利用虚拟化平台进行内存泄露探测

汪小林[1] 王振林[2] 孙逸峰[1] 刘 毅[1] 张彬彬[1] 罗英伟[1]

[1]（北京大学信息科学技术学院 北京 100871）

[2]（密歇根理工大学计算机科学系 霍顿 密歇根 49931 美国）

摘 要 文中利用虚拟机管理器，透明地记录应用程序对资源的申请、释放以及使用情况，提供了探测内存泄露的辅助信息. 此机制首先不需要修改或重新编译源程序；其次，带来的性能损失很小. 两者结合可以构建在线内存泄露探测和汇报机制. 不仅如此，基于虚拟机环境的内存泄露探测还具备通用性，且不需要特殊的硬件支持. 所有这些特性，是已有的解决方案所不能兼有的. 实验结果表明：基于虚拟机环境的内存泄露探测机制具有实用性，性能损失也被控制在 10% 以内，能够运用在实际的生产环境中.

关键词 内存泄露探测；虚拟机；虚拟化平台；虚拟机管理器

中图法分类号 TP311 **DOI 号**：10.3724/SP.J.1016.2010.00463

Detecting Memory Leak Via VMM

WANG Xiao-Lin[1] WANG Zhen-Lin[2] SUN Yi-Feng[1] LIU Yi[1] ZHANG Bin-Bin[1] LUO Ying-Wei[1]

[1]（*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871）

[2]（*Department of Computer Science, Michigan Technological University, Houghton, MI* 49931, *USA*）

Abstract In this paper, virtualization technology is utilized to transparently record the allocation and release of memory resources applied by applications running on virtual machine (VM), and these records provide the auxiliary information to detect memory leaks hiding in the binary code. Firstly, this mechanism does not require source code modification or recompilation; secondly, the performance overhead is very small, which makes it possible to build online memory leak detection and reporting mechanisms, free application developers from designing test suite and improve the chances of finding more memory leaks. Besides, memory leak detection based on the virtual environment also provides versatility without needing special hardware supports: not only user-mode application can be detected, but the operating system kernel; and both Linux and Windows are supported. Existing research cannot bring all of these features together. The experimental results show that the memory leak detection mechanism has a limited performance overhead which is less than 10%, and the information produced by the mechanism can help programmer to track out the possible memory leaks efficiently.

Keywords memory leak detection; virtual machine; virtualization platform; virtual machine monitor

---

收稿日期：2009-04-20；最终修改稿收到日期：2009-09-16. 本课题得到国家"九七三"重点基础研究发展规划项目基金(2007CB310900)、国家自然科学基金(90718028，60873052)、国家"八六三"高技术研究发展计划项目基金(2008AA01Z112)、教育部-英特尔信息技术专项科研基金(MOE-INTEL-08-09)资助. **汪小林**，男，1972 年生，博士，副教授，主要研究方向为系统虚拟化、地理信息系统. **王振林**，男，1970 年生，博士，副教授，主要研究方向为体系结构、编译、系统虚拟化. **孙逸峰**，男，1983 年生，博士研究生，主要研究方向为系统虚拟化. **刘 毅**，男，1985 年生，硕士研究生，主要研究方向为系统虚拟化. **张彬彬**，女，1982 年生，博士研究生，主要研究方向为系统虚拟化. **罗英伟**（通信作者），男，1971 年生，博士，教授，主要研究领域为系统虚拟化、地理信息系统. E-mail：lyw@pku.edu.cn.

1 引 言

内存泄露是指被申请的内存资源在程序运行的某一时刻后再也不被使用和释放.如果被泄露的是虚拟内存,则此程序本身能够使用的虚拟内存空间因此变少;如果被泄露的是物理内存,则整个系统减少了能够使用的物理地址.内存泄露会使应用程序申请动态内存失败,导致服务中止,严重时会导致整个系统因资源耗竭而崩溃.对于运行时间很短的程序,内存泄露一般不是问题,但是对于长期运行的程序,例如运行在服务器上的服务和操作系统本身,内存泄露会带来严重的后果,可能会导致系统服务中止.一直以来,内存泄露都是造成计算机安全事故的主要原因之一.

一些编程语言,如 Java,通过垃圾回收等方式,自身提供了内存回收的机制.这种机制不但不能保证消除内存泄露,而且还会带来性能的损失.而另外执行效率很高的一些编程语言,如 C 和 C++,则将内存分配和释放的操作完全交付给程序员.在逻辑非常庞大的程序中,内存泄露很难避免.因为内存泄露探测的重要性,先前已经有很多这方面的工作.这些工作基本分为两类:第一类是静态检查程序语义.这类方法认为,正确的程序代码,应该符合预定的规则,例如,通过 malloc 函数申请的内存,在接下来运行的所有代码分支都应该有一个 free 函数.可将代码的路径抽象为布尔限制路径,对动态内存的指针加以跟踪.对于大型的工程,这个方法比较耗时,例如分析 GNU/Linux 的内核代码需要一个处理器一整天的时间[1].也有将代码抽象为一个变量流通图的,图的边表示代码分支[2].这类方法直接对代码进行分析,实现复杂,目前还有一些难点尚未解决,例如循环的处理[1].另一类是运行时动态检测法.即在程序运行时记录程序动态分配的内存资源和释放信息,然后分析是否存在内存泄露.例如 Purify① 和 SafeMem[3],都属于这一类.动态检测法受限于测试程序,因为测试程序所覆盖的代码非常有限,无法激发出所有潜在的问题.

本文通过虚拟机管理器(Virtual Machine Monitor,VMM)平台,在其上虚拟机(Virtual Machine,Guest OS,VM)运行时,动态截获虚拟机中申请和释放内存的函数,并记录下来,用于辅助甄别内存泄露,是基于第二类的方法.通过内存虚拟化技术的协助,我们可以监控应用程序对这些内存资源的应用情况.然后,应用一些规则,找出内存泄露的嫌疑.例如长时间未被释放且未被访问的内存可能是内存泄露.这种方法相对现有的方法有以下优点:

(1)既不需要修改被探测程序的源代码,也不需要重新编译,为被测试代码提供了透明性.如 Purify 和 SafeMem,前者通过编译插入指令以获取应用程序访问内存的所有行为,然后这些行为被用于判断内存泄露和内存访问地址越界等问题.SafeMem 则需要重新封装资源申请和释放函数,甚至需要给操作系统添加新的系统调用,这些需求一定程度上不利于它们的应用.其次,如果探测内存泄露方法给应用程序带来很多性能损失或者占用很大额外资源的话,其应用只能仅限于程序调试阶段,其可发现的内存泄露受限于测试用例.Purify 虽然能够捕捉到程序中大量的内存访问,但是,因为需要截获应用程序所有的内存操作,Purify 导致应用程序的性能损失很大,通常应用程序的性能降低达到 2~3 倍,或者更多①.性能的降低限制了其使用范围,而且加重了其对测试程序的依赖.本文中的内存泄露探测机制是基于虚拟机来实现的,经测试其性能损失少于 10%,能够在真实服务中使用.这同时带来的另外一个优点在于,在真实使用中,代码的执行覆盖范围更广,能够发现潜在的内存泄露.

(2)本文方法不但适用于用户态的应用程序,而且还适用于操作系统内核.存在于内核中的内存泄露,比应用程序中的内存泄露带来的危害更大,而且调试更困难.尽管有很多工具用于测试用户态程序的内存泄露,但是调试内核代码的工具仍旧比较缺乏.PinOS[4] 在虚拟机管理器之上,通过软件动态翻译的方法,提供了调试操作系统内核代码的机制.然而,软件动态翻译使得其性能下降很大,大多数情况下降低到 50~60 倍,这限制了其使用环境,难以在真实环境中使用.而且,其运行时需要插入部分代码到虚拟机的内核地址空间,因此占用操作系统的一部分地址空间,这破坏了透明性.

本文基于虚拟机管理器的实现也提供了平台通用性,其既适用于 GNU/Linux 操作系统,也适用于 Windows 操作系统.对操作系统的版本也不做要求,只要其代码体系符合 Intel X86 的指令规范.而 SafeMem 虽然性能降低很小,但是需要给操作系统添加新的系统调用接口和系统调用处理函数,因此欠

① Rational Purify. Purify: Fast Detection of Memory Leaks and Access Errors. http://www.ibm.com/software/awdtools/purify/

缺了平台的通用性.最后,本论文的机制不需要特殊硬件的支持,而SafeMem是基于ECC控制器实现的.

接下来,我们首先总体介绍利用虚拟化平台进行内存探测的机制;其次详细介绍对运行在虚拟机中的应用程序申请和释放内存资源的捕获和记录,然后分析应用程序所申请的内存资源,选择出存在内存泄露的嫌疑部分进行监控.在实验章节中,本文描述了基于虚拟平台进行内存泄露探测的有效性和性能代价.最后是结论和下一步工作.

2 实现原理

内存泄露是指已经被申请的内存资源没有被合理地释放,而导致这部分资源不能被系统重新利用的一种现象.内存泄露广泛存在或者隐藏于程序之中,被泄漏的内存资源不能被重新利用且不再被访问.对于应用程序,内存泄露会使得应用程序的虚拟内存空间耗竭,导致任务的中断和失败.对于操作系统,内存泄露不仅会减少可用的内核虚拟内存地址空间,而且还会不断蚕食整个系统可用的物理内存页面,导致整机不得不重新启动.

内存申请和释放的接口具有统一、简单的特色,例如Windows提供的动态内存管理接口GlobalAlloc和GlobalFree、Posix规定的malloc和free接口、Linux内核的vmalloc和vfree以及get_page和put_page等接口.某些应用程序实现了自身的内存管理,但是其接口也是比较统一的.这些接口不但统一简单,其数量也非常有限.正常情况下,通过某个申请函数获得资源需要通过对应的释放函数来释放.本文利用这一点,在虚拟化平台中透明地截取了资源申请和释放的接口,维护一个动态资源使用列表,详尽和全面地掌握被监控对象使用内存资源的情况.

如果有内存泄露存在的话,这个列表理论上包含了所有被泄漏的内存,但又不仅限于被泄漏的内存,因为被虚拟机正确使用的动态内存资源也包含在这个列表中.接下来的工作是从这个列表中选择出最有可能是内存泄露的项.选择的主导思想是:(1)某个地址持续地分配长时间不被释放的内存;(2)选择被占用但长时间未被访问的内存.前者可以从记录获取,实现后者需要分两步:首先确定被监控的对象,然后通过内存虚拟化技术进行监控.

在接下来的几个小节中,详细介绍每个步骤的设计原理和方案.

2.1 截获资源申请和释放

Purify和SafeMem等实现方案通过修改源代码或者修改编译器的方式,来截取资源申请和释放;而通过虚拟化平台可以更简单更透明地实现这个功能.我们以malloc和free为例来说明,下面列出了一小段常见的代码.

```
void *PTR=malloc(LEN);
if (PTR==NULL) err("Out of memory.");
/ * utilizing the memory starting at PTR of size LEN * /
if (PTR) free(PTR);
```

这段C语言代码首先通过malloc函数申请长度为LEN个字节的动态内存,然后判断是否申请成功.如果成功,则使用这些内存资源进行计算.最后,当不再需要PTR所指的资源时,通过free函数释放这些内存.在这个场景中,我们不仅需要截获malloc函数的调用,获取其参数和返回结果,而且也需要截获free函数的调用,获取其参数,也即被释放的动态资源.换句话说,应用程序当前正在使用的动态内存信息(包括其起始地址、长度、时间)和调用IP(Instruction Pointer)地址都需要在虚拟机管理器中维护.

在虚拟化平台中,可以通过替换指令的方式,透明地截获申请函数和释放函数.为了实现透明的结果,首先分析一下上面C语言程序在Linux操作系统之上对应的汇编代码:

```
8048426: 89 04 24              mov   %eax,(%esp)
8048429: e8 de fe ff ff        call  804830c <malloc@plt>
804842e: 89 45 fc              mov   %eax,0xfffffffc(%ebp)
8048431: 83 7d fc 00           cmpl  $0x0,0xfffffffc(%ebp)
8048435: 75 18                 jne   804844f <main+0x4f>
8048437: c7 04 24 54 85 04 08  movl  $0x8048554,(%esp)
804843e: e8 e9 fe ff ff        call  804832c <printf@plt>
8048443: c7 04 24 01 00 00 00  movl  $0x1,(%esp)
804844a: e8 ed fe ff ff        call  804833c <exit@plt>
804844f: 83 7d fc 00           cmpl  $0x0,0xfffffffc(%ebp)
8048453: 74 0b                 je    8048460 <main+0x60>
8048455: 8b 45 fc              mov   0xfffffffc(%ebp),%eax
8048458: 89 04 24              mov   %eax,(%esp)
804845b: e8 ec fe ff ff        call  804834c <free@plt>
```

可以看到,在地址为0x8048426的地方,变量LEN中保存的数值被压到栈上,接下来的一条指令调用了malloc函数,程序的运行跳转到malloc函数所在的地址0x804830c.在malloc函数返回后,程序在地址0x804842e处继续执行.如果malloc函数申请资源成功,程序最终会在地址0x804845b处调用free函数,释放这段内存空间.

内存资源申请函数和释放函数,例如malloc和free,作为系统的调用接口,其在内存中的地址很容

易捕获得到，在上面的示例程序中，malloc 和 free 的地址分别为 0x804830c 和 0x804834c. 获得内存申请函数和释放函数的地址之后，截获其执行需要如下几个步骤：

1. 在函数地址处(例如 0x804830c 处)插入能够使得虚拟机无条件陷入到虚拟机管理器的指令. 例如非法指令，在 Intel 的 VT 平台中可以插入 VMCALL 指令. 我们把这种指令称为陷入指令，原位置被替换的指令称为被替换指令，把被植入陷入指令的函数称为受监控函数.

2. 运行在虚拟机中的应用程序调用受监控函数(如 malloc)之后，虚拟机马上陷入虚拟机管理器.

3. 在虚拟机管理器中，分析受监控函数的类别，分析其参数，获取函数调用栈，更新应用程序占用内存资源的信息. 如果是资源申请函数则增加内存占用信息，而资源释放函数则需要删除相应的内存信息. 例如，如果陷入的函数是 malloc，其只有一个参数表明申请内存的长度，我们可以在当前应用程序的堆栈上获得. 如果陷入的是 free 函数，其参数表明需要释放的内存. 这里获取函数调用栈的目的是为了程序员调试的方便.

4. 在虚拟机管理器中模拟执行被替换指令. 之所以不恢复被替换指令，原因是为了保证能监控到应用程序并发调用该受监控函数，例如多线程程序中多个线程有可能同时通过调用 malloc 函数申请内存资源.

5. 如果该受监控函数在申请资源，在陷入指令的下一条指令处(在上述例子中，地址 0x804842e 处)，需要插入陷入指令，以便让我们获悉函数调用结束，分析申请结果.

6. 虚拟机管理器命令虚拟机继续执行，虚拟机会执行受监控函数的其它代码，执行结束后，返回到调用处，上例中监控 malloc 的第一次调用，则是地址 0x804842e 处.

7. 返回处陷入，因为步 5 已经将一个陷入指令插入到返回处；在虚拟机管理器中，分析受监控函数返回的结果. 对于 malloc 函数，其返回结果是所申请到内存资源的地址，如果申请失败，则返回值是 NULL.

通过以上方法获取应用程序的动态内存资源使用情况之后，我们即时地维护应用程序申请动态内存的列表. 如果应用程序中存在内存泄露的话，被泄露的内存必然在我们所维护的表中. 从动态内存列表中精确找出被泄露的内存项是一个难点；然而，我们可以先找到疑似泄露的动态内存项，然后将信息(函数调用栈、调用地址和申请资源大小等)交由程序员处理. 本论文的目标是，找到内存泄露的嫌疑项，并提高准确率. 除此之外，通过本文的方法，还可捕捉到程序中多次释放资源的错误操作，例如将一段动态内存调用 free 释放后，又再次调用 free 释放，根据 free 接口的规范，其结果是不可预料的，也会对程序的正常运行带来隐患.

本小节阐述了截获内存申请和释放的机制，接下来的两个小节中，将着重介绍推选内存泄露嫌疑项的规则和步骤以及提高推选准确率的一些策略和方法.

2.2 判断内存泄露的规则

2.1 节中介绍了截获应用程序申请和释放动态内存的机制. 这个机制能够使得虚拟机管理器维护应用程序当前使用的全部动态内存信息，这些信息包括受监控函数的调用 IP 地址、动态内存申请函数的种类、所申请动态内存的地址和大小以及申请时间等. 如果应用程序存在内存泄露的话，那么被泄露的内存就隐藏在我们维护的列表中. 我们判断是否存在内存泄露的嫌疑基于如下几条规则：

(1) 在应用程序运行的过程中，有长时间未被应用程序访问的动态内存；这里的运行是指应用程序实际占用处理器的状态，不包括程序阻塞等待的状态；

(2) 应用程序的某一地址持续申请动态内存且部分不释放，导致应用程序占用的动态内存随着运行时间的延续越来越多；

(3) 在应用程序退出时，有未被释放的内存段.

规则 1 基于被泄漏内存的最基本的特征：应用程序在语义层面上，丢弃了内存段并不再访问该内存段. 规则 1 需要虚拟机管理器对应用程序的内存的访问情况进行监控. 这种监控行为是需要虚拟机管理器付出性能代价的，因为在监控中，会导致虚拟机管理器对虚拟机正常运行添加额外的干预和陷入，我们在下一小节中会详细描述监控的步骤. 监控应用程序所申请的所有内存资源的访问是不太实际的，因为这有可能引起大量的虚拟机陷入，严重影响虚拟机管理器的整体性能. 为了克服这一点，从而减少虚拟机管理器性能的损失，应当降低被监控内存片断的数目. 在本文的设计中，监控机制和监控策略是分开的. 我们在虚拟机管理器中添加了监控机制，而决定谁被监控、谁不被监控、被监控内存片断的数目等策略性决定则在高层动态设定. 监控策略会根据动态内存的监控历史、分配时间和虚拟机的性能反馈等因素进行判断.

规则 2 是基于统计的，不需要监控动态内存的被访问情况，因此开销比规则 1 少，但是准确性比规则 1 差. 此规则基于这样的观察，内存泄露的点具有重复性，在某一 IP 点上如果发生过内存泄露，那么在此点上还会发生内存泄露.

规则 3 基于对良好编程原则的考虑：程序应该在退出时，已经优雅地释放掉所有申请的资源，而不

是将这些清理工作托付给操作系统.

总之,我们根据应用程序对其所申请的资源使用情况的统计,来推测是否有内存泄露发生. 规则 1 基于如下考虑:因为被泄露的内存的特征之一是不再被应用程序访问,所以最好的嫌疑者是那些在应用程序的运行中,长时间没有被访问的动态内存. 规则 2 基于如下考虑:程序的执行线随着上下文的不同而不同,在某个执行线上程序员可能会忘记释放资源. 如果程序不断经过这个忘记释放资源的执行线,则会不断产生内存泄露. 规则 3 基于编程习惯.

2.3　监控内存

本小节主要描述在虚拟机管理器中,实现对应用程序所申请的动态内存的访问进行监控的机制. 根据判断内存泄露的规则 1,在程序运行期间长时间未被有效利用的动态内存资源,可能是被泄漏的内存. 为了了解动态内存资源被利用的情况,需要对该动态内存资源进行访问监控. 监控的基本思想是,在应用程序访问被监控的动态内存时,虚拟机管理器能够获悉这一访问行为. Safemem 利用硬件 ECC 实现了监控,而虚拟机管理器不需要额外的硬件支持,通过内存虚拟化即可实现监控. 本小节内容首先着重描述基于影子页表的内存虚拟化方法,其次描述在内存虚拟化方法中实现内存监控的原理和方法,最后分析在虚拟机管理器中内存监控的有效性.

2.3.1　内存虚拟化方法

虚拟机与传统计算机相比,其内存系统多了一种地址,共包括以下 3 种地址:

机器地址(Host Physical Address,HPA),指真实硬件的机器地址,即地址总线上应该出现的地址信号;

物理地址(Guest Physical Address, GPA),指经过 VMM 抽象的、虚拟机所看到的伪物理地址;

虚拟地址(Guest Virtual Address, GVA),指 Guest OS 提供给其应用程序使用的线性地址空间.

显然,VMM 的内存模块负责完成物理地址到机器地址的映射,我们将这个映射记为 f;同时,Guest OS 的内存管理模块要完成虚拟地址到物理地址的映射,我们将这个映射记为 g. 于是,虚拟地址、物理地址和真实地址之间的关系如图 1 所示.

图 1　机器地址、物理地址和虚拟地址的关系

在没有硬件内存虚拟化支持的情况下,KVM 实现内存虚拟化的方法是影子页表技术[5-6]. 影子页表技术为 Guest OS 的每个页表维护一个“影子页表”,并将合成后的映射关系写入到“影子”中,Guest OS 的页表内容则保持不变. 最后,VMM 将影子页表交给内存管理模块(Memory Management Unit, MMU)进行地址转换. 试举一例说明:假设一个运行在 Guest OS 中的进程,当其访问 0x12345678 这个虚拟地址时,因为在影子页表中尚未为其建立页表项,所以会产生缺页异常,使得 Guest OS 陷入到虚拟机管理器中. 虚拟机管理器首先查看 Guest OS 的页表中是否存在有效的映射 g. 如果不存在,则将异常交付给 Guest OS 处理. 如果存在,虚拟机管理器在影子页表项中直接建立映射 $f \cdot g$,从虚拟地址页 0x12345000 转换为物理页的地址. 当此影子页表项被清除时,虚拟机管理器再负责将影子页表项中的 Dirty 位和 Access 位同步到 Guest OS 的页表中去.

2.3.2　内存监控的原理和方法

内存监控的一个选择是通过管理虚拟机的影子页表实现. 在虚拟机运行时,内存管理模块载入影子页表,试图完成从 GVA 到 HPA 的映射. 如果影子页表中已经存在某个 GVA 到其 HPA 的映射,那么这个转换过程会自动完成;否则,虚拟机会陷入到虚拟机管理器中,由后者完善 GVA 到 HPA 的映射. 基于影子页表的这种特征,通过如下步骤即可完成对内存的监控:

首先,计算出内存片断所跨越的所有页面,例如在页面大小为 4KB 的虚拟机中,首地址为 0x80a8400,长度为 10KB 的内存片断,跨越了 3 个页面:其中占第 1 个和第 3 个页面的部分,完全占用第 2 个页面(如图 2(a)所示). 接着,在影子页表中消除对这些页面的 GVA 到 HPA 的映射关系. 因为这些页面的映射关系被消除,那么只要虚拟机试图访问该内存片断,都会导致虚拟机的陷入(如图 2(b)所示). 最后,在虚拟机陷入时,分析陷入指令所访问的内存地址是否属于该内存片断.

通过内存虚拟化机制实现的内存监控,是基于页面级,即所监控的最小内存单位是内存页. 内存页通常大小是 4KB,也有其它大小,例如 2MB 和 4MB. 在进行监控时,虚拟机对页面范围内的所有内存地址访问都会导致陷入,因此,监控的单位大小越小越好.

基于硬件 ECC 的监控是更细粒度的监控机制,被 SafeMem 所采用. 如果虚拟机管理器存在 ECC

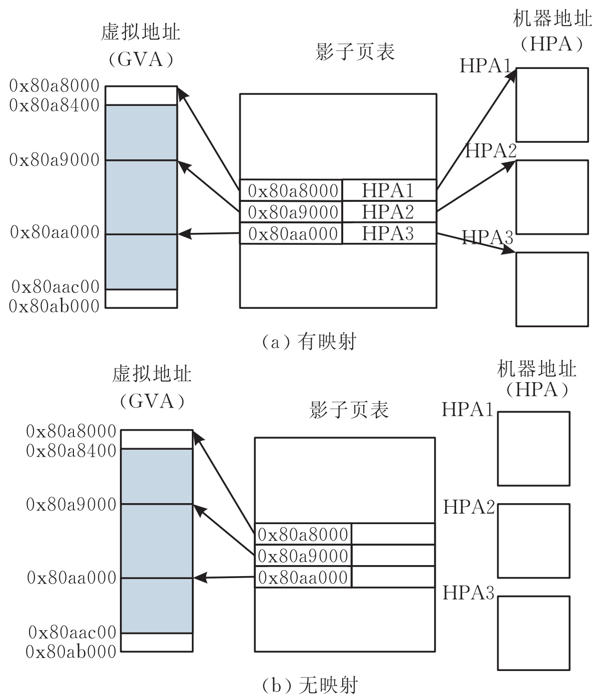

图 2 内存监控原理

的支持，也能够实现 Cache line 大小粒度的监控，SafeMem 所使用的方法同样适用于虚拟机管理器.

这种内存监控的原理，也适用于其它内存虚拟化方法，例如直接页表访问. 即使是硬件辅助内存虚拟化技术，例如 EPT(Extended Page Tables)[7] 或者 NPT(Nested Page Tables)[8]，在其上实现内存监控，核心思想是通用的，即从虚拟机管理器中，消除虚拟机从 GVA 到 HPA 的内存地址映射关系.

2.3.3 虚拟机管理器监控内存的有效性分析

虚拟机管理器监控内存引入的性能开销主要源于截获资源申请和释放的函数和影子页表导致的虚拟机陷入. 每次陷入的开销公式如下：

$$\begin{aligned}\text{COST[陷入]}=&\text{COST[VM 切换到 VMM]}+\\&\text{COST[分析+模拟指令]}+\\&\text{COST[VMM 切换到 VM]}+\\&\text{COST[Cache]}+\text{COST[TLB]}.\end{aligned}$$

虚拟机和虚拟机管理器之间的切换会引入很大的性能开销. 首先，切换本身会耗费很多的处理器周期，因为在切换过程中，处理器不仅需要保存原状态，而且需要载入新状态. 其次，这种切换还会导致额外的 Cache 失效，因为虚拟机管理器和虚拟机共享 Cache 且代码和数据并不相同. 最后，TLB (Translate Lookaside Buffer)[7] 被刷新的开销. TLB 缓存着虚拟地址到物理地址的映射，避免了 MMU 在地址映射时对存放在内存中的页表的重复遍历访问. 因为虚拟机管理器和虚拟机使用不同的页表，所以每次切换都会导致 TLB 的刷新操作，这个开销是非常大的. 然而，随着硬件的不断发展，新的技术，如 VPID(Virtual Process ID)[7]，能够避免或减少在虚拟机管理器和虚拟机之间切换时 TLB 的刷新，从而降低 TLB 刷新的开销.

截获资源申请和释放的函数的开销与这些函数调用点的次数成正比. 性能的主要开销在于对内存使用进行监控. 虚拟机管理器监控动态内存的访问情况，会导致额外的附带开销. 为了监控占用部分内存页面的动态内存，需要监控整个页面被访问情况. 如果该页面中，动态内存片断之外的内存页面当前被虚拟机监控器频繁使用，则对该页面的监控因为频繁地陷入会导致虚拟机性能的急剧下降. 在制定监控策略时，应当避免监控这种页面.

在实际应用程序运行时，如 malloc 等动态内存申请方法所申请的动态内存一般处在应用程序的堆栈上，该动态内存地址的前后内存很有可能也属于动态内存，因此，对一个动态内存的监控，有可能会附带监控其它多个动态内存，这个特性对我们监控内存的访问是有好处的，因为同样的性能损失，可以完成对多个动态内存的监控.

为了提高虚拟机管理器监控内存的效率，除了避免对热页面的监控，还应当减小监控粒度. 对于页面级的监控，4KB 大小的页面要优于 2MB 大小的页面，因为虚拟机对前者的访问次数一般要少于对后者的访问次数. 如果硬件提供更细粒度的机制，应当优先使用硬件提供的机制.

3 在 KVM 平台上的实现

我们在 KMV-84 虚拟机管理器之上实现了内存泄露的探测. KVM 是基于 GNU/Linux 实现的虚拟机管理器，以模块的形式运行在操作系统中. 虚拟机表现为操作系统中的一个 QEMU① 进程，该进程通过和 KVM 模块的交互，实现了处理器虚拟化、内存虚拟化以及部分硬件的虚拟化；其它硬件设备如网卡和外存的虚拟化，由 QEMU 实现. 因为 KVM 是操作系统内部的一个模块，所以其调试和运行非常方便，通过模块的动态加载和卸除即可实现更新，这点是 Xen[6] 无法提供的，在其上每次修改都需要

① http://www.nongnu.org/qemu/

重启硬件.

本实验中，虚拟机使用的操作系统是比较主流的 Redhat 4.1.1-52 版本. 接下来讨论实现中的各个细节问题.

申请和释放函数地址的获取. 存在于用户地址空间的函数例如 malloc 和 free 函数，通过调试程序(如 GDB)即可获取其地址. 将应用程序的可执行代码进行反汇编，也很容易察看到这些函数的地址. 如在 2.3.2 小节中，截获 malloc 的调用，只需要在地址 0x804830c 处插入 VMCALL 代码即可. 内核函数如 vmalloc 和 vfree 等的地址，可在 GNU/Linux 的 sysmap 中获取.

通过 VMCALL 可以将资源申请和释放的函数地址传递到虚拟机管理器中. 这些地址是有限的、静态的. 而对这些函数的调用点是在虚拟机运行过程中动态发现的，我们需要截获的调用返回点因此也是动态的. 因为对这些地址(申请和释放资源的函数地址、函数调用返回点)的查找操作非常频繁，所以我们将这些地址以及陷入点类型等信息存放在快速查找树中，以提高查找效率.

对资源函数调用和返回的截获. 在函数的首地址和返回处插入 VMCALL 指令之后，虚拟机会在这些地址陷入到虚拟机管理器中. 陷入原因是 VMCALL. 我们在处理 VMCALL 时，判断陷入地址是否在快速查找树中，如果在，则查明陷入点的类型. 有 3 种类型：资源申请函数陷入、资源释放函数陷入和返回点陷入. 如果陷入点是资源申请函数，则需要分析并记录其参数，另外在返回点处插入 VMCALL. 如果陷入点是返回点，则需要分析函数返回结果. 函数的返回结果一般存在堆栈上或者寄存器中，不同类型的函数是不一样的，需要区分对待. 如果申请成功，需要记录动态内存的信息：首地址、大小、时间和类型等. 如果陷入点的类型是资源释放函数，则需要分析其参数，将该动态内存的分配记录删除.

完成截获后的分析和记录，调用 KVM 的 emulate_instruction 函数，模拟执行指令，跳过被插入的 VMCALL 指令. 这里的额外工作是，emulate_instruction 函数会读取虚拟机的指令进行模拟，而当前指令是被替换过的 VMCALL 指令，需要给该函数提供被替换前的指令. 解决方法是修改虚拟机管理器读取虚拟机指令的函数，对不同情况进行判断，如果读取指令地址是被我们替换过的地址，则需要返回原始指令. 然后把状态从 VMM 切换到 VM，截获工作至此完成.

函数调用栈的获取. 当一个函数被调用时，返回地址被压到栈上，然后处理器跳转到被调用的函数处开始执行. 当被调用的函数返回时，处理器从原来被压到栈上的返回地址处继续执行. 当程序执行的过程中出现多层调用时，栈上就会保存一系列对应的返回地址. 由这些返回地址很容易获取到其所在的一系列函数. 这一系列函数被称作调用栈. 资源分配时，分配函数的调用栈能够辅助程序开发人员分析此函数调用的代码轨迹. 对于疑似被泄露的内存，通过其分配函数的调用栈可以获悉其分配时程序的上下文，从而排查出其是否真的存在内存泄露.

获取函数调用栈的难点在于如何从栈上排查出这些返回地址，因为 Intel X86 体系结构中，栈上不仅保存着返回地址，还保存着函数的参数. 准确地获取函数调用栈需要结合分析应用程序的代码和堆栈上的数据，这种方法比较耗时. 本文采用的是应用在 GNU/Linux 内核中的方法. 该方法逐条读取堆栈中的数据，如果该数据落在代码地址空间中，则该数据被认为是一个返回地址. 在实际的运行过程中，该方法不仅快，而且正确率高.

在 KVM 的影子页表机制中实现对内存的监控. 实现过程需要两个步骤：(1) 对于欲监控内存区域所跨越的每个页，在影子页表中清除其 Page Table Entry(PTE)，使得以后对这些页面的访问会产生 Page Fault. 因为我们修改了 PTE，改变了 GVA 到 HPA 的映射关系，所以还需要清除 TLB，使原来的映射失效. (2) 在虚拟机发生 Page Fault 时，判断被访问地址是否在我们保护的范围内，甄别出需要额外处理的部分. 对于被监控区域的访问，记录被访问时间和类型(读或写)，解除对此内存区域的监控. KVM 默认的处理程序会为这个页面建立新的 PTE，至此监控解除.

监控策略的实现. 我们将监控策略以进程的方式运行，策略进程和虚拟机模块通过命令进行数据交互，这种实现方式主要基于如下的考虑. 首先，监控策略的推断是一个复杂且繁琐的过程，不适合实现在内核中，在进程中实现能够更加灵活. 其次，将高层功能从虚拟机管理器中抽取出来，有利于虚拟机管理器的稳定. 监控策略的步骤如下：(1) 策略程序向虚拟机管理器发送命令，获取当前未被释放的动态内存的信息列表，包括内存起始地址、长度、申

请时间、最后访问时间、类型以及函数调用栈等信息. (2)策略进程从列表中推选出需要监控的内存段,发送命令给虚拟机管理器. 最后,策略进程汇总内存段的各种信息,判断其是否存在内存泄露嫌疑.

4 实验评估

前面介绍了基于虚拟机管理器探测内存泄露的实现,本节说明探测内存泄露相关实验的结果. 我们主要关注探测机制对虚拟机带来的性能损失和内存探测有效性两个方面. 针对各部分实验结果,分析了原因,并指出了可能的改进办法. 最终实验结果表明,基于虚拟机管理器进行内存泄露的探测,其实现对应用程序造成的性能损失很小,低于 10%; 在公认的存在内存泄露的开源软件中, 发现了内存泄露的嫌疑. 但因为受测试环境的限制,所发现的内存泄露嫌疑比较有限. 如果能在真实应用环境中进行测试,可能会得到更好的结果.

4.1 实验环境

我们使用的测试环境为,Intel Core™2 CPU@1.86GHz,双核 CPU,2MB Cache,2GB 内存;KVM-84 版本,虚拟机中内核版本为 Linux2.6.24.3,只配备了一个硬盘分区;SATA 硬盘,单网卡;编译器为 GCC4.1.1;虚拟机管理器所在的操作系统版本也是 Linux2.6.24.3. 每次测试时除了受测试虚拟机运行外,无其它虚拟机在执行. 虚拟机只配置了一个 VCPU(Virtual CPU).

首先,我们测试了本实现对虚拟机性能的影响; 然后,测试了公认的存在内存泄露的开源软件中,内存分配和释放的情况,并分析了内存泄露的嫌疑.

4.2 性能损失测试

我们设计了两个实验,测试了本实现本身对运行在 KVM 之上的应用程序性能的影响. 实验 1 测试的程序是常见的编译器 GCC4.1.1,我们截获了 GCC 中的所有的内存申请和释放函数调用,然后分别测试了编译 proftpd-1.3.2rc4 源代码的时间;实验 2 是我们设计的程序,该程序调用了 6 亿次的 malloc 和 free 函数. 最终结果如表 1 所示.

表 1 应用程序在虚拟机中的性能对比

	KVM-84 上运行时间/s	添加了内存泄露探测机制的 KVM-84 上运行时间/s
GCC4.1.1	28	30
6 亿次调用 malloc 和 free	68.889	69.083

由表 1 可知,具有异常频繁内存申请和释放的 GCC4.1.1 程序,在正常的 KVM 虚拟机上编译一个应用程序所需时间为 28s,而在添加了内存泄露探测机制的 KVM 虚拟机上编译同一个应用程序所需时间为 30s,可见内存泄露探测机制引起的性能损失在 10%以内. 而对于我们设计的 6 亿次 malloc 和 free 函数调用的程序,内存泄露探测机制引起的性能损失则更少,几乎可以忽略不计.

4.3 有效性

为了测试基于虚拟机管理器进行内存泄露机制的有效性,我们选择了公认存在内存泄露的两个开源软件:proftpd-1.2.9 和 squid-2.4,并对其分别进行了测试.

测试 proftpd-1.2.9 的实验环境是:通过多个客户端程序,向服务器发送 1000 次 SIZE 命令. 因为 SIZE 命令会导致 proftpd 发生内存泄露. 在测试程序完成后,尚未被 proftpd 释放的动态内存详细信息见表 2.

表 2 proftpd 中未释放内存片段的详细信息

申请内存的地址	调用函数类型	调用次数	申请内存的长度/Byte
0x804e559	malloc()	29	不统一
0x804e778	malloc()	1	8
0x8051239	malloc()	35	绝大部分是 524
0x80584f8	realloc()	1	4097

从表 2 中我们可以看出,在地址 0x804e778 和 0x80584f8 处只有一次没有释放的 malloc 内存分配,分配的内存长度分别为 8 个字节和 4097 个字节;而在地址 0x804e559 和 0x8051239 处没有释放的 malloc 分配记录则有数十次,并且在地址 0x8051239 处分配的内存长度绝大部分都是相同的,因此我们可以推测此处很有可能是一个内存泄露点.

测试 squid-2.4 的方法是同时启动 5 个客户端程序,每个程序向 squid 服务连续发送 100 个请求. 在 5 个客户端程序都结束时,squid 服务未释放的动态内存资源见表 3.

表 3 squid 中未释放内存片段的详细信息

申请内存的地址	调用函数类型	调用次数	申请内存的长度/Byte
0x80a8e4a	calloc()	113	不统一
0x80a8f88	malloc()	14	64
0x80a8edf	realloc()	2	192

从表 3 中我们可以看出,在 IP 地址 0x80a8e4a

处有上百次的 calloc 分配都没有释放，它很有可能是一个内存泄露点；在地址 0x80a8f88 处有 14 次 malloc 内存分配没有释放，且分配的内存长度都是 64 字节，它也可能是一个内存泄露点；而在地址 0x80a8edf 处只有两次 realloc 内存分配没有释放.

如表 2 和表 3 所示，存在内存泄露的可能地址是非常有限的，通过函数调用栈，调试人员很容易就能够定位存在内存泄露的点. 正如上面所说，proftpd 测试中，SIZE 命令必然会导致内存泄露，所以在表 2 中的地址中，必然存在内存泄露.

5　结论以及下一步工作

虚拟机技术的不断发展使得虚拟化的代价越来越低，同时也推动了虚拟化技术的应用. 利用虚拟化技术来提高操作系统的安全性近来受到人们越来越多的重视. 基于虚拟化平台已经有了一系列的研究. 其中一项是利用虚拟机的可复制性建立 HoneyFarm[9]，以捕获、记录和分析黑客对系统的攻击行为，从而提高系统的安全性. 另外，Lycosid[10] 利用虚拟机管理器对资源的绝对控制和对操作系统的透明性，提出了一套检测和标识隐藏进程的机制；有很多病毒程序能够在操作系统中隐藏自己，增加被用户发现的难度. 本文利用虚拟机管理器对其上虚拟机的掌控性和透明性，提出了探测内存泄露的一种机制，该机制提供了内存探测的平台通用性，不需要修改或者重新编译应用程序的源代码，并且，虚拟机的性能损失在 10%以内.

本文的基本思路是：在虚拟机平台中通过在虚拟机(Guest OS)中插入指令的方式，透明地拦截应用程序申请和释放内存资源的函数调用，例如 malloc、free 等动态内存申请和释放的函数，或者 Linux 内核的 vmalloc 和 vfree 函数，维护应用程序所使用动态内存资源的列表. 因为被泄露的内存不会被应用程序所访问，所以接下来需要从这个列表中推选出一部分，进行访问的监控. 那些长时间不被释放且不被访问的内存资源，则是很好的内存泄露的嫌疑. 为了减少监控内存访问的性能损失，本文做了一些重要的优化，例如放弃对热页面的监控.

本研究发现，利用虚拟化技术探测内存泄露具有可行性. 首先是在线性，根据实验，利用虚拟机管理器探测内存泄露给虚拟机带来的性能损失小于 10%. 测试程序的逻辑是有限的，有些内存泄露只有在真正应用时才能发现. 其次是有效性，通过实验，在公认的存在内存泄露的开源软件中，确实能够发现内存泄露的嫌疑. 另外的优势是通用性，该方法不仅适用于 GNU/Linux 操作系统，而且对 Windows 也适用，不仅适用于用户程序，对操作系统内核也同样适用. 最后是实现的简单性，该方法不需要修改现有的操作系统，只需要对支撑虚拟机运行的虚拟机管理器进行修改.

我们下一步工作是，在虚拟机之内确认内存泄露之后，通过动态代码修补的技术，动态修正运行在虚拟机中的应用程序，以保证应用程序所提供的服务不会中止.

参　考　文　献

[1] Xie Yichen, Aiken Alex. Context- and path-sensitive memory leak detection//Proceedings of the ESEC/FSE'05. Lisbon, Portugal, 2005: 115-125

[2] Tsai T, Vaidyanathan K, Gross K. Low-overhead run-time memory leak detection and recovery//Proceedings of the PRDC'06. Riverside, USA, 2006: 329-340

[3] Qin Feng, Lu Shan, Zhou Yuanyuan. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs//Proceedings of the HPCA'05. San Francisco, USA, 2005: 291-302

[4] Bungale Prashanth P, Luk Chi-Keung. PinOS: A programmable framework for whole-system dynamic instrumentation//Proceedings of the ACM Conference on Virtual Execution Environments(VEE'07). San Diego, California, USA, 2007: 137-147

[5] Waldspurger Carl A. Memory resource management in VMware ESX server//Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02). Boston, USA, 2002: 181-194

[6] Barham P et al. Xen and the Art of virtualization//Proceedings of the 19th ACM Symposium on Operating System Principles(SOSP'03). New York, USA, 2003: 164-177

[7] Neiger G et al. Intel virtualization technology: Hardware support for efficient processor virtualization. Intel Technology Journal, 2006, 10(3): 167-177

[8] AMD. AMD-V™ Nested Paging. Revision: 1.0. Issue Date: July, 2008

[9] Michael Vrable et al. Scalability, fidelity, and containment in the potemkin virtual honeyfarm//Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP'05). Brighton, United Kingdom, 2005: 148-162

[10] Jones S T, Arpaci-Dusseau A C, Arpaci-Dusseau R H. VMM-based hidden process detection and identification using Lycosid//Proceedings of the ACM Conference on Virtual Execution Environments (VEE'08). Seattle, USA, 2008: 91-100

WANG Xiao-Lin, born in 1972, Ph. D., associate professor. His research interests include system virtualization, GIS.

WANG Zhen-Lin, born in 1970, Ph. D., associate professor. His research interests include architecture, compile, system virtualization.

SUN Yi-Feng, born in 1983, Ph. D. candidate. His research interests focus on system virtualization.

LIU Yi, born in 1985, M. S. candidate. His research interests focus on system virtualization.

ZHANG Bin-Bin, born in 1982, Ph. D. candidate. Her research interests focus on system virtualization.

LUO Ying-Wei, born in 1971, Ph. D., professor. His research interests include system virtualization, GIS.

Background

Memory leaks are a common problem in code written in languages with explicit memory management, and are known to be a major cause of reliability and performance issues in software. Many researchers have worked on the problem of detecting memory leaks, but memory leaks remain to be the hardest bugs to detect.

This paper describes a new schema that can detect memory leaks in production systems. Virtualization technology is utilized to transparently record the allocation and release of memory resources applied by applications running on virtual machine, and these records provide the auxiliary information to detect memory leaks hiding in the binary code.

Research areas of the authors include live migration of virtual machine, online cloning of virtual machine, remote memory sharing of virtual machine, dynamic para-virtualization, and security issues in virtual machine. Those works are mainly supported by the National Grand Fundamental Research 973 Program of China titled "the Research on Fundamental Theory and Approach of Computing System Virtualization (grant No. 2007CB310900)".

The subject of this paper is on the domain of virtual machine security.