	√	×
F-PROT 7.2.39	×	×	√	×	×	√	×	×

5.5.2　系统级恶意进程的捕捉

RTMonitor 除了对内核级 rootkit 的隐藏性具有良好的检测效果之外，对权限更高的系统级 rootkit 的隐藏性检测依然有效，本节以 Horse Pill33[①] 作为实验对象进行说明. Horse Pill 是 Michael Leibowitz 在 2016 年 Black Hat 上提出的一种新型的 rootkit，它通过感染 ramdisk 接管整个操作系统，并通过容器原语欺骗系统所有者，其对操作系统本身不仅具有隐藏性，还具有很强的欺骗性. 在感染过程中，Horse Pill 利用操作系统中 initrd 动态生成的特点向其中注入恶意二进制可执行文件 run-init，取代原有的 init，以此获取系统的控制权. 当操作系统重启之后，run-init 会向系统所有者提供提前设计好的虚假内部语义视图，这类似于将系统所有者圈定在攻击者为其设定的容器内. 在容器之外，攻击者可以自由设定后门之类的恶意进程，而这些恶意进程对系统所有者是不可见的.

RTMonitor 对 Horse Pill 的捕捉效果如图 13 所示. 图中左侧窗口是在 TVM 内部使用 top 指令捕获的系统内部视图，左侧是 RTMonitor 在 SVM 内对 TVM 内部的实时监视视图. 从图中可以看出，TVM 内部视图的 1 号进程是 systemd，而在 RTMonitor 捕获的实时视图中 1 号进程是 run-init，而且 systemd 进程 pid 并不是内部视图显示的 1，而是 328. 此外，外部实时视图中的 dnscat 不断地被创建执行，但它并未在内部视图中出现. 这说明内部视图中的 1 号进程 systemd 是虚假进程，它的真实“身份”是 328 号进程，而操作系统真实的 1 号进程是被隐藏了的 run-init.

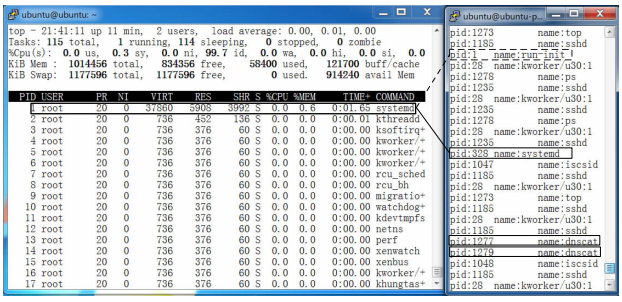


图 13　对系统级 rootkit 的捕捉结果

5.6　性能开销

图 14 为运行 RTMonitor 对 SVM 和 TVM 所产生的性能开销. 图中的所有测试结果均做了归一化处理，转化为相对性能的百分比，1 表示未运行 RTMonitor 时的系统性能，Y 轴代表性能负载，柱状图形越高，说明性能开销越大.

从图中可以看出，RTMonitor 对 SVM 和 TVM 均产生了不同程度的性能开销，且对 SVM 影响较大. 此外，从图中还可以发现，对 Pi 这类的 CPU 密集型应用产生的性能开销在 SVM 和 TVM 内分别为 21%和 17.6%；对 ReadFile 类的应用产生的性能开销分别为 15.2%和 15.1%；对其它普通应用产

① http://www.pill.horse/.

生的性能开销大概在 15%～18%之间. 与 RTKDSM 引入的性能开销对比(6 个数据集的监视开销超过 40%),RTMonitor 的开销相对较小.

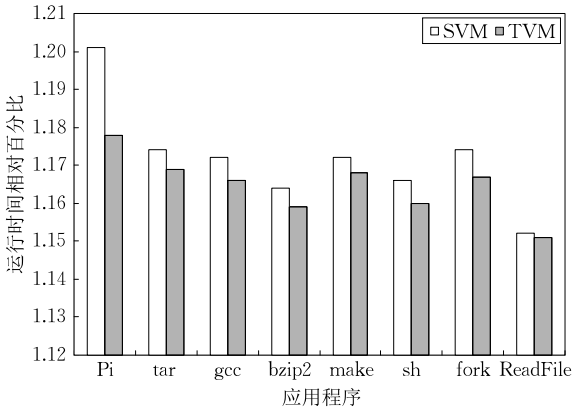


图 14 RTMonitor 产生的性能开销

RTMonitor 部署在 SVM 内,其所产生的计算负载和存储负载均由 SVM 处理,而对 TVM 采用非介入的方式进行监视,其性能开销主要来自整体性能损耗,所以对 TVM 的影响较小. RTMonitor 所产生的存储负载主要来自于两部分:一是存储内核栈基址的内存缓冲池;另一个是自适应收缩扩张的双向循环链表. 计算负载则主要由语义解析、与 hypervisor 的交互及自适应机制三部分产生. 由于 RTMonitor 需要实时监测进程切换并不断地进行内存映射操作,二者都要采用超级调用完成特权级的多次切换,计算资源消耗较多. 此外各类任务以、缓冲池和双向链表的调度操作都会影响系统的计算速度,因此对 CPU 密集新进程的执行影响较大.

6 结束语

本文提出了一种虚拟机进程实时监视技术 RTMonitor,通过监视进程资源调度避免了被绕过的风险,因此对各类隐藏进程具有较强的抗干扰性. 为应对进程执行的随机性和突发性特点,采用自适应机制,使得内存映射任务、语义解析任务及数据存储节点数量能够自适应地增加和减少,提高了进程捕捉率,降低了进程捕捉延时. 实验结果表明,RTMonitor 对运行周期较长的进程的捕捉率接近 100%,对在极端情况下的进程(4.2 节第一种执行情形)捕捉率超过 95%,捕捉延时控制在 2～18 ms 范围内.

目前,RTMonitor 还存在一些不足之处. 首先其性能开销较大,会对整个虚拟化平台造成影响. 此外,本文是基于 hypervisor 绝对安全的假设展开的,未涉及 hypervisor 安全性问题的讨论. 这些都将作为我们未来的工作进一步深入研究.

致 谢 感谢所有审稿专家对本文提出的宝贵意见.

参 考 文 献

[1] Riley R. A framework for prototyping and testing data-only rootkit attacks. Computers & Security, 2013, 37(9): 62-71

[2] Riley R, Jiang X, Xu D. Multi-aspect profiling of kernel rootkit behavior//Proceedings of the EUROSYS Conference. Nuremberg, Germany, 2009:47-60

[3] Cui Chao-Yuan, Wu Yun, Li Ping, et al. Narrowing the semantic gap in virtual machine introspection. Journal on Communications, 2015, 36(8): 31-37(in Chinese)
(崔超远, 乌云, 李平等. 虚拟机自省中一种消除语义鸿沟的方法. 通信学报, 2015, 36(8): 31-37)

[4] More A, Tapaswi S. Virtual machine introspection: Towards bridging the semantic gap. Journal of Cloud Computing, 2014, 3(1): 1-14

[5] Hebbal Y, Laniepce S, Menaud J M. Virtual Machine Introspection: Techniques and applications//Proceedings of the International Conference on Availability, Reliability and Security. Toulouse, France, 2015: 676-685

[6] Li Bao-Hui, Xu Ke-Fu, Zhang Peng, et al. Research and application progress of Virtual Machine Introspection technology. Journal of Software, 2016, 27(6): 1384-1401(in Chinese)
(李保琿, 徐克付, 张鹏等. 虚拟机自省技术研究与应用进展. 软件学报, 2016, 27(6): 1384-1401)

[7] Li Yong-Gang, Cui Chao-Yuan, Li Ping. Get the process content of guest OS based on VMI. China Computer Federation Magazine, 2016, 37(6): 1697-1700(in Chinese)
(李勇钢, 崔超远, 李平. 基于虚拟机自省的客户机进程内容获取. 计算机工程与设计, 2016, 37(6): 1697-1700)

[8] Wang G, Estrada Z J, Pham C, et al. Hypervisor introspection: A technique for evading passive virtual machine monitoring//Proceedings of the Usenix Conference on Offensive Technologies. Washington, USA, 2015: 12

[9] Vogl S, Kilic F, Schneider C, et al. X-TIER: Kernel module injection//Network and System Security. Berlin, Germany: Springer, 2013: 192-205

[10] Carbone M, Conover M, Montague B, et al. Secure and robust monitoring of virtual machines through guest-assisted introspection//Proceedings of the International Workshop on Recent Advances in Intrusion Detection. Berlin, Germany, 2012: 22-41

[11] Dolan-Gavitt B, Leek T, Zhivich M, et al. Virtuoso: Narrowing the semantic gap in Virtual Machine Introspection//Proceedings of the Security and Privacy IEEE. Oakland, California, USA, 2011: 297-312

[12] Fu Y, Lin Z. Space traveling across VM: Automatically bridging the semantic gap in Virtual Machine Introspection via online kernel data redirection//Proceedings of the 2012 IEEE Symposium on Security and Privacy. San Francisco, California, USA, 2012: 586-600

[13] Jiang Xu-Xian, Wang Xin-Yuan, Xu Dong-Yan. Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. Acm Transactions on Information & System Security, 2010, 13(2): 128--138

[14] Hizver J, Chiueh T C. Real-time deep virtual machine introspection and its applications. Acm Sigplan Notices, 2014, 49(7): 3-14

[15] Chisnall D. The Definitive Guide to the Xen Hypervisor. New Jersey, USA: Prentice Hall Press, 2013: 10-13

[16] Liu Y, Xia Y, Guan H, et al. Concurrent and consistent virtual machine introspection with hardware transactional memory//Proceedings of the IEEE, International Symposium on High PERFORMANCE Computer Architecture. Orlando, USA, 2014: 416-427

[17] Xiao J, Lu L, Huang H, et al. Hyperprobe: Towards virtual machine extrospection//Proceedings of the 29th Large Installation System Administration Conference (LISA15). Washington, USA, 2015: 1-12

[18] Xu Y, Bailey M, Noble B, et al. Small is better: Avoiding latency traps in virtualized data centers//Proceedings of the Symposium on Cloud Computing. Santa Clara, USA, 2013: 1-16

[19] Milenkoski A, Payne B D, Antunes N, et al. Experience report: An analysis of hypercall handler vulnerabilities//Proceedings of the IEEE International Symposium on Software Reliability Engineering. Naples, Italy, 2014: 100-111

[20] Shi Lei. Xen Virtualization. Wuhan: Huazhong University of Science and Technology Press, 2009: 65-67(in Chinese)
(石磊. Xen 虚拟化技术. 武汉：华中科技大学出版社，2009：65-67)

[21] Guo Xu. Professional Linux kernel architecture//Professional Linux Kernel Architecture. Beijing: Posts & Telecom Press, 2008: 220-230(in Chinese)
(郭旭(译). 深入 Linux 内核架构. 北京：人民邮电出版社，2010：220-230)

[22] Suneja S, Isci C, De Lara E, et al. Exploring VM introspection: Techniques and trade-offs//Proceedings of the ACM Sigplan/Sigops International Conference on Virtual Execution Environments. Istanbul, Turkey, 2015: 133-146

[23] Chen Li-Jun. Understanding the Linux Kernel. Second Edition. Beijing: China Electric Power Press, 2001: 89(in Chinese)
(陈莉君等(译). 深入理解 Linux 内核. 第 2 版. 北京：电力出版社，2001：89)

CUI Chao-Yuan, born in 1972. Ph. D., professor. His main research direction includes system virtualization, cloud computing, information and communication security.

LI Yong-Gang, born in 1988, Ph. D. candidate. His current research interests include operating system, information security and cloud computing.

WU Yun, born in 1974, Ph. D., associate professor. Her main research direction is artifical intelligence.

SUN Bing-Yu, born in 1974, Ph. D., professor. His main research direction is pattern recognition.

《计算机学报》

Background

Virtualization is the key technology to support cloud computing, and its security is particularly important. This article belongs to the field of virtualization security research. Traditional security tools are mostly host-based, resulting in the lack of adequate isolation between the target operating system and security tools. As a result, once the target machine is infected, the security tool runs in untrusted environment, causing its presence to be bypassed or spoofed. For virtualized environments, security tools can be placed outside of the virtual machine, providing a good isolation between them. At present, the "out-of-box" technology uses the method of memory snapshot for a single or periodic detection. This causes the monitored process to be discontinuous leading to a higher undetected probability. The time-based detection method is similar to single-memory snapshot, and the internal state of the virtual machine is checked at a certain moment for analysis. The detection method based on periodic scan captures memory at regular intervals, with longer intervals between the two scans. However, the malware's activation time is random. If malware has not been started or completed during the scan time or scan interval, the security tool will not be able to detect malicious activity.

In this paper, a real-time monitoring method of virtual machine process based on adaptive mechanism RTMonitor is

proposed. RTMonitor is deployed outside of the virtual machine to monitor its internal state in real time.

The implementation of processes has the characteristics of randomness and suddenness. These characteristics put forward high requirements on RTMonitor. RTMonitor uses adaptive mechanism to overcome the process of sudden and random nature. RTMonitor adopts the action separation strategy, which separates the process switching capturing module, the memory capturing module and the semantic analysis module into independently implementing entities that are logically related to each other. Process memory capturing entities and semantic analysis entities are implemented through the multitasking concurrent mechanism, and they are dynamically expanded or contracted according to process numbers. To compensate for speed differences among execution entities and reduce missed inspections, RTMonitor has established a caching mechanism that automatically shrinks and expands, caching process context and memory. To monitor process switching RTMonitor captures the process context switching action in real time and treats it as a process switching flag. In a word, RTMonitor captures the virtual machine process context switching rapidly to monitor process switching continuously. And then it gets process memory raw data through the memory mapping. Finally, RTMonitor translates the original memory data into high-level semantics based on semantic knowledge library.

RTMonitor can monitor processes running in milliseconds. The capture rate is more than 95% when processes run less than 10 ms, and it is close to 100% when processes run more than 10 ms. At the same time, RTMonitor controls the capture delay 2—15 ms. In addition, RTMonitor can detect the processes hidden by all kinds of rootkits, ensuring the completeness of detection.

This paper is supported by the National Natural Science Foundation of China (No. 31371340) and the National Key Technologies Research and Development Program of China (No. 2016YFB0502600).

《计算机学报》