doi:10.3969/j.issn.1000-436x.2013.z1.004

基于虚拟机自省的运行时内存泄漏检测模型

肖如良，姜军，倪友聪，杜欣，谢国庆，蔡声镇
（福建师范大学 软件学院，福建 福州 350108）

摘 要：云计算及数据中心领域中已广泛采用虚拟化技术来尽可能消除虚拟计算环境中的内存泄漏是提高其可靠性的一种重要途径。提出了一种基于虚拟机自省机制的运行时内存泄漏的信息流检测模型与内存泄漏的判定方法，设计并实现了该模型的原型系统。通过对原型系统的有效性与性能评估实验分析，结果表明，该模型方法能有效地检测出运行时内存泄漏，并且具有较好的性能。

关键词：虚拟机；自省；内存泄漏；有效性；性能
中图分类号：TP302 **文献标识码**：A **文章编号**：1000-436X(2013)Z1-0021-10

Model of runtime memory leak detection based on the virtual machine introspection

XIAO Ru-liang, JIANG Jun, NI You-cong, DU Xin, XIE Guo-qing, CAI Sheng-zhen
(Faculty of Software, Fujian Normal University, Fuzhou 350108, China)

Abstract: Virtualization technology has been widely used in the field of cloud computing and data center, and it is an important way to improve reliability of system under the virtual computing environment using runtime memory leak detecting to eliminate memory leaks. A model of runtime memory leak detection based on the virtual machine introspection and an approach to predication on memory leak were proposed. The prototype system of the model was designed and implemented. By analyzing and evaluating effectiveness and performance of the prototype system, the results show that these models and methods can effectively detect memory leaks, and have better performance.

Key words: virtual machine; introspection; memory leak; effectiveness; performance

1 引言

近年来，硬件辅助虚拟化技术(virtualization)的功能日渐完美与成熟，使 X86 架构在服务器领域有了广泛的应用。虚拟化技术为各种应用提供了高性能和高可靠的廉价服务器，也为云计算、数据中心提供了可行的技术支持，比如 VMware 公司、Citrix 公司的各种云计算及数据中心的解决方案。基于计算机系统结构的各个层次，各种主流虚拟化方案中的虚拟化技术相应地可分层实现，依次可分为硬件级虚拟化、指令级虚拟化、操作系统级虚拟化、编程语言级虚拟化、程序库级虚拟化。从系统结构来看，虚拟机管理器 VMM 是整个虚拟系统的核心，它承担了资源的调度、分配和管理，在保证多个虚拟机能够相互隔离的同时运行多个客户机操作系统 Guest OS。内存虚拟化是 VMM 的重要功能之一，它对内存实现三级内存管理：机器地址、伪物理地址、虚拟地址，伪物理地址是经过 VMM 抽象的虚拟机 VM 所能看到的地址，虚拟地址是 Guest OS 提供给其他应用程序使用的线性地址空间。由于内存是虚拟机最频繁访问的设备之一，给 VMM 的内存管理带来了严重的挑战，特别是云计算、数据中心领域中长时间不停机应用系统的内存泄漏，可能使虚拟机系统发生崩溃，以致造成计算机安全事故。

内存泄漏是指被申请的内存资源没有被合理

收稿日期：2013-06-22
基金项目：福建省科技计划重大基金资助项目（2011H6006）
Foundation Item: Science and Technology Program Key Project of Fujian Province (2011H6006)

地释放而在某一时刻以后不能再被使用和释放的现象。对于虚拟化系统的三级内存机制，如果泄漏的是虚拟内存，将导致虚拟机 VM 内存空间减少，促使 VMM 的气球驱动机制不断地调整内存空间；如果泄漏的是伪物理地址，会使得 VM 的 Guest OS 性能大幅下降而促使 VMM 启动 Swap 机制，系统性能会显著降低；如果泄漏的是机器地址空间，会给整个虚拟机系统带来严重的后果。通常，这 3 个层级的内存泄漏不是孤立的，相互之间是关联的。

Purify、SafeMen 主要针对虚拟机 VM 内应用程序所在的虚拟内存层面的内存泄漏进行检测，Purify 需要编译插装指令以捕捉应用程序的内存申请与释放行为，制定适当的规则来判定内存泄漏行为[1]。而 SaftMen 则是需要特殊的 ECC 内存硬件支持，添加新的系统调用，在一定程度上需要一定的额外资源与性能损失才能支持 KVM 虚拟化环境下的内存泄漏检测[2]。这二个典型的工作主要是在 VM 内完成内存泄漏的检测工作；PinOS 在虚拟机管理器 VMM 之上通过软件动态翻译的方法对伪物理地址空间的 Guest OS 调试内核代码的机制，在运行时插入代码到 Guest OS 的内核地址空间中[3]；北京大学的汪小林提出了一种利用 KVM 虚拟机管理器 VMM 捕捉虚拟机 VM 内应用程序对内存资源的请求与释放函数[4]，这种方法尽管不需要重新编译应用程序，但需要替换掉部分 VM 指令。

本文的方法主要是利用虚拟机与虚拟机管理器的自省机制，通过建立应用程序运行的 VM 状态(state)自省、VMM 的状态自省，结合内存泄漏的自省判定准则，生成内存泄漏视图 View。

2 虚拟化体系结构中的监控机制与自省机制

虚拟机 VM 是指在一个硬件平台上模拟多个独立的、ISA 结构和实际硬件相同的虚拟硬件系统，在每一个虚拟硬件系统上都可以运行不同的操作系统，即客户操作系统(guest OS)。而虚拟机管理器 VMM 是一个位于计算机硬件和操作系统之间的软件层，负责管理和隔离上层运行的多个虚拟机。运行在 VM 中的 guest OS 通过虚拟机管理器 VMM 访问实际的物理资源。因而整个虚拟机系统是以 VMM 为中心来实施资源的管理与控制。在这个虚拟化体系结构中，位于上层的 VM 与位于下层的 VMM 对整个系统功能的支撑是由一种监控机制(VS, virtualization surveillance)来体现的[5]。武汉大学的王丽娜提出了一种基于 VMM 的隐藏进程检测系统，该系统驻留在被监控虚拟机外部，通过交叉视图的方式检测隐藏进程[6]。

对应于体系结构的上层虚拟机，文献[7]提出的 SIM 系统实现了一种典型的 VS 安全监控机制，在虚拟机内加载内核模块来拦截目标虚拟机的内部事件。基本机制是被监控的系统运行在目标虚拟机中，安全工具部署在一个隔离的虚拟域（安全域）中。文献[2~4]都实现了这种架构，支持在虚拟机内部署钩子函数，拦截某些事件例如进程创建、文件读写等。文献[7]特别实现了一种安全的机制，当这些钩子函数加载到客户操作系统中时，向虚拟机管理器通知其占据的内存空间，由内存保护模块根据钩子函数所在的内存页面对其进行保护。这种框架结构如图 1 所示。

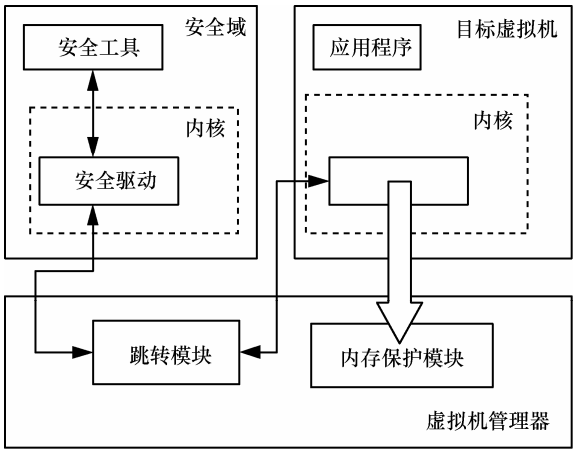

图 1 一种典型的 VS 安全监控框架

对应于体系结构下层的虚拟机管理器实现的监控机制，必须透过 VM，同时把 VM 与 VMM 二者结合起来实现。最典型的是 2002 年由斯坦福大学的 GARFINKEL T 和 ROSENBLUM M 首次在文献[8]中提出的一种能够观察到被监控系统的内部状态、同时与被监控系统隔离的入侵检测架构。该架构利用虚拟机管理器能够直接观察到被监控系统的内部状态，通过直接访问其内存来重构出客户操作系统的内核数据结构来进行检测。这种在虚拟机外部监控虚拟机内部运行状态的方法称为虚拟机自省(VMI, virtual machine introspection)。如图 2 所示，是一个基于 VMI 的入侵检测系统框架设计的高层视图。

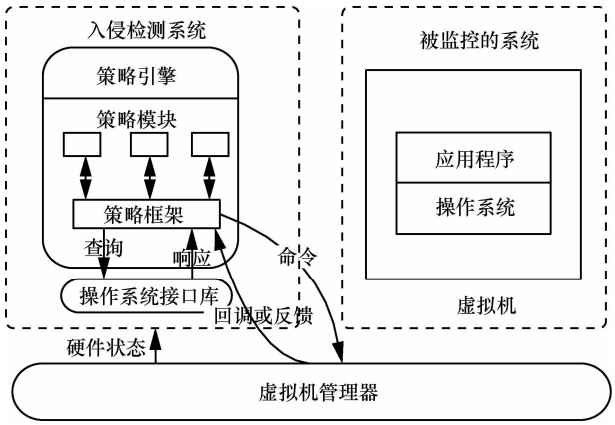

图 2　一个基于 VMI 的入侵检测系统的框架

自省意指自我检查，需要设置一个特权虚拟机来检查与监控其他虚拟机，并通过 VMM 来获取被监控系统所在虚拟机的各种状态信息进行状态比较，因而是一种具有一定自我检查意义的机制。

以上的 2 种监控机制是从基于可信计算、入侵检测等安全意义来考虑的，从效率上看，前者要频繁地拦截操作系统中的行为，效率更低；而后者从 VMM 层面去查看 VM 中的状态信息，建立 OS 状态的映射，效率相对较高。从操作语义层面来看，前者所拦截的 OS 系统调用属于高层操作语义，容易被系统理解所截获的操作语义意图，而后者中含有二层语义，既有 VM 的上层操作语义，也有 VMM 的下层操作语义，要经过语义双向转换映射才能捕捉到系统的操作行为。也就是说，这种由 VMI 机制的实现要解决一个语义鸿沟(semantic gap)的问题。针对语义鸿沟，Garfinkel 在文献[9]中实现了一个入侵检测系统 VMI IDS；2011 年慕尼黑工业大学的 Christian Schneider 提出了一种通用的 VMI 语义解决方案，并实现了一个工具 Insight 实现语义的可转换与理解[10]。这种实现是基于一种视图生成器 VGC 进行的，操作语义所对应的各种状态信息集成在 VGC 组件中。

针对虚拟机系统中应用程序的内存泄漏检测，文献[2~4]实现的是前面的框架。作者在结合虚拟机监控机制，并从这种内省机制上受到启发，构建了一种新的面向虚拟计算环境的内存泄漏检测方法，是以上第二种框架的一种应用。但作者并不需要设置一种具有特权意义的虚拟机，系统中需要的特权操作由下层的 VMM 实现，作者吸取文献[9,10]的思想，也提出了一种语义鸿沟的解决形式，结合视图生成组件的优点，所定义的视图只需在内存泄漏这个意义上来限定，相对入侵检测来说，视图内容的生成、语义层面的转换具有自身的特点。

3　基于自省机制的运行时内存泄漏检测模型

为了进行内存泄漏分析的描述，考虑到从虚拟机操作系统层面进行基于虚拟机内省的内存泄漏检测（MLD/VMI, memory leak detection based on VMI），采用的框架如图 3 所示。

VM *i* 作为内存泄漏监控端虚拟机并不具有特权，VM *j* 作为被监控端虚拟机，它们分别可以设置多个类似的虚拟机；下层的虚拟机管理器 VMM 可部署一个内存泄漏检测服务维护模块(SMM, service maintain module)处理有关底层操作事务，并可通过 VMM 利用虚拟机自省机制将获取被监控端的有关内存操作事务信息。

内存泄漏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重新利用的一种现象。无论是托管语言环境（如 C#、Java 等）还是非托管运行机制(如 C\C++等)下，泄漏行为会导致内存资源耗尽或无页面可分。通过操作系统对内存分配事件、内存使用情况进行跟踪，人们可知道什么时候分配的内存不再使用。堆的相关使用信息主要包括随时间而发生的 allocaton type、age、size、thread ID，一个堆事件(event)伴随了堆的 alloc、realloc、free，这些事件的跟踪可由操作系统来提供，从而保证了系统工作的可靠性与有效性。因此，操作系统所跟踪的事务中所记载的堆事件主要构成的细节可由时间戳(timestamp)、分配类型(allocation type)、地址(address)、大小(size)、进程(process ID)、线程（thread ID）、堆柄(heap handle)、源(source)组成。所有这些信息构成了与应用程序的内存泄漏行为相关的因素。这些因素对于底层 VMM 来说是不能理解的，作者必须建立一个上层与下层的语义转换函数来表达，这个函数在 VMM 层中的 SMM 与上层的 Monitor 中的有关模块建立了一种双向的语义转换来实现。

下面将建立这个基于 MMI 的运行时内存泄漏框架所对应的模型。

3.1　基于自省机制检测的流模型

对于模型下层，下文给出状态(流)、视图(流)、分类等概念的形式定义，为之后建议的语义转换奠定基础。被测虚拟机端的目标对象可以是虚拟机中的客户操作系统 guest OS，也可以是目标应用程序

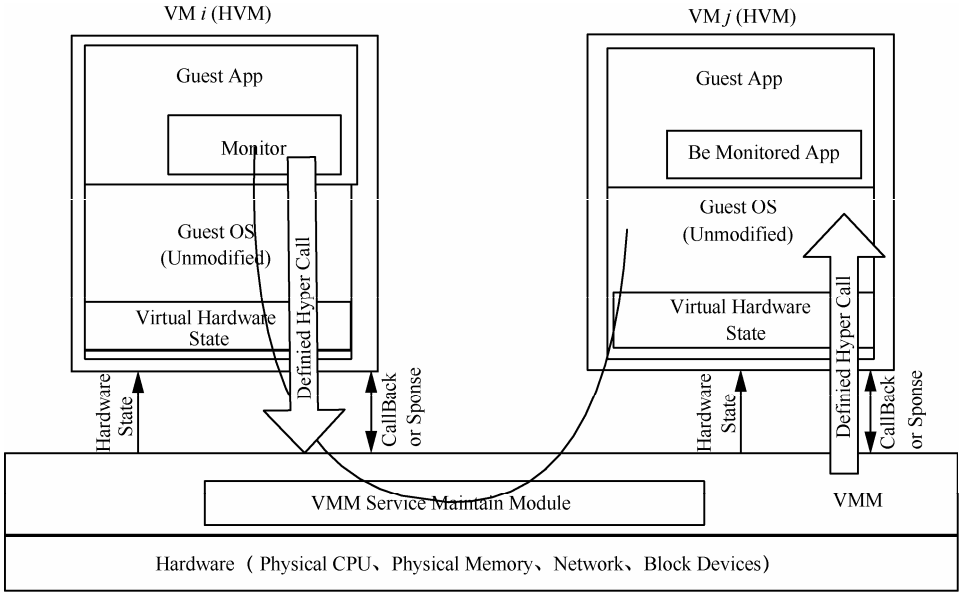

图 3　基于 VMI 的内存泄漏检测框架

本身。目标应用程序的状态对 guest OS 是可见的，整个被测端 VM 对于 VMM 来说，其状态也是可见的，比如 CPU Register、RAM、Connected Devices 等状态信息。

定义 1　基本数据类型，包括 *OS_ident* 作为一种类型表明操作系统的型别，如所在虚拟机标识、操作系统的种类、版本号等。

定义 2　应用系统活动类型，包括等待处理器的 *waiting* 状态与处理器运行执行应用程序模式 *runing*，即定义 *Os_activity*::=*running*|*waiting*；执行模式类型 *Execution_mode*, 表明所在虚拟机的特权状态，可选的状态有用户态(*User mode*, *usr*)与特权态(*Supervisor*, *svc*)，由特权指令来进行状态的转换，*execution_mode*::= *usr* | *svc*；*VM_ident* 作为一种类型表明虚拟机类别，如 Xen 虚拟化技术下的 *VM_ident*::={*Dom*0 | *Domu*}；

系统调用把应用程序的请求传给内核，调用相应的的内核函数完成所需的处理，将处理结果返回给应用程序，在本文的工作中，作者重点关注的系统调用是内存资源的申请与释放，如 *malloc*、*alloc*、*realloc*、*free* 等，*Systemcall*::={*malloc*| *alloc*|*realloc*| *free*}

超级调用 *Hypercall* 是指在特权状态下为完成特定任务而实现的操作行为，如监控并捕捉系统调用的操作等。在本文的实现中将定义 6 个超级调用 *Hypercall*。

于是系统操作 *Os_action* 可以定义为：*Os_action*::=*Systemcall*|*Hypercall*。

定义 3　内存映射是指从虚拟内存地址到伪物理地址与从伪物理地址到机器地址 2 个方面的映射。分别是：*OS_map*::={*Vadd*→*Padd*} 和 *VM_map*::={*Padd*→*Madd*}，前者是在一个虚拟机操作系统内发生，后者是在虚拟机管理器内发生。

定义 4　内存对象(memory object)。经系统调用如 *malloc*、*realloc*、*calloc* 等分配的内存块，是一种内存映射 *OS_map*∧ *VM_map* 的结果。活内存对象(live memory object)：没有被释放的内存对象，是可以进行读写 RW(*v*:*Option Value*)、释放等相关操作的内存对象。因而∀*mo*∈*MemoryObject*, ◊*mo*∈*LiveMemoryObject*。

内存对象生命期(lifetime of memory object)：从内存对象分配内存到被释放的这段时间。它的取值为在一个数学意义上的区间[,∞)内，∀*t*∈*Lifetimeof*−*MemoryObject*, *t*∈[0,∞)。

在一个应用程序中，内存对象的最大生命期在达到一个预定的阈值时，往往表达了活内存对象的存活能力，可以用来对内存泄漏进行表征的一个重要参考点。

内存对象组（MOG, memory object group）：一种具有某种共性的内存对象的集合，本文主要从内存对象生命期的意义来对内存对象进行分组，同一组中的内存对象具有同类意义的生命周期。

综合起来，一个 guest OS 内的页面内容 *Page_*

content 要么是可以读写的数据，要么是页表 *PT* 所标示的内存映射形式，或不可读写的未初始化内存，可以进一步定义如下：

Page_content ::=*RW*(*v*:*Option Value*) |*PT(va_to_ma*: *vadd*→*madd)*|*MOG*

Page_Owner::=*VMM*|*OS*|*No_Other*

Page::={ *Page_content* , *Page_Ower*}

定义 5　运行时内存泄漏(RML, runtime memory leak)：指应用程序在虚拟机 HVM 运行过程中由客户机操作系统所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应用程序或操作系统重新利用的一种现象；平凡内存泄漏(TML, trivial memory leak)：指发生泄漏的次数有限，仅造成整个计算机系统内存的浪费或 HVM 分页时造成执行速度变慢，对软件的可靠性与可用性的影响较少；连续的内存泄漏(CML, continuous memory leak)：指发生泄漏的点能引起程序用完系统的虚拟内存，最终造成系统崩溃。

ML::=*TML*|*CML*

∀*ml*∈ *ML*,*ml*∈ *TML*∨*CML*→(*ml*∈ *TML*)∨(*ml*∈ *CML*)

∀*mog*∈ *MOG*, *mog*∈ *ML*∨¬*ML*

对于长时间运行的服务器程序来说，崩溃是灾难性的，特别是对于 Web 服务来说，由于不能提供服务而造成业务的损失。

从这里不难得出以下结论。

定理 1　如果应用程序存在内存泄露的话，那么发生运行时内存泄漏的点就隐藏在人们维护的内存对象组中。

定义 6　内部状态 $S\mathrm{B}_{int\mathrm{B}}$ 是所有对虚拟机客户操作系统可见的 VM 内存状态的集合、内部状态 $S\mathrm{B}_{int\mathrm{B}}$ 包括了当前虚拟机的 *OS_ident* 信息、伪物理地址的当前内存页表等。

$S\mathrm{B}_{int\mathrm{B}}$::={*os*:*OS_ident*, *execution_mode*, *om*:*OS_map*, *Page_content*, *Page_Owner* }

外部状态 $S\mathrm{B}_{ext\mathrm{B}}$ 是所有对 VMM 可见的 VM 内存状态的集合，外部状态 $S\mathrm{B}_{ext\mathrm{B}}$ 包括了当前虚拟机的 *VM_ident* 信息，伪物理地址到机器地址的映射等。

$S\mathrm{B}_{ext\mathrm{B}}$::={ *VM_ident*, *VM_map* }

以上为二进制字符串的长度 $S\mathrm{B}_{ext\mathrm{B}} \subseteq \{0, 1\}\mathrm{P}^{m\mathrm{P}}$。一个虚拟机状态包含了该虚拟机的内部状态与外部状态，于是可定义为：*State*::= $S\mathrm{B}_{int\mathrm{B}} \cup S\mathrm{B}_{ext\mathrm{B}}$。*State* 是所有可见的值的集合，对于内存泄漏检测而言，通常分为 3 类：泄漏(leak)与不泄漏(unleak)2 种状态，但有一些具有泄漏的特征而并没有能确定是泄漏或不是泄漏的中间状态，称为泄漏嫌疑(suspected_leak)因此，对于一个具体的场景，可以进一步定义虚拟机状态的分类如下。

Classify::={ *leak* | *unleak* | *suspected_leak* }

从而，状态的分类可以在一个分类映射下进行，映射如下：

f: *State*→*Classify*

这个映射 *f* 也是反映在内存泄漏的判定规则层面，在下文中进行定义。

通过状态的分类可以明确虚拟机系统内部与外部的状态泄漏类型，这种分类对应于某一个时间点状态而言，处于后台或者说属于系统内部的内存泄漏判定操作。但对于反映状态信息来说，如果能像数据库查询语言一样构建一种视图信息，将视点定位在用户查询的角度，并在一段时间内进行反映状态的分类，可以更好地反映出监控虚拟机的泄漏或未泄漏的结果。

定义 7　视图 View 可以具体地表示出虚拟机状态 *s*∈ *State* 的一系列可视化信息，如果不泄漏时，则为空集；如果泄漏，则为泄漏的相关信息。视图与状态相对应地定义出 2 种类型：内部视图与外部视图。内部视图 $V\mathrm{B}_{int\mathrm{B}}$ 从内部状态 $S\mathrm{B}_{int\mathrm{B}}$ 投影生成，外部视图 $V\mathrm{B}_{ext\mathrm{B}}$ 从外部状态 $S\mathrm{B}_{ext\mathrm{B}}$ 投影生成，*View*= $V\mathrm{B}_{int\mathrm{B}} \cup V\mathrm{B}_{ext\mathrm{B}}$。内部视图生成与外部视图生成分别定义如下：

λ: $S\mathrm{B}_{int\mathrm{B}} \rightarrow V\mathrm{B}_{int\mathrm{B}}$

μ: $S\mathrm{B}_{ext\mathrm{B}} \rightarrow V\mathrm{B}_{ext\mathrm{B}}$

从而视图的分类可以在一个分类映射下进行，映射为

g: *View*→*Classify*

分类规则 *g* 由内存泄漏的判定规则来实施。

在以上的状态、视图的定义都是对应于一个时间点而言。如果是多个观测时间点，对于状态的分类而言，可以做各状态的并集即可，∀$s\mathrm{B}_{1\mathrm{B}}$,$s\mathrm{B}_{2\mathrm{B}}$∈*State*，$s\mathrm{B}_{1\mathrm{B}} \cup s\mathrm{B}_{2\mathrm{B}}$∈*State*，对于视图的分类也是如是：∀$v\mathrm{B}_{1\mathrm{B}}$, $v\mathrm{B}_{2\mathrm{B}}$∈*View*，$s\mathrm{B}_{1\mathrm{B}} \cup s\mathrm{B}_{2\mathrm{B}}$∈*View*。但对应于流而言，由于时间视点不同，作者给出了相应的流的定义。

定义 8　状态流 *Stateflow* 是指一段时间视点所对应的多个状态的笛卡尔积，*Stateflow*::=*State*×*Stateflow*，视图流 *Viewflow*::= *View*×*Viewflow*，这个定义是一种迭代定义方式，统称为笛卡尔(Descartes)映射算子转换。

以上的相关信息流向处理过程如图 4 所示。

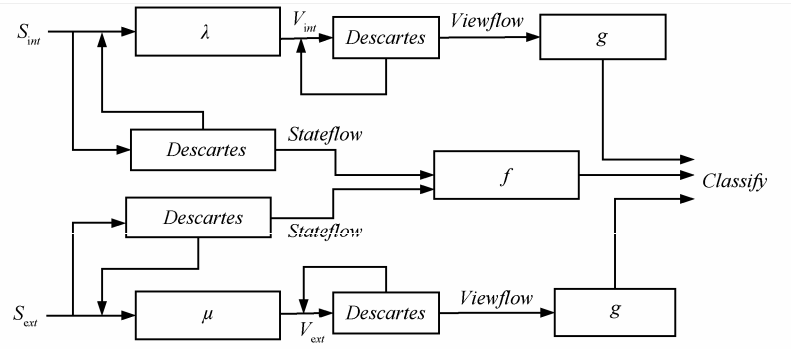

图 4　状态流与视图流的内存泄漏识别 MLR 模型

图 4 所示的信息流图表达了基于虚拟机自省 VMI 的一种内存泄漏检测模型，其中，状态的转换是由于内存操作如内存资源的申请与释放（*malloc*、*alloc*、*realloc*、*free*）动作而产生的，相应地视图也会发生转换，因而随着时间视点的变化产生了可观测的状态流、视图流。下面先对内存泄漏判定相关的概念进行定义，然后再给出图 4 模型中的判定状态流和视图流内存泄漏的规则，以构建判定算子（函数）*f*、*g* 等。

3.2　内存泄漏判定

由于各个 VM 中的 guest OS 可能各不相同，为了能实现跨平台检测内存泄漏，忽略掉各 VM 中的异构因素，由于内存泄漏的捕捉可以分别在各个层级内来实现，根据前面所定的框架，作者将内存泄漏发生的表现层次定位在 HVM 这种虚拟机的伪物理内存层。

作者更主要关注的是连续内存泄漏 CML 形式的检测。CML 有 2 种主要的内存泄漏类型如下，不同的类型有不同的特征。第 1 种 CML 类型表现为该组内存对象总是快速增长，而每一个内存对象都有不受限制的生命期。

定义 9　内存对象总是泄漏(ALeak, memory object always leak)：在应用程序所有可能的执行路径上都不会释放的一组对象。

ALeak 这类内存对象的特征明显，检测相对容易。从 ALeak 的定义容易得出内存泄漏 ALeak 判定算子定理如下。

定理 2　$\forall v\in Viewflow$，$(\exists s\in State\wedge \exists o\alpha=running\in Os\_activity)\wedge (ALeak(s)=\text{true})\rightarrow s\subseteq ML$，同样地，$\forall s\in Stateflow$，$(\exists s\in State\wedge \exists o\alpha=running\in Os\_activity)\wedge (ALeak(s)=\text{true})\rightarrow s\subseteq ML$。

应用程序运行时如果有一个内存对象组总是泄漏，则 ALeak 一定会内存泄漏。

第 2 种 CML 类型表现为在同一内存对象组中有些具有有限的生命期，而其他对象则具有无限的生命期。这个类型的对象泄漏内存的速度较慢，但经过一段时间的运行后，仍会耗尽内存资源导致程序崩溃。这种内存泄漏更难检测，因为只是有些执行路径上发生泄漏。

定义 10　内存对象有时泄漏(SLeak, memory object sometimes leak)：在有些执行路径上程序会释放已分配内存的对象，但在其他路径上程序并不会释放分配的内存对象。

结合定理 1 与定理 2，可以得出定理 3。

定理 3　$\forall v\in Viewflow,(v\in\subseteq ML)\rightarrow(\exists mog\in MOG)\wedge (mog\in ML)$；同样，$\forall s\in Stateflow, (s\subseteq ML)\rightarrow(\exists mog\in MOG)\wedge(mog\subseteq s)\wedge(mog\in ML)$。即如果当前状态流或视图流中有内存泄漏发生，则一定存在一个内存对象组发生内存泄漏。

定理 2 与定理 3 即构成了内存泄漏的判定。

4　原型实现与实验分析

本文将实现这个内存泄漏信息流检测形式化模型，给出一种案例实现思想，之后通过实现得出的实验数据进行分析。

4.1　实现思想

前面的内存泄漏信息流检测模型是基于自省机制来实现的。这种自省机制统计应用程序所申请的资源使用情况，以准确推测是否有内存泄漏发生。在程序运行过程中必须持续不断地检测泄漏数据，主要包括 3 个基本过程：首先，动态分析受监控程序的内存使用行为，依据所观测到的内存对象生命期的稳定性对其行为进行内存对象分组；其次，基于观测到的行为特征检测潜在的内存泄漏；最后，依据相关规则进行内存泄漏判定。这 3 个步骤所占用的系统资源极少。下面分别介绍这 3 个基本过程。

4.1.1 观测内存使用行为特征，实现内存对象分组

对每一个内存对象组，对其内存分配与释放的行为实施收集，必须记录其相关的信息如：生命期信息、内存使用信息。前者对当前最大生命期(maxLifeTime)与最大生命期的稳定时间(stableTime)进行记载，而所有的时间都是以 CPU 时间来度量的；后者包含了当前活的对象、最后分配时间、内存对象组占用的总的内存空间大小。对于每一个活内存对象，也必须记录其内存分配的时间，同一个内存对象组内所有活内存对象都以双向链表的形式来表示，以便快速查找。

对于每一次内存分配，也就是说一个新的活内存对象添加到这个双向链表以后，内存对象组的关联信息必须进行更新，如对象个数、最后分配时间、占用的总空间大小等。

同样，对于每一次内存释放，相关的组对象信息(如当前的活对象数、组占用的总空间数)也要相应地更新，同时还有一个重要的生命期数据(lifeTime)而进行计算求得，通过其分配时间与当前时间之差可得，如果生命期数据在一个可容易的范围内(预设的一个阈值)，少于该内存对象组相关的最大生命期(maxLifeTime)，则相关的 *maxLifeTime* 也并没有改变，而且 *stableTime* 随着它的 CPU 处理时间而增加，否则，相关的 *maxLifeTime* 必须更新，而且 *stableTime* 重置为 0。

在这个过程中，涉及到 *malloc*()、*calloc*()、*realloc*()、*free*()等内存调用函数，作者统一称之为内存分配(allocation)与释放(deallocation)，或为了方便仅以 *malloc*()、*free*()表达。

4.1.2 检测潜在的内存泄漏

如果程序运行过程中并没有做内存申请与释放操作，那么内存泄漏的检测过程就不需执行。只有在程序经过一个 *warmUpTime* 时间以后，在出现内存分配(allocation)与释放(deallocation)操作时才触发这个检测过程。针对 2 种不同的泄漏类型，采用的过程如下。

1) 检测 ALeak 的过程。每一个组中活内存对象的数量是否超过了预设的阈值。

①如果是超过了，则进一步检查该组使用的内存是不是连续地增长；可通过检查最后的分配时间来确定，如果有一段较长时间（最后分配时间减去当前时间）都没有内存使用增长，则不会发生内存泄漏。

②如果没有超过，也可能是在初始时程序分配了许多内存对象，在整个执行过程中这些对象都在使用。

③如果最后内存分配时间离当前时间很近，内存使用仍然在增长，由此这组内存对象是泄漏的嫌疑对象。

2) 检测 SLeak 的过程。

①检测每一个组中活内存对象的生命期(lifeTime)，一个对象被看作泄漏嫌疑时，它具有以下 4 个特征。

a) 该对象的活动时间超过了 2 倍 *maxLifeTime*。

b) 该对象所在的内存对象组的最大生命期 *maxLifeTime* 在一个预设阈值时间还长的时间内相对稳定。

c) 该对象在合法的 *maxLifeTime* 内动态内存部分不释放，但持续申请，导致所占用的内存越来越多。

d) 该对象在合法的 *maxLifeTime* 内已释放，但多次重复释放导致不可预计的行为发生。

②如果第 2 个特征不是真的，则证据不足。

以上 2 种情况的检测都只是在程序有内存申请与释放行为时才触发这个检测过程，而且这个检测只是在一个预设的检测周期内进行，因而所用的内存开销不大。

4.1.3 内存泄漏的确认规则

毫无疑问，Aleak 一定会内存泄漏。而对于 Sleak 而言有可能只是泄漏嫌疑。内存泄漏的判定，作者在后续的 VMLD 实现方法中构建了一种判定规则。

规则 1 对于在检测过程中已经标记为泄漏嫌疑的对象，在预定的时间阈值内，如果这个内存对象被再次访问，则该内存对象只是可能泄漏(PLeak, probably leak)，否则该内存对象确定为泄漏。

规则 2 如果这个嫌疑是 *PLeak*，则重置这个内存对象的分配时间为当前时间，以更好地确认这个嫌疑对象。如果一个对象能重新变为嫌疑泄漏对象，则确认为内存泄漏对象。

以上的确认过程并没有占用大的额外开销，仅仅是对少数嫌疑的泄漏对象进行确认，因此程序的运行性能并不会受到大的影响。

4.2 原型系统框架

本文的实现是在虚拟机与虚拟机管理器内检测并确认内存泄漏。该方法不同于文献[4]的方法，也不同于 Purify[1]、SafeMem[2]等现有的方法。对

支撑虚拟机运行的虚拟机管理器进行修改，基于自省机制通过编制虚拟机管理器环境下的超级调用来实现对内存泄漏对象的检测、标识可能的泄漏嫌疑对象，并对进行虚拟管理器环境下的泄漏进行确认。

本文拟在 Xen 虚拟机管理器之上实现内存泄漏的检测与确认。Xen 源自英国剑桥大学计算机实验室主持的一个开源虚拟化项目[注1]。同其他的硬件虚拟化技术一样，在 Xen 系统中存在一个轻量级的软件层，向运行在它之上的虚拟机提供虚拟硬件资源，同时分配和管理这些资源，并保证虚拟机之间的相互隔离。这个轻量级的软件管理层称为虚拟机管理器(VMM)。在 Xen 系统中，VMM 又称为管理程序(Xen hypervisor)，或简称为 Xen，而虚拟机则被称为虚拟域(Domain)，每一个 Domain 用来安装客户操作系统(guest OS)，由 Xen 控制，以高效地利用 CPU 的物理资源。每个客户操作系统可以管理自身的应用。这种管理包括每个程序在规定时间内的响应到执行，是通过 Xen 调度到虚拟机中实现。在众多的 Domain 中存在一个特权域用来辅助管理其他虚拟域，这个特权域称为 Domain 0，是其他虚拟主机的管理者和控制者，Domain 0 (Dom 0)，而其他域称为 Domain *U* (Dom *U*)。Xen 向 Domain 提供了管理和虚拟硬件的 API 抽象层，使 Dom 0 拥有真实的设备驱动区(原生设备驱动，native device driver)，能够直接访问物理硬件，它负责与 Xen 提供的管理工作 API 交互，可以管理和控制其他 Domain。本文所提出的虚拟化内存泄漏检测框架(VMLD, virtualization memory leak detecting framework)实现由 4 个大的功能模块组成的，如图 5 所示。

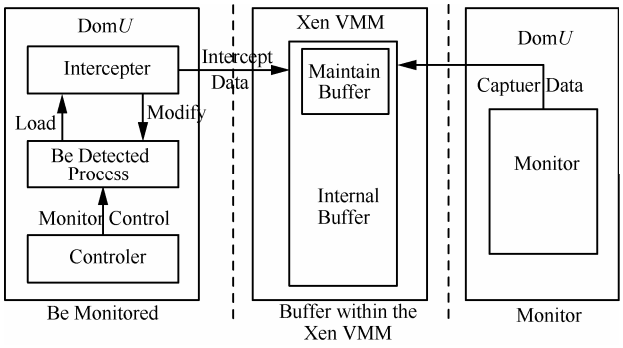

图 5　基于 Xen 平台的 VMLD 体系结构

注1：http://xen.org/.

它们分别是 Xen 虚拟机管理器缓冲(buffer within Xen VMM)维护模块(maintain buffer)、控制模块(controller)、拦截模块（interceptor）和监视模块(monitor)。在被监控端，被测进程(be detected process)是未经修改的正在运行的一个进程。控制器(controller)是用来控制被测进程，使被测进程主动加载拦截器和向被加载的拦截器(intercept)发送指令，使拦截器对被测进程进行修改，拦截内存的申请和释放行为。拦截器通过 Xen 提供的超级调用服务将拦截到的数据传送到 Xen 内，等待进一步处理。经过修改的 Xen 虚拟机管理器维护着一个内部缓冲区(internal buffer)，用于暂存数据。监视器(monitor)拦截到的内存申请释放操作都会通过超级调用送往这个缓冲区。在缓冲区被锁定之后应用程序却崩溃的时候，可以通过另一个超级调用重置这个缓冲区。监控器通过 Xen 提供的超级调用服务从 Xen 的内部缓冲区中获取数据监视器拦截到的函数调用行为，然后通过图形界面而展现给用户，同时将数据储存下来以备后续分析用。通过自定义 6 个有关的超级调用来辅助完成对内存相关操作的捕捉，并获得有关的数据。

Monitor 主要是获取调用时的参数以及函数返回值等信息。其中，原函数的调用由 interceptor 负责提供。在经过 interceptor 修改了 malloc 和 free 的函数入口以后，调用 malloc 和 free 就被重定向到 monitor 中。此时可以直接通过读取栈中的地址，得知程序是要申请多大的内存空间或者释放哪里的内存空间。读取部分可以直接通过编写汇编语言实现，当然更简单的方法是自己编写一个具有同样声明的函数，这样对于被测进程来说，调用 malloc 就相当于直接调用 monitor 中的相关函数，于是获取参数的方式就变得和编写 C 语言代码时完全一样：直接访问形参。在图 6 中展示了调用 malloc 时，进入 monitor 时的场景。

4.3　实验结果分析

原型系统的评估可以分为 2 部分：有效性评估与性能评估。其有效性测试关注的是内存泄漏检测是否能正确地检测出已知的泄漏点，性能测试给出了其他内存泄漏检测系统与本原型系统的性能比较。

本文所有的实现在以下环境下完成：Intel HM61, 双核酷睿二代 i3 处理器 (i3-2120 CPU), 3.3 GHz，3 MB Cache, 2 GB 内存。Xen 4.1.2 版本，Linux 内核版本 Fedora 16 Linux-3.1.0-7.fc16.i686.PAE，CentOS 6，所用编译器 GCC 的版本 4.4.6，SATA 串行硬盘。

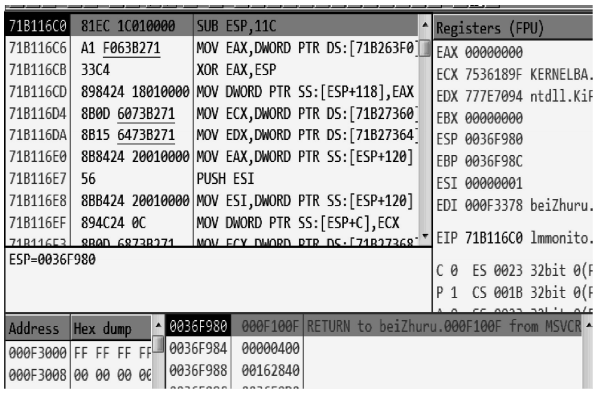

图 6　调用 malloc 时，进入 Monitor 时的场景

在评估过程中，作者比较了 VMLD、Purify[1] 这 2 个系统的开销。对于 SafeMen[2]和 Purify 等系统，它们都能够检查每一个内存操作并报告非法的访问。Purify 是一个非常有代表性的系统，它使用了一种 mark-and sweep 算法来对堆中的内存泄漏进行评估，特别是应用了保护性指针进行整个堆内存的跟踪[1]，因而对程序性能特别是服务器程序的性能要承担大的开销，有一定的代价，故在实际运行环境下一般不适合使用，而经常用于测试的环境下。尽管 SafeMen 比 Purify 具有较好的性能[2]，但必须有 ECC 内存这种特别物理器件的支持，因而通用性太差。所以作者在进行性能评估时选择与 Purify 做相关比较。

表 1　　VMLD 方法的有效性检测

被测程序	内在对象组数量	*Sleak*		*Aleak*	总泄漏数
		Suspected	*Leak*		
ProFTPD	57	27	5	3	11
Squid	79	31	7	2	13

从表 1 中可以看出，VMLD 系统在这样 2 个公认出现内存泄漏的开源系统中能实现有效的检测，最后被确认为泄漏的结果与 Purify 是一致的。同时，本文的实验可以跟踪出进程级泄漏，并能准确地定位到泄漏所在模块。为软件开发过程中产品上线前的测试大大减少测试时间。

对于 VMLD 系统的内部运行而言，具有较好的效率。在 10 个 *maxLifeTime* 时间内能实现有效的检测，如表 2 所示，从中可以看出，VMLD 系统在每一个 *maxLifeTime* 时间完成各阶段任务的处理时间非常少，具有较好的运行效率。

作者在 Xen 虚拟环境下分别运行 Squid 与 ProFTP 进行测试，通过运行 VMLD 检测系统与没有运行 VMLD 的整体性能进行比较。拟采用 LmbenchF [注2]F 性能分析工具进行比较，由于 VMLD 系统对检测对象的检测与分析主要是基于系统调用来实施捕捉内存资源的申请与释放，为了充分测试 VMLD 在实际运行时对整体系统的影响，从 Lmbench 中选择了与系统调用相关的基准测试指标进行 2 种情况下对比分析引入 VMLD 所带来的性能开销。

表 3 给出了相应的测试结果。分别选取了 4 项最基本的系统调用(null、open/close、read、write)，作为测试的依据。由表 3 可以看出，在 Xen 环境下分别运行 ProFTP 与 Squid 二者的测试结果相差并不大，基本上可以看作是一致的。但在 VMLD 系统加载以后，再分别来测试这 2 个程序，因为 VMLD 需要修改 Xen，同时对运行的进程进行动态注入，此时通过 LMbench 测试的相应数据要受到显著影响，就整体性能而言，因为 VMLD 频繁捕捉系统内存的申请与释放，肯定要带来较大的性能开销。

表 2　在 10 个 *maxLifeTime* 时间内，VMLD 系统平均在每一个 *maxLifeTime* 时间完成各阶段任务的处理时间（单位：s）

被测程序	完成分组时间	*Aleak* 检测时间	*Sleak* 检测时间	嫌疑的内存泄漏确认时间
ProFTPD	31	2.3	1.6	1.1
Squid	55	2.1	2.3	2.1

表 3　　LMbench 基准测试结果（单位：μs）

被测程序	*null*	*Open/close*	*read*	*write*
Xen+ProFTPD	0.55	13.3	2.3	2.1
Xen+Squid	0.56	14.2	2.3	1.8
Xen+VMLD+ProFTPD	5.71	17.3	4.6	6.1
Xen+VMLD+Squid	6.13	22.1	4.3	6.1

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系统调用、对 Xen hypervisor 管理器内的缓冲区维护以及频繁切换虚拟机所做的开销。其中，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。切换本身会耗费很多的处理器周期，因为切换本身而言，处理器不仅需要保存原状态，而且需要载入新状态，所以数据缓冲区的维护开销也不可忽视。

5　结束语

随着云计算、数据中心的发展，虚拟化计算环

注2：http://lmbench.sourceforge.net/.

境的应用越来越广泛，因没有被合理的释放而不能再被使用和释放的内存泄漏给 VMM 的内存管理带来了严重的挑战，特别是长时间不停机的应用系统，可能使虚拟机系统发生崩溃。本文利用虚拟机与虚拟机管理器的自省机制，通过建立应用程序运行的虚拟机状态自省，构建了内存泄漏的判定准则，可以生成内存泄漏视图；也实现了一种 VMLD 原型框架，并通过性能实验得出：本文的方法具有较好的检测能力和性能。但由于虚拟机管理器监控内存必须截获系统调用，虚拟机的切换也会引入很大的性能开销，因而性能方面还有待在下一阶段的工作中进一步改进。

参考文献：

[1] PURIFY R. Purify: fast detection of memory leaks and access errors[EB/OL]. http://www-01.ibm.com/software/awdtools/purify, 2012.

[2] QIN F, LU S, ZHOU Y Y. SafeMem: exploiting ECC memory for detecting memory leaks and memory corruption during production runs[A]. Proceedings of the HPCA[C]. San Francisco, USA, 2005. 291-302.

[3] BUNGALE P P, LUKC K. PinOS: a programmable framework for whole system dynamic instrumentation[A]. Proceedings of the ACM Conference on Virtual Execution Environments[C]. San Diego, California, USA, 2010. 137-147.

[4] 汪小林, 王振林, 孙逸峰等. 利用虚拟化平台进行内存泄露探测[J]. 计算机学报, 2010, 33(3):463-472.
WANG X L, WANG Z L, SUN Y F, *et al*. Detecting memory leak via VMM[J]. Chinese Journal of Computers, 2010, 33(3):463-472.

[5] 项国富, 金海, 邹德清等. 基于虚拟化的安全监控[J]. 软件学报, 2012, 23(8):2173-2187.
XIANG G F, JIN H, ZOU D Q, *et al*. Virtualization-based security monitoring[J]. Journal of Software, 2012, 23(8):2173-2187.

[6] 王丽娜, 高汉军, 刘炜等. 利用虚拟机监视器检测及管理隐藏进程[J]. 计算机研究与发展, 2011,48(8):1534-1541.
WANG L N, GAO H J, LIU W, *et al*. Detecting and managing hidden process via hypervisor[J]. Journal of Computer Research and Development, 2011,48(8):1534-1541.

[7] SHARIF M, LEE W, CUI W, *et al*. Secure in-VM monitoring using hardware virtualization[A]. Proc of the 16th ACM Conf on Computer and Communications Security[C]. New York, USA, 2009. 477-487.

[8] GARFINKEL T, ROSENBLUM M. A virtual machine introspection based architecture for intrusion detection[A]. Proc of the 10th Network and Distributed System Security Symp[C]. Berkeley: USENIX Association, 2003. 191-206.

[9] CHRISTIAN H, SCHNEIDER H, PFOHH H. A universal semantic bridge for virtual machine introspection[A]. HICISS[C]. 2011. 370-373.

[10] DOLAN-GAVITT B, LEEK T, ZHIVICH M, *et al*. Virtuoso: narrowing the semantic gap in virtual machine introspection[A]. IEEE Symposium on Security and Privacy[C]. 2011.297-312.

作者简介：

肖如良（1966-），男，湖南娄底人，博士，福建师范大学教授、硕士生导师，主要研究方向为云计算与物联网、Web 智能等。

姜军（1989-），男，福建三明人，福建师范大学硕士生，主要研究方向为虚拟化、内存泄漏检测等。

倪友聪（1975-），男，安徽合肥人，博士，福建师范大学讲师，主要研究方向为软件体系结构、虚拟化与云计算。

杜欣（1979-），女，新疆石河子人，博士，福建师范大学副教授，主要研究方向为演化计算、模式识别等。

谢国庆（1972-），男，湖南娄底人，硕士，福建师范大学讲师，主要研究方向为嵌入式系统。

蔡声镇（1954-），男，福建晋江人，福建师范大学教授、硕士生导师，主要研究方向为嵌入式系统、信息系统等。