# 一种基于虚拟机的动态内存泄露检测方法

蔡志强[1,2] 丁丽萍[1] 贺也平[1]

[1](中国科学院软件研究所基础软件国家工程研究中心 北京 100190)

[2](中国科学院研究生院 北京 100190)

**摘 要** 内存泄露是一种常见的系统安全问题。虚拟技术是云计算的关键技术,虚拟机环境下的内存泄露不容忽视。而基于虚拟机的内存泄露检测技术尚未成熟。分析虚拟机 Xen 内核源码中与内存分配有关的代码,提出一种动态检测虚拟机中内存泄露的方法。该方法记录应用程序对资源的申请、释放以及使用情况,插入监测代码,最终检测出内存泄露的代码。实验结果表明,该方法能够有效地检测 Xen 虚拟机中的内存泄露。

**关键词** 内存泄露 安全检测 虚拟机 Xen

**中图分类号** TP312 **文献标识码** A

DOI: 10.3969/j.issn.1000-386x.2012.09.003

## A VIRTUAL MACHINE-BASED DETECTION METHOD FOR DYNAMIC MEMORY LEAK

Cai Zhiqiang[1,2] Ding Liping[1] He Yeping[1]

[1](*National Engineering Research Center of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*)

[2](*Graduate University of Chinese Academy of Sciences, Beijing 100190, China*)

**Abstract** Memory leak is a common problem of system security. Virtualisation is a key technology for the cloud computing in which the memory leak detection can't be ignored. However the virtual machine-based memory leak detection technique has not been mature yet. In this paper we analyse the codes pertinent to memory allocation in source code of Xen Virtual Machine's kernel, and present a method to dynamically detect the memory leak problem in virtual machine. The method records the resources application, release and use of the application programs. By injecting monitoring code, we succeed in detecting the code of memory leak eventually. Experimental results show that, the method in this paper can effectively detect the memory leak problem of Xen Virtual Machine.

**Keywords** Memory leak Security detection Virtual machine Xen

## 0 引 言

内存泄漏[1]是指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。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。一般,我们常说的内存泄露是指堆内存的泄露。目前,虚拟技术作为云计算的关键技术得到了迅速发展,而虚拟机的安全问题越来越受到学术界和工业界的重视,虚拟机的内存管理和普通应用程序的内存管理一样,存在着内存泄露的风险。

一般地,内存泄漏的检测方法分为两大类[2],一类是动态检测方法,另一类是静态检测方法。动态检测方法主要原理是在程序中进行动态内存分配时,在堆中作以标记。当程序退出并释放所有已分配的内存时,检查堆上残留的对象,这些残留的对象就是程序中泄漏的内存。这种分析方法能够直接发现实际发生的程序缺陷。但是,动态特性要求实际执行程序引入较大性能和时间开销,另外由于程序执行的路径覆盖存在死角,检测结果完备性不足,漏报率较高。静态检测方法通过分析程序源代码,模拟所有可能执行路径,判定程序可能执行路径中存在的安全缺陷。这种方法无需实际执行程序,克服了动态分析性能开销较大的问题,但是由于静态分析无法精确判断程序的输入、环境变量等信息,模拟执行路径中可能存在不可行路径。因此,静态分析方法较高的误报率是至今未解的一个重要难题。本文通过研究 Xen 虚拟机的内存管理机制,提出了一种动态检测虚拟机中内存泄露的方法,该方法通过在虚拟机运行时,修改开源软件 Valgrind 的源码,在其中插入监测代码,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。与现有技术相比,本文所述的方法能够发现潜在的内存泄露,且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。

## 1 背景和相关工作

### 1.1 Xen 简介

Xen[3]是剑桥大学开发的一种开源虚拟机监控器(VMM 或

收稿日期:2011-12-02。国家自然科学基金项目(90818011);"核高基"国家科技重大专项(2010ZX01036-001-002);中国科学院知识创新工程重要方向项目(KGCX2-YW-125)。蔡志强,硕士生,主研领域:系统安全与可信计算。丁丽萍,研究员。贺也平,研究员。

Hypervisor) 。Xen 直接运行在物理硬件之上,并向上提供可以运行操作系统的虚拟化环境,称为域。VMM 具有最高特权级,控制运行在其上的域。Xen 上有一个特权域称为 Dom0,其他域称为 DomU 。Dom0 控制硬件设备并为用户提供管理 DomU 的接口。Dom0 是 Xen 启动后运行的第一个虚拟域,在系统中具有重要的作用,一方面它具有管理控制功能,为使用者提供用户界面; 另一方面它是设备驱动程序域,运行着所有硬件设备的驱动。Xen 的体系结构如图 1 所示。

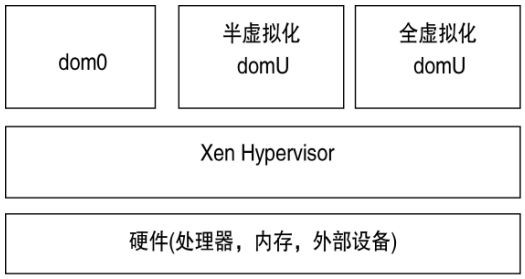


图 1　Xen 的体系结构

Xen 支持半虚拟化和全虚拟化的客户机。在半虚拟化方式下,客户机需要修改操作系统源代码,通过 Xen 提供的 Hypercall 接口来完成特权操作; 全虚拟化不要求修改客户机代码,但是需要 CPU 的硬件虚拟化支持[4] ,主要用于 Windows 等闭源操作系统。

### 1.2　相关工作

目前,国内外内存泄露检测的技术和方法很多,检测工具也有很多已经产业化。一般地,用于检测虚拟机内存泄露的方法[5] 主要有:

1) *静态方法*　这类方法采用基于源代码的静态检测方法,主要检查程序语义它是利用对源代码的静态分析来查找程序缺陷的一种检测方法。由于检测目标程序时不需要运行测试用例,减轻了构建测试平台的开销( 如计算能力要求,存储空间要求等) 。静态分析能够遍历程序中的每一个分支,它所提供的信息一般都比较全面而正确,而随之而来的也是分析实现的高代价和相对较高的误报率。

2) *动态方法*　动态检测方法包括运行时动态检测法和内存回收的方法。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 IBM Rational Purify 工具属于这一类。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。动态分析是在实际运行过程中基于具体数据的基础上,因此具有较高的准确性。另一种动态检测方法是内存回收的方法。一些编程语言如 Java 采用了 Smart Pointer 和 Garbage Collection[6] 的机制来进行内存的管理。内存回收算法的功能是自动进行内存搜索,从而免去程序员的手工操作,然而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,因而它在 C/C + + 领域的应用不是十分普遍。垃圾收集存在一些缺陷: 垃圾收集暂停、因为垃圾收集而产生的 CPU 时间损失等。因此,垃圾收集需要付出程序执行效率上的代价,包括对性能的影响、暂停、配置复杂性和不确定的结束等[7] 。

本文通过研究 Xen 虚拟机的内存管理机制和虚拟机中的内存虚拟化技术,提出了一种动态检测虚拟机中内存泄露的方法。该方法基于虚拟机 Xen 的虚拟机监控器 VMM,对运行在虚拟机中的应用程序申请和释放内存资源进行动态捕获和记录,属于动态的检测方法。通过修改开源软件 Valgrind[8] 的源码,在其中插入监测代码[9] ,并且修改 Valgrind 与 Xen 虚拟机交互的 Hypercall 代码,进而动态截获虚拟机中申请和释放内存的函数,并在此基础上分析应用程序所申请的内存资源,推测出存在内存泄露的原因并进行监控。同时,利用 Valgrind 工具的判定内存泄露的规则,找出系统中应用程序的内存泄露的原因并最终给出分析报告。

## 2　方法的设计与实现

### 2.1　原　理

研究发现,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 Global-Free、Posix 规定的 malloc 和 free 接口、Linux 内核的 kmalloc、vmalloc 和 vfree 等接口[10] 。因此,动态检测方法通过研究内存管理机制,截获内存申请和释放的函数,对源代码进行修改。例如可以通过重载 malloc/free 等内存管理函数、memcpy/strcpy/memset 等内存读取函数和指针运算等,以达到控制内存操作功能。在 C 语言中,对于内存的分配与回收是通过 malloc 以及 free 实现的( C ++ 中类似,用 new 及 delete) 。因此,可以重载这两个方法,通过钩子程序及宏定义的方式替换,即将标准 C 语言中的 malloc 以及 free 函数利用钩子程序将其重定向到自己实现的另外的两个函数 my_malloc 及 my_free 中,在这两个自定义的函数中它们所起的作用同标准 C 语言库中的 malloc 以及 free 所起的作用基本一致,即进行地址的分配与回收。但是,其中增加了两个功能,一个功能是记录下调用这个函数的 C 源文件以及在源文件中所处的位置。另外一个功能是在自定义的一张 Hash 表中增加或删除此次调用的所有信息。综上,通过管理所有内存块,无论是堆、栈还是全局变量,只要有指针引用它,它就被记录到一个全局表中[11] 。记录的信息包括内存块的起始地址和大小等。对于在堆里分配的动态内存,可以通过重载内存管理函数来实现。对于全局变量等静态内存,可以从符号表中得到这些信息。

虚拟机 Xen 的虚拟机管理器 Hypervisor[12] 的内存泄露检测方法与已有的检测方法的显著不同之处在于: 因为 Xen 采用了影子页表[13] 、内存虚拟化管理[14] 等技术,使得 Guest OS 对内存的申请和释放得通过陷入 Hypervisor 的超级系统调用 Hyper-call。因此,为了检测 Xen Hypervisor 的内存泄露问题,本文提出修改 Valgrind,使其支持对 Hypercall 的检测,从而完成对于 Xen 的内存泄露检测。因为 Valgrind 工具的检测原理是通过修改可执行文件来进行内存泄露检测,所以不需要重新编译程序。但它并不是在执行前对可执行文件和所有相关的共享库进行一次性修改,而是和应用程序在同一个进程中运行,动态地修改即将执行的下一段代码。Valgrind 是插件式设计的,它的 Core 部分负责对应用程序的整体控制,并把即将修改的代码,转换成一种中间格式,这种格式类似于 RISC 指令,然后把中间代码传给插件。插件根据要求对中间代码修改,然后把修改后的结果交给 Core。Core 接下来把修改后的中间代码转换成原始的 x86 指令,并执行它。

基于以上原理,本文提出一种基于虚拟机的动态检测内存泄露的方法。该方法的具体实现为首先分析 Xen 内核源码中与内存分配和释放相关的代码,重点分析对象为 Xen 源码中内存管理模块的 Hypercall 函数,然后修改开源工具 Valgrind,在 Valgrind 的 MemCheck 功能模块里面,利用钩子函数和宏定义的

方式重载/替换插入监测代码，动态截获 Xen 中申请和释放内存的函数，将这些函数替换为我们定义的钩子函数，当函数启动时，函数里记录调用函数的文件及行号信息，并且定义一张全局 Hash 表来记录所有调用信息和内存分配情况，包括堆、栈和全局变量，同时根据此信息维护一个动态资源使用列表。当函数退出时，扫描最后的游离对象。当虚拟机运行时，重编译后的 Valgrind 的 memcheck 工具用于实现内存泄露的检测。

### 2.2 整体架构

本文提出的虚拟机内存泄露检测方法是在虚拟机监控器(VMM，在 Xen 中称为 Hypervisor)[15] 中实现的，并对客户操作系统和应用程序透明。图 2 为此方法的整体架构。

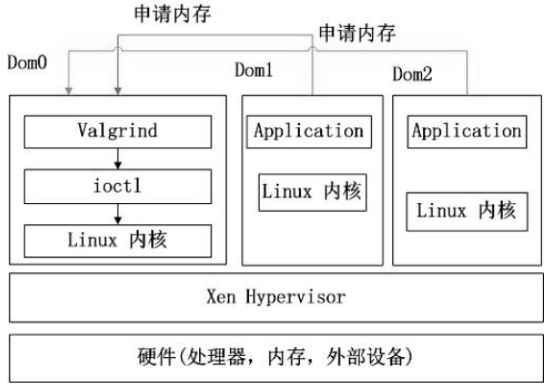


图 2 内存泄露检测整体架构

DomU 里的应用程序向 Dom0 申请内存，当虚拟机运行时，启动修改后的 Valgrind，通过 ioctl 接口陷入 Xen Hypervisor 的 Hypercall 调用，利用 Valgrind 里的检测函数，截获这些系统调用，替换为钩子函数，获得调用函数的内存分配和释放信息，并分析其内存泄露情况。

### 2.3 关键技术

(1) 内存虚拟化技术

虚拟化为整个机器提供虚拟的设备，内存虚拟化的一般方法包括虚拟化页表和影子页表技术。VMM 对物理内存有最终的控制权，它控制将客户物理地址空间映射到主机物理地址空间从而顺利的实现内存虚拟化。

虚拟化页表有几种实现形式，第一种是被称为保护访问的页表。其主要特点为：① 使用了页保护来探测页表的变化。② 读页表的时候会产生 VM 退出，Guest OS 保留原始的 PTE 值。

第二种实现内存虚拟化技术的方式是影子页表。首先 Guest OS 要能够自由的改变页表。Xen 实现内存虚拟化的方法是影子页表技术[16]。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 MMU(Memory Management Unit)[17] 进行地址转换。

(2) Hypercall 的实现机制[18]

与系统调用类似，Xen 中的 Hypercall 是通过软中断(中断号 0x82)来实现的。首先在文件 xen/include/public/xen.h 中定义了 45 个超级调用，其中有 7 个是平台相关调用。而在文件 xen/arch/x86/x86_32/entry.S 中定义了超级调用表，通过超级调用号索引就可以方便的找到对应的处理函数。超级调用页是 Xen 为 Guest OS 准备的一个页，可以做到不同 Guest OS 有不同的超级调用页内容。它的实现过程为：_HYPERVISOR_xxxx 在超级调用页上找到相应的代码；_hypercall2() 调用超级调用页内的代码；通过 hypercall_page_initialise 实现 HVM、Dom0、DomU 的不同跳转；调用 hypercall() 来检查有效性、执行相应的服务例程(do_xxhyper 名_)并返回。

超级调用只能由内核来调用，而应用程序无法直接调用。应用程序申请超级调用的过程为：首先打开 Xen 提供的内核驱动：/proc/xen/privcmd；然后通过 ioctl 系统调用来间接调用 hypercall；如下所示：

```
fd = open( " /proc/xen/privcmd" , O_RDWR) ;
privcmd_hypercall_t hcall = {
__HYPERVISOR_print_string,
{ message, 0, 0, 0, 0}
};
ioctl( fd, IOCTL_PRIVCMD_HYPERCALL, &hcall) ;
```

复杂一点的超级调用申请的过程为(以_HYPERVISOR_domctl 超级调用为例)：首先通过 pyxc_domain_create() 获取要创建的 domain 的相关信息；然后通过 xc_domain_create() 创建控制结构体变量 domctl；接着通过 do_domctl() 生成超级调用请求；然后传递请求到 OS 内核：do_xen_hypercall()；最后 do_privcmd 通过 ioctl 来完成由 3 环到 1 环的转变，并完成超级调用。

## 3 实验验证与分析

如前所述，本文提出的方法主要采用动态分析方法，采用了 Valgrind 套件中的 Memcheck 工具，通过修改 Valgrind，使得它能支持 Xen 的 privcmd ioctls / hypercalls，从而利用 Memcheck 工具，动态检测 Xen Hypervisor 中实际存在的内存泄露问题。

Valgrind 是一个 GPL 的软件工具包。该工具包主要用于 Linux(For x86, amd64 and ppc32)程序的内存调试和代码剖析。Valgrind 的工具包主要包括 Memcheck、Callgrind、Cachegrind 等工具，每个工具都能完成一项任务调试、检测或分析。可以检测内存泄露、线程违例和 Cache 的使用等。

### 3.1 实验环境及实验过程

在开发基于 Xen 虚拟机的安全操作系统的过程中，我们运用了第 3 节中介绍的内存泄露检测方法对 Xen 虚拟机的脆弱性进行了验证。实验采用的硬件平台为联想 ThinkCentre 台式电脑(CPU 为英特尔酷睿 2 四核 Q6600@2.40GHz，内存为 4GB)，系统搭建了以 Fedora8(内核为 2.6.18.8)为 Domain0，以 Xen 3.3.3 为Hypervisor，以 Fedora12 为 DomainU 的实验环境。实验过程分为两步：

第一步：工具的改进、配置与安装。首先下载 valgrind-3.6.1 源码，依据第 3 节所述的方法编写支持 Hypercall 的补丁。在 Xen 内存管理函数里，我们定义了若干 Hook 函数用于截获其内存申请和释放，具体包括了受监控函数的类别、参数、获取函数调用栈、申请和释放函数的地址、动态更新内存资源等占用信息。并且维护资源使用列表，程序退出后扫描剩余孤儿对象。我们将上述补丁文件名为 xen_valgrind_patch.patch。使用命令：patch -p1 <../xen_valgrind_patch.patch 运行该补丁。然后使用命令 aclocal && autoheader && automake -a && autoconf，通过 Gnu autoconf 和 automake 工具重新生成 Makefile 文件。最后使用命令./configure -with-xen 进行配置，并使用 make 工具对修改后的 Valgrind 源代码进行编译和安装。至此，支持 Hypervisor

的 Valgrind 安装完成。

**第二步: 动态执行和分析。** 运行修改后的 Valgrind 工具 memcheck,分析目标代码中内存泄露和错误使用。例如为了检测 GCC 工具的内存泄露,启动 Valgrind: Valgrind --tool = memcheck --track - origins = yes --trace - children = yes --leak - check = full --show - reachable = yes --log-file = "out. log" gcc - o a. out test. c; 其中 --leak - check = full 指的是完全检查内存泄漏, --show - reachable = yes 是显示内存泄漏的地点, --trace - children = yes 是跟入子进程, --log - file = filename 将输出的信息写入到 filename 的文件里。最后分析文件 out. log 输出的 debug 信息并定位发现内存泄露的代码位置。

### 3.2　实验结果

经过对实验文件 out. log 进行分析,实验表面,在本文所述平台上,利用本文设计的工具,可动态的检测出虚拟机下的内存泄露情况,实验结果归纳如下:

(1) 确定性内存泄露。检测结果代码片段为:

```
==3916== 14 bytes in 1 blocks are indirectly lost in loss record 1 of 3
==3916== at 0x4005EAE: malloc (vg_replace_malloc. c:236)
==3916==    by 0x43A88A: vasprintf (in /lib/libc - 2.7. so)
==3916==    by 0x41FA6D: asprintf (in /lib/libc - 2.7. so)
==3916==    by 0x4A4F5ECB: setup_probe_watch (in /usr/lib/libblktap. so. 3.0.0)
==3916==
==3916== 30 (16 direct, 14 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 3
==3916== at 0x4005EAE: malloc (vg_replace_malloc. c:236)
==3916==    by 0x4A4F5EDC: setup_probe_watch (in /usr/lib/libblktap. so. 3.0.0)
==3916==
==3916== 136 bytes in 1 blocks are possibly lost in loss record 3 of 3
==3916== at 0x40051ED: calloc (vg_replace_malloc. c:467)
==3916==    by 0x3CC109: _dl_allocate_tls (in /lib/ld - 2.7. so)
==3916== by 0x56CBFE: pthread_create@@GLIBC_2.1 (in /lib/libpthread - 2.7. so)
==3916==    by 0x804BE54: bidir_daemon_launch (bidir-daemon. c:83)
==3916==    by 0x76F0B: ???
==3916==
==3916== LEAK SUMMARY:
==3916==    definitely lost: 16 bytes in 1 blocks
==3916==    indirectly lost: 14 bytes in 1 blocks
==3916==      possibly lost: 136 bytes in 1 blocks
==3916==    still reachable: 0 bytes in 0 blocks
==3916==         suppressed: 0 bytes in 0 blocks
```

此处表明 setup_probe_watch 中的 236 行调用了 malloc 函数,但是没有对其进行处理,造成了内存泄露,而 setup_probe_watch 在 libblktap. so. 3.0.0 库中。libblktap. so. 3.0.0 是 Blktap 的动态运行库,而 Blktap 是一个运行在 Domain 0 用户空间的程序,给用户层提供对虚拟磁盘文件读写操作的接口。blktap 目录中定义了一些用户层的虚拟磁盘访问接口。

(2) 使用未初始化的内存。检测结果代码片段为:

```
==3913== Conditional jump or move depends on uninitialised value (s)
==3913==    by 0x478CA75: pyxc_sched_id_get (xc. c:1540)
==3913== Uninitialised value was created by a stack allocation
==3913==    at 0x4A4AF480: xc_sched_id (in /usr/lib/libxenctrl. so. 4.0.0)
==3913== Use of uninitialised value of size 4
==3913==    by 0x478E98B: pyxc_physinfo (stdio2. h:34)
==3913== Uninitialised value was created by a stack allocation
==3913==    at 0x478E8EF: pyxc_physinfo (xc. c:1152)
==3913== Conditional jump or move depends on uninitialised value (s)
==3913==    by 0x478B8F1: pyxc_numainfo (xc. c:1336)
==3913== Uninitialised value was created by a stack allocation
==3913==    at 0x4A4A6290: xc_availheap (in /usr/lib/libxenctrl. so. 4.0.00029
```

此处表明在文件 xc. c 的 1540 行处调用的 pyxc_sched_id_get 函数使用了未初始化的变量,该函数被 libxenctrl 的 xc_sched_id 调用; 同样在 xc. c 文件的 1152 行处调用了 pyxc_physinfo 使用了未初始化的变量,在 xc. c 文件的 1336 行处调用了 pyxc_numainfo 使用了未初始化的变量,该函数被 libxenctrl 的 xc_availheap 调用。xc. c 是 xen 的影子页表实现的重要组成部分,它主要实现了对分页 paging 的支持; 而 libxenctrl 是一个 C 库,它提供了一些简单易用的 API,使用户程序可以方便地和 Hypervisor 进行通信。它的工作原理很简单,它封装了 dom0 中的/proc/xen/privcmd,/dev/xen/evtchn 以及/dev/xen/gntdev 提供的 IOCTL 接口。

### 3.3　性能评价与分析

本文设计了如下实验用来测试本实现对运行在 Xen 之上的应用程序的性能影响。利用本论文设计的工具,我们截获了 GCC 中的内存申请和释放的函数,然后检测虚拟机 Xen 平台下 GCC 的内存泄露情况,并且对比了非虚拟机平台下 GCC 的内存泄露情况。最终结果如表 1 所示。

表 1　内存泄露检测在 Xen 的性能对比

| | 非虚拟机 Xen 平台 | Xen - 3.3.3 - mld |
|---|---|---|
| GCC 4.4.5 | 25 秒 | 27 秒 |

由表 1 可知, GCC4.4.5 在正常的 Xen 虚拟机上编译一个应用程序所需时间为 25 秒,而利用本文的内存泄露检测工具在 Xen 虚拟机上编译同一个应用程序所需时间为 27 秒,可见内存泄露探测工具引起的性能损失在 10% 以内。

## 4　结　语

内存泄露是影响信息系统研究可用、可靠的基础性问题。由于内存虚拟化技术的应用、虚拟机的内存管理更加复杂,引起内存泄露的因素极大地增加了。

本文通过分析 Xen 虚拟机内核中影响内存资源分配和释放的资源,设计并实现了一种基于 Xen 虚拟机的动态内存泄露检测方法。与其他基于虚拟机架构的方案相比,该方法可扩展性强、易用性好,对系统资源的保护比较全面,并且具有较高的效率。存在的不足和改进的方向如下:

(1) 能够检测 new/delete/malloc 引起的内存泄漏问题,但是对于一些内存分配的 Windows API 函数(如 RtlHeapAlloc)并没有拦截检测,尽管这些函数出现概率不高,但是在特定的程序中可能频繁出现。下一步将增加相应的拦截函数,跟踪此类函数产生的堆内存的变化情况,以检查此类内存泄漏问题。

(下转第 153 页)

表 1 基于统计的 2-Gram 语言模型的结果统计

| | *Correct*(%) | *Acc*(%) | *H* | *S* | *D* | *I* | *N* |
|---|---|---|---|---|---|---|---|
| 单词 | 7.22 | – | 77 | 989 | – | – | 1066 |
| 字母 | 66.7 | 45.68 | 4837 | 2352 | 121 | 1498 | 7310 |

表 2 基于上下文有关的规则语言模型的结果统计

| | *Correct*(%) | *Acc*(%) | *H* | *S* | *D* | *I* | *N* |
|---|---|---|---|---|---|---|---|
| 单词 | 93.62 | – | 998 | 68 | – | – | 1066 |
| 字母 | 97.25 | 95.55 | 7109 | 185 | 16 | 124 | 7310 |

表 1 和表 2 描述了两种模型应用在联机手写维吾尔文单词识别中的结果统计，从中可以看出基于上下文有关的规则语言模型的正确识别率远高于基于统计的 2-Gram 语言模型的正确识别率，原因在于，上下文有关的规则语言模型很大程度上弥补了由训练得到的模型不够合理产生的误识，例如，如图 1 所示单词，单词字母数目多，复杂程度高，但基于规则的语言模型正确识别出了该单词，基于统计的模型未能识别出。

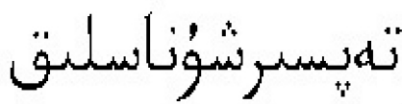

图 1 示例单词

此外，由于维吾尔文单词字母组成差异性较大，样本数据量不充分，2-Gram 统计语言模型里面字母关联的概率只是前后两个字母关联的概率，不能完全表示每个单词中全部字母连接关系的规律性，而且由该语言模型识别产生的不正确的维吾尔文单词概率较高。

实验结果表明，基于 HMM 的联机手写维吾尔文单词识别中两种语言模型的比较研究是具有合理性的，且基于上下文有关的规则语言模型运用在小量单词库的识别中识别效果相对较好。

## 4 结 语

本文从联机手写维吾尔文单词识别中两种语言模型的比较研究得出，2-Gram 统计语言模型具有一定的局限性，也可以尝试其他的三元文法或是 N 元文法，进一步改进统计语言模型，充分发挥统计语言模型的优点。此外，把两种语言模型结合起来，取长补短，对于维吾尔文单词识别也具有一定的可行性。

## 参 考 文 献

[1] 哈力木拉提，阿孜古丽. 多字体印刷维吾尔文字符识别系统的研究与发展[J]. 计算机学报，2004，27(11)：1480－1484.
[2] 宋扬. 基于 HMM 的联机手写汉字识别[D]. 西安电子科技大学，2009：35－37.
[3] Biadsy F，El-Sana J，Habash N. Online Arabic Handwriting Recognition Using Hidden Markov Models [C]//Tenth International Workshop on Frontiers in Handwriting Recognition，2006.
[4] Slimane F，Ingold R，Alimi A M，et al. Duration Models for Arabic Text Recognition using Hidden Markov Models [C]//2008 International Conference on Computational Intelligence for Modelling，Control and Automation，2008：838-843.
[5] Hamdani M，Abed H E，Kherallah M，et al. Alimi. Combining Multiple HMMs Using On-line and Off-line Features for Off-line Arabic Handwriting Recognition [C]//International Conference on Document Analysis and Recognition，2009：201-205.
[6] 田斌，田红心，易克初. 一种改进的汉语 N 元文法统计语言模型[J]. 西安电子科技大学学报，2000，27(1)：62-64.
[7] 陶梅. 基于 HTK 的维吾尔语连续语音识别研究[D]. 新疆大学，2008：28-39.
[8] Young S，Evermann G，Gales M. The HTK Book [M]. Cambridge University Engineering Department，2006：110-112.

(上接第 13 页)

(2) 鉴于多线程程序在实际中的广泛应用，下一步将扩展实现支持多线程的堆内存泄漏检测。

## 参 考 文 献

[1] Rational Purify. Purify: Fast Detection of Memory Leaks and Access Errors [EB/OL]. http://www.ibm.com/software/awdtools/purify/.
[2] Qin Feng，Lu Shan，Zhou Yuanyuan. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption during Production Runs [C]//HPCA'05，2005.
[3] Paul Barham，Boris Dragovic，Keir Fraser. Xen and the art of virtualization [C]//Proceedings of the nineteenth ACM symposium on Operating systems principles，October 19-22，2003.
[4] Intel. Intel Virtualization Technology: Hardware support for efficient processor virtualization [S]. Intel.
[5] Xie Yichen，Alex Aiken. Context-and Path-sensitive Memory Leak Detection [C]// Proceedings of ESEC/FSE 2005，Lisbon，Portugal，2005.
[6] Helena Aberg Ostlund. Memory Leak Detection with the JRockit JVM [M]. 2005.
[7] Tsai T，Vaidyanathan K，Gross K. Low-Overhead Run-Time Memory Leak Detection and Recovery [C]//PRDC'06，2006.
[8] Julian Seward，Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision [C]// Proceedings of the USENIX Annual Technical Conference，USA，2005：17-30.
[9] Prashanth P Bungale，Luk Chi－Keung. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation [C]//ACM Conference on Virtual Execution Envi-ronments (VEE'07)，2007.
[10] 李肖坚，夏冰，钟达夫. 一种缓冲区溢出防御虚拟机的研究与设计[C]//2005 年国家网络与信息安全技术研讨会. 2005.
[11] Goldberg R P. Architecture of Virtual Machines [C]//Proceeding of the Workshop on virtual computer systems，1973：74-112.
[12] Reiner Sailer，Trent Jaeger，Enriquillo Valdez，et al. Building a MAC-Based Security Ar-chitecture for the XEN Open-Source Hypervisor [C]//21st Annual Computer Security Applications Conference，December 2005，Arizona，2005.
[13] Carl A Waldspurger. Memory Resource Management in VMware ESX Server [C]//Proceedings of Fifth Symposium on Operating Systems Design and Implementation (OSDI)，2002.
[14] Barham P，Dragovic B，Fraser K，et al. Xen and the Art of Virtualization [C]//Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP03)，ACM，2003.
[15] 鲁松. 计算机虚拟化技术及应用 [M]. 北京：机械工业出版社，2008：9－11.
[16] Keahey K，Doering K，Foster I. From sandbox to playground: Dynamic virtual environments in the grid. [C]//5th International Workshop on Grid Computing (Grid 2004).
[17] Pratt I，Magenheimer D，Blanchard H，et al. The Ongoing Evolution of Xen [C]//Proceedings of the Linux Symposium，2006，2：255-266.
[18] Avi Kiviy，Yaniv Kamay，Dor Laor，et al. KVM：the linux virtual machine monitor [C]//proceedings of the Linux Symposium，Canada，June 2007.