基于动态插桩的C/C++内存泄漏检测工具的设计与实现

曾佳平，杨秋辉†，汪华龙，徐保平，黄 蔚
（四川大学 计算机（软件）学院，成都 610065）

摘 要：针对C、C++程序常出现的内存泄漏、内存越界访问、内存的不匹配释放等错误进行了研究，分析了现有的内存错误检测工具和方法，在基于开源的动态二进制插桩框架Pin的基础上，采用函数族的内存信息块管理方法和生命周期法，实现了在Linux平台下运行的内存检测工具MemGuard原型。该原型能有效地检测出内存泄漏、内存越界访问、内存的不匹配释放等问题，并通过与运行在Valgrind上的工具Memcheck的对比实验证明了该原型的有效性以及高效性。

关键词：内存泄漏；动态二进制插桩；生命周期；MemGuard原型；Valgrind
中图分类号：TP334 **文献标志码**：A **文章编号**：1001-3695(2015)06-1737-05
doi:10.3969/j.issn.1001-3695.2015.06.030

Design and implementation of memory leak detection tool of C/C++ based on dynamic instrumentation

Zeng Jiaping, Yang Qiuhui†, Wang Hualong, Xu Baoping, Huang Wei
(*College of Computer (Software), Sichuan University, Chengdu* 610065, *China*)

Abstract: This paper studied the issue of memory leak, cross-border access memory, the memory does not match the release of C or C++ program. By studying the existing tools and methods, this paper used the method of memory block information management of function-based family and life cycle to develop a prototype of memory leak detection tool MemGuard which was based on Pin of dynamic binary instrumentation framework of open source. The prototype can effectively detect memory leaks, cross-border access memory, memory does not match the release. Finally, through the Compared experiment to Memcheck running on the Valgrind proves the high effectiveness of the prototype of MemGuard.

Key words: memory leaks; dynamic binary instrumentation; life cycle; prototype of MemGuard; Valgrind

C、C++因其灵活性、高效性等特点一直以来都是主流程序设计语言之一。它们与Java等高级语言相比，在编程中程序员需要自己管理内存，并对程序中所涉及的内存操作有很清晰的认识。内存问题很难让人察觉，特别是内存泄漏，它不同于其他内存错误，如多次释放、野指针、或者是数组越界等容易暴露出来，内存泄漏错误一般隐藏得比较深，在数万行的代码之中寻找内存泄漏无异于大海捞针。因此借助内存分析工具来检验内存错误是非常具有实用价值的，不仅能够提高软件的质量，还能缩短软件的开发周期。

目前国内外已经有一些内存检测工具，如PC-Lint、Polyspace属于静态检测工具，memwatch、dmalloc、Valgrind、Fense、DDMEM属于动态检测工具，但它们都有其相应的缺点，比如memwatch、dmalloc、DDMEM、Fense需要修改源程序，Valgrind上的工具Memcheck只能运行在Linux平台，同时开销也比较高，影响程序的运行效率。

本文针对嵌入式软件动态内存泄漏检测的问题，在调研和分析了现有内存检测方法的基础上，基于动态代码插装工具Pin，设计实现了一个动态内存检测工具MemGuard，能够检测C/C++中出现的内存泄漏、内存访问越界、不匹配释放、多重释放等内存错误，不仅支持C标准内存管理库函数，也支持C++动态内存分配，同时也能对用户自定义的内存管理函数进行监控和管理。最后设计实验对MemGurad的功能、性能进行验证，同时将其与现有的内存检测工具Memcheck进行对比，结果显示MemGuard能够实现嵌入式软件的内存检测，且有良好的性能指标。该工具的主要特点是不需要修改被测程序源代码，对被测程序的运行效率影响较小，进行内存检测的系统开销较小。

1 MemGuard的分析与设计

嵌入式软件内存问题主要包括内存泄漏、内存访问越界、不匹配释放、多重释放等内存错误。对内存问题的检测主要还是通过动态分析的方法来获取，跟踪程序运行时的内存操作使用情况。动态分析方法是通过在实际环境中运行程序来分析程序性质或者寻找缺陷的过程，需要向程序提供具体的数据。

收稿日期：2014-05-18；**修回日期**：2014-07-13

作者简介：曾佳平(1989-)，男，四川成都人，硕士研究生，主要研究方向为软件自动化测试；杨秋辉(1970-)，女(通信作者)，山东青岛人，副教授，博士，主要研究方向为软件工程、软件自动化测试、软件项目管理、实时软件系统、数据库系统应用、计算机信息与网络(yangqiuhui@scu.edu.cn)；汪华龙(1988-)，男，硕士研究生，主要研究方向为软件自动化测试；徐保平(1990-)，女，山东人，硕士研究生，主要研究方向为软件自动化测试；黄蔚(1990-)，女，广西人，硕士研究生，主要研究方向为软件自动化测试.

由于其是在真实的环境中运行，所以具有比较高的准确性。

在分析程序内存问题时，一个关键的步骤便是获取程序的内存相关操作信息，获取程序运行的内存相关信息是进行内存分析的基础。

MemGuard 要检测程序存在的内存问题，就必须截获内存操作信息。本文中采取的方式是截获相关内存管理函数的控制权，如 C 标准库中的库函数 malloc、realloc 等，C ++ 中的操作符 new、delete 等。获得内存管理函数的控制权以后，可以采取监控和替换的方式在被测程序中添加插桩代码，记录内存的操作行为，包括记录内存相关信息，并对这些信息进行统一管理。记录内存相关信息后，需要有一系列的规则和方法来判定内存出现的错误，比如怎么判定程序中出现内存泄漏错误，或者出现指针的多次释放等。内存检测工具检测出内存错误后，需要报告内存相关错误，同时还需要提供可以定位该错误的相关信息，比如产生该内存错误语句所在的文件名和行号，甚至还需要提供当产生错误时当前堆栈中相关函数的调用关系。

1.1 工具设计思路和系统框架

MemGuard 通过 Pin[1-3] 提供的相关 API，获取应用程序中与内存相关函数的控制权。获取执行权限以后，可以插入插桩代码来对内存相关操作进行监控。比如申请函数，用一个数据结构（内存信息块）记录该函数中所涉及的内存相关信息，如申请内存的大小，返回给用户内存块首地址、申请的时间、当前堆栈中最近几层函数的调用地址等。MemGuard、Pin、应用程序的关系如图 1 所示。

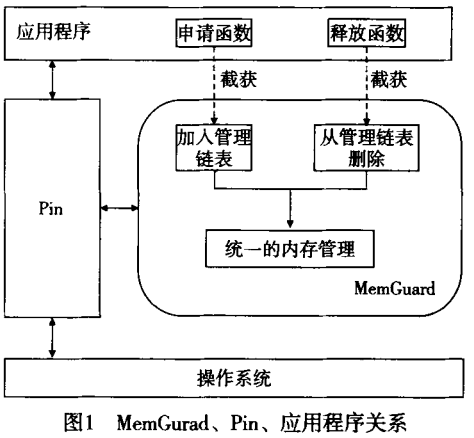

图1 MemGurad、Pin、应用程序关系

MemGuard 包括内存管理函数接口模块、内存块管理模块、内存错误检测模块、日志输出模块、应用程序函数调用跟踪模块等。其中内存管理函数接口模块是 MemGuard 和被测程序的接口模块，日志输出模块是 MemGuard 和用户的交互模块。后文中将给出内存泄漏模块的设计，其他模块原理大体相同，都是采用内存信息模块中的数据进行内存错误判定。各模块的组织关系如图 2 所示。

1.2 功能模块设计

本节主要给出内存分配函数接管模块、内存信息管理模块以及内存错误检测模块中内存泄漏模块的设计。由于内存泄漏是内存错误模块的核心模块，且其他子模块原理大体相同，加上篇幅所限，本文不再赘述其他模块的设计过程。

1.2.1 内存分配函数接管模块设计

1）C 动态内存函数接管　本文中采用 Pin 提供的接口对 C 标准内存管理函数进行接管，不需要对源代码进行任何修改，也不需要重新编译源代码。分析 C 标准库中的三个分配函数可以发现，对于 realloc 和 calloc 都是可以通过 malloc 来实现的，因此对于 C 标准库的函数可以归类为申请函数 malloc 和释放函数 free，在函数族管理中配对的申请和释放函数就为 malloc 和 free。因此将 malloc 函数替换以后，就接管了 malloc 函数的执行权限，可以在里面记录内存块的相关信息；在替换的 free 函数后要进行相应的收尾工作，将与该内存块相对应的信息删除或者释放。

2）C ++ 动态内存接管　在采用 Pin 来定位函数信息时，需要确认函数的符号信息，通过找到指定的符号来确定函数调用的起始处。对于 C ++ 中分配动态内存的 new 和 new[ ] 在不同编译器中编译的符号不一样，可以编写一段只包含 new[ ] 和 delete[ ] 的小代码 test. c 来查看当前编译器编译出的符号信息，在 Linux 环境中采用 g ++ -S test. c 的命令将该程序编译成汇编文件，再查找汇编文件中 main 例程中的 call 指令，便可知在该编译器下对应的符号信息。在 C ++ 程序中通过截获对应的符号信息就可以获取到该函数的控制权限，捕获的原理在 C 函数族已经阐述，此处不再描述。

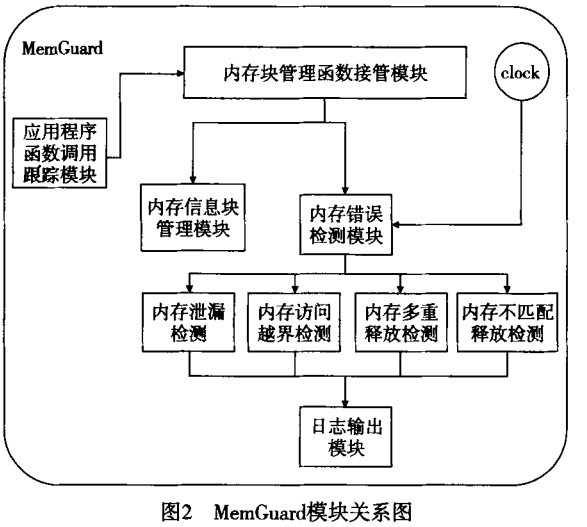

图2 MemGuard模块关系图

1.2.2 内存信息块管理模块设计

内存信息块是用来记录被测程序中申请的动态内存块的相关信息的数据结构。对于不同函数族申请的内存信息块放在一个全局链表中进行管理比较混乱而且不容易区分，因此采用以函数族为单位的管理方式来解决此问题。每一个函数族都有一个完全属于本函数族的内存信息块链表，所有的函数族表头都链成一个链表，由全局指针 g_FuncManageHead 索引。

1.2.3 内存泄漏检测模块设计

本文采用内存块的生命周期法[4] 解决运行过程中的内存泄漏问题。采用生命周期法需要将程序中分配的所有内存信息块统一进行分组管理，以执行到内存分配函数时当前堆栈中的最近四层函数返回地址作为关键字进行分组。内存块分配时的流程如图 4 所示。

内存块释放流程如下：

a）通过内存块管理信息中记录的组号，找到该内存块所在的组链表。

b）计算当前内存块的生存周期，如果该内存块的生存周期大于该组所保存的最大生命周期，则将该组中最大生命周期更新为当前内存块的生存周期。

c）将该内存块管理信息从该组中删除。

内存块释放的流程如图 4 所示。

内存泄漏判定方法为：在很多程序中，对动态内存的申请

和释放都有一定的规律性,将内存块依据函数的调用关系来分组,可以认为同一组内存的内存块具有相似的行为,也即生存周期应该大致相同。如果有一个组中的内存块生存周期大大超过本组的其他内存块的生存周期,则可以认为这块内存是可能泄漏的内存,称为疑似泄漏。内存泄漏的判定,是约定当一个内存块的生存周期大于所在组中最大生命周期的两倍[4],则将该块内存标记为疑似泄漏,然后再对该内存块进行是否有读写操作的判定,如果该内存块在一段时间内都没有写操作,就认为该内存块可能是泄漏的内存,将该内存块报告为泄漏的内存块,类型为疑似泄漏。

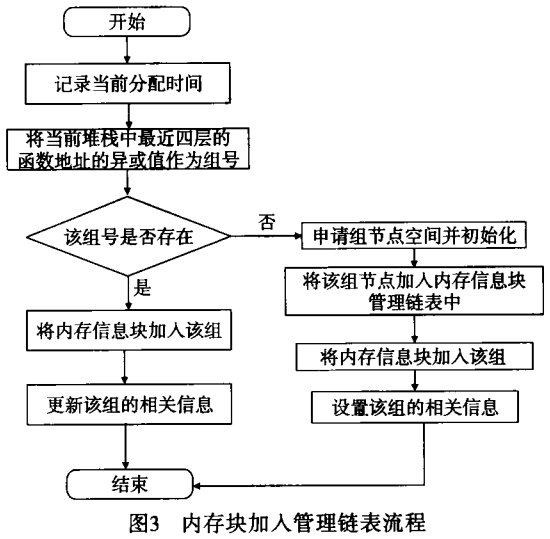

图3　内存块加入管理链表流程

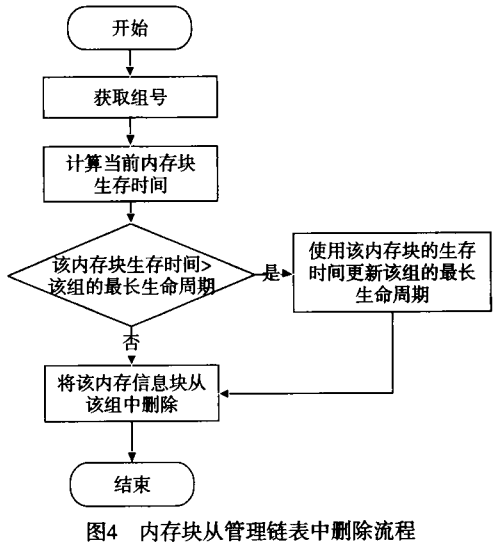

图4　内存块从管理链表中删除流程

程序结束以后检测各个函数族中是否还有没有释放的内存信息块,如果有则表示存在内存泄漏,报告内存泄漏错误,内存泄漏类型是确定泄漏。

2　MemGuard 工具的实验验证

本章给出了新的内存检测工具 MemGuard 与开源的内存检测工具 Memcheck 在功能和性能方面进行对比测试,并对测试结果进行分析比较。

Memcheck 是运行在 Valgrind 上用来检测内存相关错误的一个重量级的工具,是 Valgrind 使用最为广泛的一个工具,也是 Valgrind 指定的默认工具,它能够发现大量用 C、C ++ 编写的程序中存在的内存错误问题,所以本文采用 Memcheck 与 MemGuard 进行比较。

2.1　工具设计思路和系统框架

1)测试环境

硬件环境:Intel i386 平台。

软件环境:CentOS 5.5;

gcc、g ++ 4.1.2。

2)被测程序

为了能够演示对 C 标准内存管理函数族的支持,在 C 代码中,本文人工地模拟了内存泄漏、内存写越界、多重释放等错误,在被测程序中采用了三种方式来申请内存块,都采用 free 来进行释放。同时将程序代码放在两个不同文件中,是为了演示工具能支持由多个文件编译成的应用程序进行检测,同时还能得到文件名和行号信息。

对于 C ++ 函数族,本文着重对函数的不匹配释放的验证。在 C ++ 函数族中分析了 operator new[ ]、operator new 以及 operator delete[ ]、operator delete 的区别,知道对于简单数据类型和复杂数据类型这些函数的处理有所不同,因此在构造测试代码时,构造了 delete 释放由 new[ ] 申请的基本、复杂数据类型以及 delete[ ] 释放由 new 申请的基本、复杂数据类型四种不匹配错误。

3)实验结果及总结

通过对 Memcheck 和 MemGuard 的测试实验发现,两个工具都能够检测出程序中出现的内存错误。对于多重释放错误,MemGuard 不仅报出了多重释放的错误,同时还提供了第一次、第二次释放时释放函数所在的文件名及行号。对于内存的写越界,MemGuard 和 Memcheck 都能报出发生写越界的具体位置。对于内存泄漏错误,两个工具都能报告发生错误的地方,并同时提供最近几层的函数调用关系,程序员可以通过调用关系找寻到发生内存泄漏时的调用路径,从而分析该块内存泄漏的原因并排除该问题。因此 MemGurad 可以完全地支持对 C 标准内存管理函数的支持,并且能够提供十分精确的定位信息。在检测结果中可以看到对于这四种不同情境的不匹配释放错误,本文的 MemGuard 都可以准确地捕捉到。同时也模拟了 new[ ]、new 正常申请内存块,delete[ ]、delete 正常释放内存块。

2.2　MemGurad 与 Memcheck 时间性、空间性能对比实验

2.2.1　检测用户程序

此处选取的程序为包含内存管理函数 malloc 和 free 的普通程序,为了能够测试出采用工具运行后的执行时间的关系,在被测程序中加入不同数量级的循环。采用 Linux 自带的命令 time 来统计程序运行时间,表 1 的实验数据都是运行 20 次以后取平均值的结果。

表 1　用户程序时间性能测试结果

被测程序	原程序运行	MemGuard (JIT)	MemGuard (Probe)	Memcheck	备注
test1	0.021	1.648	0.390	0.627	程序中无循环,只有简单语句
test2	0.021	1.638	0.406	0.596	程序中 100 次级别循环
test3	0.023	1.680	0.429	0.604	程序中 1 000 次级别循环
test4	0.080	2.106	0.492	0.700	程序中 10 000 次级别循环
test5	0.739	5.530	1.145	1.614	程序中 100 000 次级别循环
bigadd	0.020	2.180	0.424	0.631	代码量:400 行大数相加程序

普通用户程序时间性能、空间占用测试对比如图5、6所示。

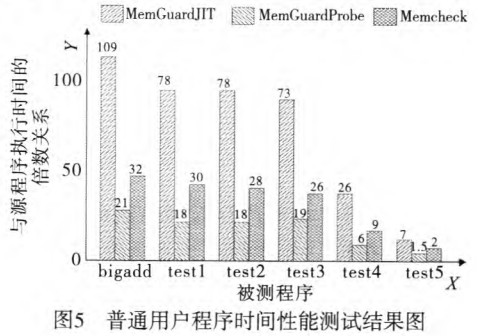

图5 普通用户程序时间性能测试结果图

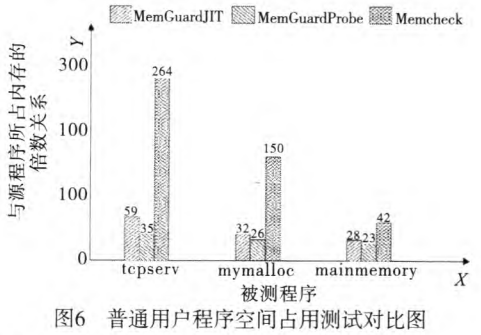

图6 普通用户程序空间占用测试对比图

2.2.2 采用 Linux 自带命令作为被测代码

为了测试本文系统对实际应用系统的检测效果，此处采用 Linux 命令程序作为被测程序。由于只是对时间性能进行测试，所以被测程序中的逻辑关系影响不大。本文采用 Linux 自带的命令 time 来统计程序运行时间。为了有可比性，在 Valgrind 中只开启工具 Memcheck 进行内存检测，时间测试结果如表 2 所示。以下实验数据都是运行 20 次以后取平均值的结果。

表 2 Linux 系统程序时间性能测试结果

被测程序	原程序运行	MemGuard (JIT)	MemGuard (Probe)	Mem check	备注
gcc	0.294	5.280	0.949	1.879	命令：gcc -o bigadd -g bigadd.c malloc 次数：343 次
ls	0.038	3.540	0.583	1.161	命令：ls -la malloc 次数：288 次
tar	0.348	6.758	1.397	1.915	命令：tar-zxvf pin-jit-runtime.tar.gz . malloc 次数：296 次
chmod	0.027	2.502	0.438	0.726	命令：chmod 755 MemGuard malloc 次数：37 次
ldd	0.087	10.104	0.989	2.372	命令：ldd bigadd malloc 次数：4 201 次
cat	0.027	2.140	0.454	0.667	命令：cat pin.log malloc 次数：31 次
tree	0.031	2.692	0.487	0.779	命令：tree malloc 次数：328 次
ps	0.055	5.156	0.612	1.435	命令：ps-aux malloc 次数：519 次
ifconfig	0.095	3.592	0.680	0.949	命令：ifconfig eth0 192.168.1.234 netmask 255.255.255.0 malloc 次数：152 次
route	0.156	2.113	0.336	0.866	命令：route malloc 次数：179 次

2.2.3 实验结果分析

a）从测试结果来看无论是采用 MemGuard 进行测试还是采用 Memcheck 进行测试，都会有比较高的负载，这是由于采用动态二进制插桩方式固有的缺陷。因为需要在程序运行过程中进行分析指令，并同时用新的指令替换原有的指令，会造成比较大的性能开销。

b）如图 7、8 的结果说明 Probe 模式下的 MemGuard 比 Memcheck 有更低的负载，Memcheck 的运行效率要比 Probe 模式下的 MemGuard 慢 2 ~ 3 倍，但是 JIT 模式下的 MemGuard 的负载明显高于 Memcheck 和 Probe 模式下的 MemGuard。对于一些实时性要求不高的程序来说采用这三个工具能够检测出程序中存在的内存错误，虽然对被测程序的执行效率有一定的影响，但是牺牲一点性能而查出程序中存在的致命错误还是有价值的。

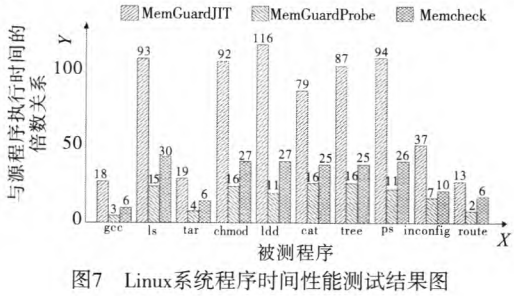

图7 Linux系统程序时间性能测试结果图

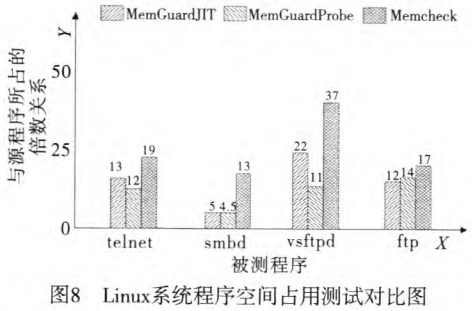

图8 Linux系统程序空间占用测试对比图

c）在对内存使用率上，Memcheck 会占用更多的内存，Memcheck 占用更多的内存跟它的内存检测原理有关，因为 Memcheck 会将进程的整个地址空间都进行一个映像，对进程中的每一个字节和 CPU 中的每个寄存器，Memcheck 都有一个 8 bit 的空间与之对应，用来记录该字节或者寄存器中的值是否有效；同时还有 1 bit 用来记录该地址是否能被写。而在 Pin 上开发出来的 MemGuard，内存方面的开销主要是 Pin 本身的开销，Pin 中开销的部分主要来自它的代码缓存，用来存储转换后的指令。因此 MemGuard 在空间上的开销是要比 Memcheck 小很多。

3 相关工作

3.1 静态检测工具

程序静态分析方法并不编译运行程序，而是通过扫描源代码或者目标代码，采用软件分析方法中的静态代码分析技术对程序进行分析。很多静态检测工具不仅能够分析出程序中可能存在的编码规范错误、典型的语句错误，甚至可以发现程序动态运行中的错误。

静态代码检测工具 PC-Lint[5] 能发现程序中不合理的编程习惯、程序的语法错误以及潜在的程序错误隐患。Polyspace[6] 能够检测程序运行错误，它采用最新的语义分析技术，以大量由数学定理提供的规则为依据，对软件的行为进行动态分析，能在编码阶段就发现程序中可能存在的运行时错误。QAC/C++ 能够自动地检查出 C、C++ 中出现的违反编程准则和标准的错误，能够在软件开发阶段避免代码中的问题。静态的代码检测工具对内存问题的检测比较有限，内存很多错误通常只能在运行过程中暴露出来，因此静态工具检测出来的内存错误结果的准确性相对较低。但是静态代码检测工具对程序的其他

方面检测的效果还是很理想,比如编码规范、一些语法上看似正确但却可能存在的错误、一些内存的申请和释放函数的不配对等。

3.2 动态检测工具

动态检测一般使用程序插桩技术[7]完成。其优点是可以监控任何想得到的信息,缺点是给系统运行带来的开销很大,从几倍到几百倍的开销都有。

3.2.1 基于源代码插桩方法

在程序源代码中插入必要的监控和处理语句,由于其实现相对简单,因此这是最为广泛使用的一种方法。典型代表是Insure++、mtrace等。

文献[8]是实现较为简单的、开销相对也较小的实现动态内存泄漏检测的研究,实现技术有一定差别,但基本上都是针对内存操作函数进行插桩,有的只检测malloc/free函数,还有的可以检测calloc/new/delete等函数,通过在程序中有malloc/free/new/delete等函数调用的地方加相应的语句完成监控。文献[9]用类似于源代码插装的方法实现内存泄漏检测,其中讨论了两种检测方式,即重载new和用宏替换。作者结论是重载new的方式要简单一些。文献[10]也是对源程序进行插桩,以便跟踪内存块的申请、使用、释放等操作。为了降低时间和空间开销,在程序运行过程中,使用了程序取样技术,即对程序取样监控:在运行过程中,程序的运行在原始版本和插桩版本之间切换,这样可以降低运行时间。其时间开销小于5%,空间开销小于10%。文献[11]用data sampling技术,通过插桩跟踪内存中一组内存对象不活跃的时间长度,确定泄漏。从内存管理角度进行分析,将基本相似的(分配时间相近、活跃时间相近等)内存对象放在相同的页面上,一个页代表其中的所有对象。

3.2.2 基于目标代码插桩方法

典型代表是Purify[12],采用了目标代码插桩技术OCI,它采用的就是在程序的目标代码中插入特殊的指令并跟踪程序的内存使用状态,采用此种技术不会对其源代码造成影响,只需要对修改后的目标代码进行重新链接就可以运行,使用非常方便,功能也很强大。Purify采用类似于垃圾回收机制的mark-and-sweep算法,即所谓的标记—清除算法。在标记阶段,它会跟踪所有从栈区和数据区到堆空间的指针,并同时标记它们引用的内存对象;在清除阶段,它会报告哪些程序不再访问到的内存对象。

3.2.3 基于动态二进制代码插桩方法

典型代表是Valgrind[13]。另外还有开发本文MemGuard工具的动态二进制插桩工具Pin。文献[14]也是用动态插桩技术实现的,基于DIOTA(dynamic instrumentation,optimization and transformation of applications)。使用引用计数法实现内存泄漏检测,可以精确地指出对某个内存的引用是在哪里丢失的。但时间开销非常大,增加了200~300倍。文献[15]针对动态插桩工具提出了离线分析的方式来优化此类工具性能。

4 结束语

本文在动态二进制插桩框架Pin的基础上设计实现了一个内存检测工具MemGuard。该工具适用于C/C++的内存错误检测,使用方便。对于被测的程序,不需要对源代码进行任何修改,也不需要进行重新编译就能直接运行,并能够给出内存错误报告,对于出现错误的地方能够准确定位文件名和行号;对于内存泄漏错误还能定位出分配该块内存的函数调用链,对于函数调用链中的每一个函数同样也能定位该函数所在的文件名和行号。

本文提出了在内存工具管理中采用基于函数族的内存管理方式来管理被测程序中不同类型的函数分配的内存信息块,并且能够支持用户自定义内存管理函数。对于一个新的内存管理函数族,只需要提供函数原型以及每个参数的意义就可以很方便地在该工具中进行监控管理,可扩展性很好。

内存检测工具MemGuard暂时只能支持Linux平台,不支持Windows平台,下一步的工作就是要将该工具多平台化,使其能够同时支持Linux、Windows等平台。由于该工具是基于动态插桩原理,所以带来了额外的开销,如何减少额外开销是下一步要改进的。另外现在嵌入式领域的内存检测工具非常缺乏,很多内存检测工具都只是对PC机的支持,对嵌入式开发平台都不支持。如何在资源非常有限的嵌入式系统,在不影响程序的性能的情况下检测出内存错误,这是将来需要研究的方面。

参考文献:

[1] Luk C K, Cohn R, Muth R, *et al*. Pin: building customized program analysis tools with dynamic instrumentation[J]. ACM Sigplan Notices, 2005, 40(6): 190-200.

[2] Back M, Charney M, Cohn R, *et al*. Analyzing parallel programs with Pin[J]. Computer, 2010, 43(3): 34-41.

[3] Hzelwood K, Klauser A. A dynamic binary instrumentation engine for the ARM architecture[C]//Proc of International Conference on Compilers, Architecture and Synthesis for Embedded Systems. New York: ACM Press, 2006: 261-270.

[4] Qin Feng, Lu Shan, Zhou Yuanyuan. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs[C]//Proc of the 19th International Symposium on High-Performance Computer Architecture. [S. l.]: IEEE Press, 2005: 291-302.

[5] Deissenboeck F, Juergens E, Hummel B, *et al*. Tool support for continuous quality control[J]. IEEE Software, 2008, 25(5): 60-67.

[6] Munier P. Polyspace®[M]//Industrial Use of Formal Methods: Formal Verification. Hoboken: Wiley, 2012: 123-153.

[7] Zhao Jianjun. Applying slicing technique to software architectures [C]//Proc of the 4th IEEE International Conference on Engineering of Complex Computer Systems. [S. l.]: IEEE Press, 1998: 87-98.

[8] Steffen J L. Adding run-time checking to the portable C compiler[J]. Software: Practice and Experience, 1992, 22(4): 305-316.

[9] Erickson C. Programmatic memory leak detection in C++[EB/OL]. (2003-06-01)[2009]. http://www.linuxjournal.com/article/6556.

[10] Hauswirth M, Chilimbi T M. Low-overhead memory leak detection using adaptive statistical profiling[J]. ACM SIGPLAN Notices, 2004, 39(11): 156-164.

[11] Novark G, Berger E D, Zorn B G. Efficiently and precisely locating memory leaks and bloat[J]. ACM Sigplan Notices, 2009, 44(6): 397-407.

[12] Dai Ziying, Mao Xiaoguang. Light-weight resource leak testing based on finalisers[J]. IET Software, 2013, 7(6): 308-316.

[13] Chahar C, Chauhan V S, Das M L. Code analysis for software and system security using open source tools[J]. Information Security Journal: a Global Perspective, 2012, 21(6): 346-352.

[14] Maebe J, Ronsse M, De Bosschere K. Precise detection of memory leaks[EB/OL]. (2004-04-27). http://www.cs.virginia.edu/woda/papers/maebe.pdf.

[15] 代声馨,洪玫,郭鑫宇,等.基于动态插桩的程序分析工具的性能改进[J].计算机应用研究,2013,30(7):2087-2090.