第39卷第2期 中国科学技术大学学报 Vol. 39, No. 2
2009年2月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Feb. 2009

文章编号:0253-2778(2009)02-0189-07

一种针对可执行代码的内存泄漏静态分析方案

龚育昌,胡 燕,张 晔,赵振西

(中国科学技术大学计算机科学技术系,安徽合肥 230027)

摘要:针对应用程序安全分析的实际需求,设计并实现了一个针对可执行代码的内存泄漏分析框架MLAB. MLAB首先从可执行代码中恢复控制流和数据流信息,依据恢复的控制流图建立程序的有限状态自动机,在此基础上运用模型检测算法分析程序可能存在的内存泄漏. 利用几个典型的程序实例详细说明了MLAB方法的工作原理,并通过基于测试程序集MiBench的实验对方法进行了验证,结果说明了该方法的有效性.

关键词:可执行代码;模型检测;内存泄漏

中图分类号:TP371 **文献标识码**:A

A static memory leak detection method for binary programs

GONG Yu-chang, HU Yan, ZHANG Ye, ZHAO Zhen-Xi

(*Dept. of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: In order to conduct memory leak analysis upon executable programs, a binary-level memory leak analysis framework MLAB is designed and implemented. First, MLAB recovers control flow and data flow information from binary programs. Then a control flow automata is constructed based on the recovered control flow graph. After that, a model checking based algorithm is applied upon the automata to perform memory leak analysis. Several sample programs are further analyzed to show in detail how the method works. Experimental results on the benchmark suite MiBench shows the effectiveness of the method.

Key words: binary code; model checking; memory leak

0 引言

用C/C++等支持显式内存分配的语言编写的程序,内存泄漏是一个严重的问题. 存在内存泄漏问题的应用程序在运行时会消耗过多的系统内存,降低系统的性能.

内存泄漏问题比较难于检测,因为它的唯一征兆就是在程序运行过程中,程序消耗的内存不断增加,而且内存消耗的增加也不一定是由内存泄漏引起的. 内存泄漏对长时间连续运行的服务器端应用程序以及嵌入式应用程序的影响甚大,因此研究如何做好有效的检测和消除程序中的内存泄漏是十分重要的工作.

内存泄漏检测方法分为动态分析和静态分析两大类. 文献[1,2]描述了检测内存泄漏的动态分析方法,这类方法利用程序运行时的信息,分析源代码中导致泄漏的内存操作位置,并据此推测导致内存泄漏的程序路径. 文献[3,4]则对内存泄漏的静态分析

收稿日期:2007-06-20;**修回日期**:2007-09-21

基金项目:教育部高等学校博士学科点专项科研基金(20050358040)资助.

作者简介:龚育昌(通讯作者),教授. E-mail: ycgong@ustc.edu.cn

方法进行了研究,提出了利用程序的静态语义信息判断程序中是否存在内存泄漏的基本方法.一种常用的静态分析方法是使用类型信息约束程序中内存操作的行为[5],Lint 工具[6,7]就是基于类型分析的内存行为分析工具.

目前,内存泄漏检测方法都或多或少地依赖于源程序中的结构信息,但是在经过各编译阶段后,程序的结构往往与源程序中的结构有很大的差异.在没有对编译器进行验证的情况下,编译器产生的符号信息以及类型信息的可信度也是一个问题.另外,对于无法获取源代码的第三方不可信程序[8~10]的分析,必须针对二进制代码进行.因此对目标机器代码的直接属性分析方法的研究具有重要的价值和意义.

分析二进制程序比针对源语言程序的分析更困难.与源程序不同,二进制程序中缺乏显式的控制流信息.目前,针对二进制代码的分析方法多数是基于二进制代码中的一些特征代码模式进行[8,10],这类分析只需对程序中特定的代码模式进行匹配分析,分析过程不需要太多的程序结构信息.

由于内存泄漏分析必须基于程序中的控制流信息和数据流信息进行,本文针对二进制程序的内存泄漏问题,设计并实现了对二进制可执行文件的内存泄漏检测进行分析的框架方案(memory leak analysis for binaries,MLAB).MLAB 将二进制代码恢复为与机器无关的编译中间表示形式,同时恢复程序中的控制流和数据流信息,并基于此运用模型检测算法[11]进行内存泄漏分析.MLAB 不需要借助任何关于源程序的符号信息,就可以完成对二进制程序的分析,具有重要的应用价值.文中详细介绍了 MLAB 的工作流程和原理,并利用具有代表性的 Benchmark 程序说明了方法的可操作性与实用性.

1 MLAB 分析框架

1.1 MLAB 的工作流程

MLAB 的基本工作流程如图 1 所示.

整个分析流程包括如下几个主要步骤:

(Ⅰ)加载二进制程序的映像到内存.这个过程需要解决二进制程序中的内部偏移地址到内存影响的起始地址的重新定位问题;

(Ⅱ)对二进制程序的内存映像进行解码的过程.将机器语言指令转换为汇编指令格式;

(Ⅲ)控制流恢复过程.包括程序的过程调用图(call graph)和程序的控制流图(control flow graph,CFG)的恢复过程.经过这一过程,将与目标机器相关的汇编指令形式转换为与目标机器无关的中间表示形式;

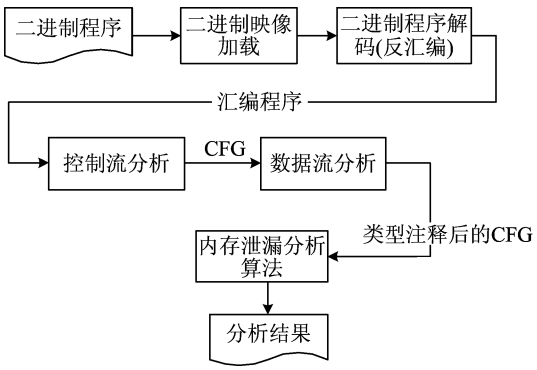

图 1 MLAB 工作流程

Fig. 1 MLAB workflow

(Ⅳ)数据流分析.这个阶段进行数据流信息的恢复,包括变量和变量的类型等信息;

(Ⅴ)内存泄漏分析的核心模块.这个阶段进行内存泄漏分析,过程包括依据程序控制流图建立程序的控制流自动机,并基于控制流自动机以及程序的数据流信息,分析程序中的内存泄漏问题.

1.2 二进制程序的映像加载与指令解码

程序分析的第一个步骤是加载二进制程序到内存,并对加载到内存的程序进行解码,恢复其汇编指令格式(反汇编).

反汇编的模式一般有两种:从程序中第一条指令开始和从 main 函数的入口开始解码的过程.解码过程的作用类似于编译程序中的词法分析器,仅有的差别就在于解码过程提供的是汇编指令,而不是高级语言的语法符号.

MLAB 中采用从 main 函数的第一条指令开始解码的方式,这样便于正确地恢复程序的控制流结构,但是前提是首先能找到 main 函数的入口地址.

1.3 程序的控制流和数据流恢复

控制流分析按照以下两个步骤进行:首先分析程序中的调用结构,恢复程序的过程调用图,建立程序流图的宏观结构;随后,对每个函数的内部代码进行控制流分析,构建其中的每个基本块和基本块之间的控制流关系,形成程序的控制流图.

程序的函数调用图恢复过程的步骤如下:

(Ⅰ)分析 main 函数的入口.在找到 main 函数的入口后,开始界定 main 函数的函数体的范围,找到 RET 语句作为最后的边界;

(Ⅱ) 从 main 函数开始,通过函数调用语句分析程序中函数的其他函数调用.

函数边界的识别借助于操作系统支持的二进制应用程序接口(application binary interface,ABI)得以完成. ABI 规定了编译生成的二进制代码所满足的函数调用规范. 分析工具依据 ABI 提供的信息,识别出二进制程序中的函数调用以及返回语句,从而界定函数的第一条和最后一条指令.

在划分出程序的函数调用结构之后,利用下面的算法分析每个函数中的控制流.

算法 1.1　控制流恢复算法

```
01  CurBB←null
02  while(Inst=GetNextInst()){//逐条汇编指令分析
03    if(Curb==null)
04      CurBB=new std::list<Instr*>();
05    switch(Inst→kind()){
06      case STMT_goto:
07        将指令 Inst 加入到当前的基本块 CurBB 中
08        将已经构建好的 CurBB 加入到 pCfg 中
09        建立 CurBB 到 Inst.target 控制流边
10        CurBB←null
11        break;
12      case STMT_RET:
13        将 Inst 添加到 CurBB 中
14        将已经构建好的 CurBB 加入到 pCfg 中
15        CurBB←null
16        break;
17      default:
18        将 Inst 添加到 CurBB 中
19        break;
20    }
21  }//while
```

算法中的 CurBB 是当前处理的基本块. 根据不同的汇编指令进行不同的处理. 在遇到 STMT_goto 时,将 goto 指令添加到 CurBB 中后,完成 CurBB 的构建,再进行新的基本块构建. STMT_RET,将 return 指令添加到 CurBB,完成整个函数的控制流图的构建. 对于普通的指令,则添加到 CurBB 中.

对程序的进一步属性分析将依赖于基于控制流图的数据流分析的结果.

2　类型分析与 PLL

2.1　类型恢复过程

变量的类型分析是对二进制程序的内存泄漏分析的重要步骤. 类型分析依赖于对汇编程序中寄存器活跃区间的分析. 在从二进制程序中恢复出来的控制流图中,变量都是以寄存器的形式表示的. 寄存器的不同活跃区间用于存放不同变量的值. 类型恢复的过程基于对汇编程序中寄存器的活跃信息进行. 通过数据流分析,建立活跃信息,分析每个汇编语句的定值到达信息.

在类型恢复过程中,首先为每个寄存器区间标注可能的类型信息.

变量的类型依据寄存器在控制流图中的使用信息进行推断. 对类型进行推断的依据是其在一个寄存器活跃区间内的使用情况.

将汇编程序中的整型变量与表示地址的变量类型区分开是类型分析中需要解决的核心难题. 例如,对于汇编指令 mov eax, 71167,人们无法判断 71167 是整型值,还是地址.

在寄存器的类型分析过程中,寄存器中的数据类型被默认为是整型的数据. 类型恢复算法的主要步骤如下:

(Ⅰ) 在解码阶段,搜集关于变量的可用类型信息;

(Ⅱ) 通过类型推断的方法,将已有的类型信息传播到程序的不同部分. 类型推断所基于的类型间的格的关系如图 2 所示.

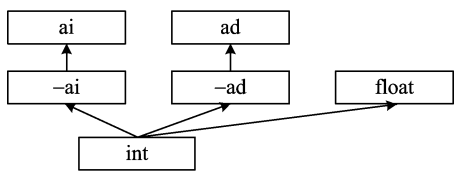

图 2　类型格

Fig. 2　Type lattice

类型信息在过程间的传播需要更多的处理. 对于程序中实现的函数接口,就必须恢复其参数类型以及返回值类型. 程序中使用的外部库函数,需要根据原型描述推断其类型. 图 3 给出了类型分析过程中使用的类型推断规则.

在解码阶段,将可能是 Addr of Inst 的数据标注为_ai, 将可能是 Addr of Data 的数据标注为_ad 类型.

(T-AddrInst)描述的是在代码段地址范围内的所有数值都有可能是指令地址. 在数据段地址范围内的寄存器数值都被标为类型_ad,表示可能是指向数据的指针(T-AddrData). 跳转目标寄存器也被视

$$\frac{r=num, num\in AddrRange(TextSegments)}{type(r)=\_ai} \quad (T\text{-}AddrInst)$$

$$\frac{r=num, num\in AddRange(DataSegments)}{type(r)=\_ai} \quad (T\text{-}AddrData)$$

$$\frac{JMP\ r, type(r)=int}{type(r)=\_ai} \quad (T\text{-}AddrInst2)$$

$$\frac{r=exp, type(r)=int, type(exp)=A}{type(r)=A} \quad (T\text{-}ASSGN)$$

$$\frac{r=call\ f, ReturnType(f)=A}{type(r)=A} \quad (T\text{-}call)$$

$$\frac{r=call\ LibFunc}{type(r)=A} \quad (T\text{-}LIB)$$

$$\frac{call\ r, type(r)=\_ai}{type(r)=ai} \quad (T\text{-}AddrInst3)$$

图 3 类型推断规则

Fig. 3 Type inference rules

为_ad(T-AddrInst2). 赋值语句的类型推断为(T-ASSGN). 函数调用的类型推断为(T-call). 对外部库函数的类型推断为(T-LIB). 对调用的函数入口地址的类型的推断为(T-AddrInst3).

在一个过程执行期间,每个寄存器都可能存放多个变量的值,因此对寄存器的活跃区间的分析极为重要.

2.2 内存操作分析与 PLL

图 1 使用 TMC 算法的作用对象是潜在泄漏位置(potential leak location,PLL),PLL 的定义如下.

定义 2.1 潜在的泄漏位置 PLL 表示为 pll=(pc, op, f),其中,op 是函数 f 中控制位置是 pc 处的程序语句,并且 op 满足条件 isPtrAssign(op) or isReturn(op).

PLL 描述了程序中可能导致内存泄漏的代码. PLL 的定义可以分为两类:一类是指针变量之间的赋值语句(isPtrAssign),另一类是程序的返回语句(isReturn).

内存泄漏的一种形态是当一个指针变量 p 是指向内存区域 Mem 的唯一指针时,对 p 的赋值将导致内存泄漏. 另一种情况是在函数返回时,如果当前函数中存在动态内存分配 p=malloc(),且 p 是一个局部变量;如果 p 没有被作为参数返回,并且没有赋给其他的任意变量,那么由于 p 的生命周期在函数返回后终止,从而导致了内存泄漏的发生.

每个 RTL 程序都可以被视为一系列的赋值语句,而由一些分支语句完成程序执行过程中的控制转移. 每个赋值表达式可以表示为 PC: lhs=rhs. 其中,PC 表示该赋值语句所处于的程序控制位置,lhs 表示表达式的左边,rhs 则表示表达式的右边部分.

程序中的 PLL 集合通过类型分析方法构建. 程序返回语句对应的 PLL 的构建相对容易,仅需要记录这些语句即可. 指针赋值类 PLL 的构建,则必须首先维护程序中的所有指针变量的信息,基于对程序的类型分析,记录程序中所有指针类型的变量. 在分析程序中所有赋值语句的过程中,记录那些赋值表达式左部是指针类型的所有赋值语句,这些语句的位置就是指针赋值类 PLL.

3 内存泄漏分析算法

MLAB 框架中所采用的内存泄漏分析算法(TMC 算法)是对基于谓词抽象的模型检测算法的拓展. 下面从算法输入集的构造、基本算法的实现、对分析方法的扩展等主要方面,来说明分析算法的结构.

3.1 算法输入集的构造

为了对内存泄漏进行分析,利用 2.2 节中描述的基于类型的方法得到 PLL 集合,该 PLL 集合作为 TMC 算法的输入.

TMC 算法首先对获得的 PLL 集合进行预处理. 在预处理过程中,基于对程序的数据流分析,排除一些不可能导致内存泄漏的 PLL. 比如,在可以通过数据流分析确定 p=null 的情况下,PLL=(pc, p=q, f) 则不可能导致泄漏. 因为这种情况下 p 不指向任何动态分配的内存区域,将 p 的指向调整为 q 所指向的内存区域是安全的.

对两种类型的 PLL,预处理方法也有所差别. 对两种情况分别进行如下描述:

(Ⅰ) return 语句表示的 PLL. 该类型的 PLL 是否导致内存泄漏,依赖于其中是否调用 malloc 函数进行了堆上内存的分配. 如果 return 语句所从属的函数中没有对堆上内存分配函数的调用,那么该 return 语句对应的 PLL 就不会造成内存泄漏. 例如算法 2.1 中的示例程序第 12 行就是一个可以认定为安全的 PLL,这是因为 main 函数中没有对 malloc 函数的调用;

(Ⅱ) 指针赋值语句对应的 PLL. 如果确定赋值语句的目标指针指向的内存块在相应的程序控制点被多个指针指向,则可以确定相应的 PLL 中对指针变量的赋值不会导致内存泄漏.

对安全的 PLL 的判定方法,可以从 PLL 集合中排除这些安全的 PLL,将经过缩减后的 PLL 集合作为算法主体的处理对象.

3.2 TMC 算法主体部分

TMC 算法对经过预处理的 PLL 集合 pll_set 中的每一个 PLL，运用模型检测方法逐个进行分析. 算法的目的是通过进一步分析，从 pll_set 中排除可以静态确定为安全的 PLL. 算法形式如下：

```
ForEach pll in pll_set do
  if(ModelCheck(pll)= NotSafe)
    leak_list. add (pll)
ForEach leak in leak_list
  print_error_path (leak)
```

ModelCheck(pll)的目标是分析 PLL 在程序抽象后的自动机中的可达性. 如果经过分析仍然无法判定 PLL 是否安全，该 PLL 就会被添加到结果的集合 leak_list 中. 与 PLL 同时加入到 leak_list 中的还有与之相对应的错误路径，即从可达树的根结点到 PLL 的路径，用于辅助程序员对程序错误的来源进行分析.

最终输出的与 leak_list 中的 PLL 相关联的错误路径，可以作为对程序中可能存在的内存泄漏的分析与测试依据.

对程序进行合理的抽象是 ModelCheck 方法实现的基础. 基于对程序的 CFA 抽象，构建可达树的过程采取的是对基于程序的过程调用图和 CFA 构成的状态转换图的深度优先遍历过程.

在构建可达树的过程中，每当到达一个 err_loc 的时候，ModelCheck 方法会暂停可达树的构建，转而分析从可达树的根到 err_loc 的路径 err_path. 通过对 err_path 上的谓词状态条件的分析结果，判定该路径是否确实导致了内存泄漏问题的发生.

可达树的路径表示为 $p=(S_0, op_1, S_1, \cdots, op_n, S_n)$. 其中，$S_i$ 是程序的状态，op_i 则表示程序中的一个操作. 路径上的操作序列决定了程序的状态转换过程.

上述基本分析方法在路径谓词的分析中需要对反例的路径进行完全分析，产生大量的谓词. 过多的谓词会大大增加算法需要检测的状态数，从而降低程序的执行效率. 针对内存泄漏这一实际问题，本文提出了基于逆向路径分析路径中潜在的内存泄漏的方法.

3.3 针对错误路径的逆向分析

为了有效地进行内存泄漏分析，本文对 ModelCheck 方法实现中的基于执行路径的反例的分析方法进行扩充，提出针对内存泄漏问题的逆向路径分析方法.

用逆向分析过程推断路径中与内存相关的操作对程序内存状态的影响，并使用推断的结果判断当前的 PLL 是否安全.

在内存状态的描述中，每个程序含有一组内存块 $M=\{m_1, m_2, \cdots, m_k\}$，其中 $m_i(i\in[1,k])$ 表示的是程序中在堆上分配的一段内存. 基于库函数 malloc 的语义可以推知 M 中的内存块之间是独立无重叠的. 每个内存块中都维护着指向该内存块的指针变量.

在逆向分析过程中，针对不同类型的 op，对内存状态进行相应的更新. 逆向分析过程开始时，路径上的内存状态为空. 在逆向分析过程中，每碰到一个与内存相关的操作，都要分析其对内存状态的可能影响，并对内存状态进行相应的更新. 对每一个 PLL，通过逆向分析对其内存状态的推断过程，目标是记录所有指针变量的最后一次赋值. 逆向分析是基于单一赋值形式(SSA)进行的. SSA 形式的变量单一赋值的特点简化了程序中变量的定值-引用链，使得程序的数据流分析得以简化. 对逆向分析中指针赋值语句的分析而言，就意味着对一个特定的指针变量的定值仅有一次，因此其对内存状态的影响能够直观地表达出来.

对逆向路径的分析方法针对几类不同类型的程序语句进行分析，算法流程如下.

算法 3.1　逆向分析算法

```
M←Φ
ForEach of in r_path
  if 操作是指针赋值形式 p=q Then
    M←M∪{mp}
    mp 的指针集合初始化为{p,q}
    if PLL. op 是 p=q 的形式 then
      if mp 的别名集合的元数数目>1 then
        return leak_false
  ELIf 操作 op 的形式是 p=malloc(n) then
    M←M∪{mp}
    将 mp 的指针集合设置为 p 的所有别名
    if PLL. op 是 p=q 的形式 then
      if mp 指向的指针中只有一个元素 then
        return leak_true;
  ELIf 操作 op 的形式是 free(p) then
    将 p 添加到内存环境 M 的 null 指针集合
    if PLL. op 是 p=q 的形式 then
      return leak_false
  ELIf 操作 op 是函数调用 call(f) then
    M(f)=在函数 f 中分配的所有内存
```

```
if PLL.op是一个return语句 then
  if M(f)中的元素都由f的局部指针变量指向 then
    return leak_true
  else
    return leak_false
```

分析过程中,当 PLL 可以根据当前推断得到的内存状态判断为安全时,逆向路径分析算法返回 leak_false,否则返回 leak_true. 分析算法对逆向分析过程中碰到的每类语句的处理方法如下所示:

(Ⅰ) 指针赋值语句 $p=q$

逆向路径中存在这样的赋值语句意味着在这条语句执行后,p 和 q 指向同一个内存区域 m_1, 也就是说 m_1=q.mem=p.mem. 依据这个状态可以更新内存的状态 M,使得 m_1 的指针集合中添加 p,q 两个指针指向. 检查更新后的内存状态 M',如果可以判断内存状态能够保证 PLL 的指针赋值语句的目标操作数不是唯一指向相应内存位置的指针,那么就可以确定 PLL 的安全性. 对于返回语句对应的 PLL,则需要记录最近一次对内存变量的使用.

(Ⅱ) 对 malloc 函数的调用

到达 PLL 的路径形式为 p=malloc(..); …; $p=q$;

如果在逆向分析抵达这一赋值语句的时候,内存状态中指向 p 的所有指针仍旧只能判定为 1 个,那么程序认为这是一个内存泄漏错误,可以直接给出警告.

(Ⅲ) 对 free 函数的调用

到达 PLL 的路径形式为 free(p); …; $p=q$;

用于释放 p 所指向的内存的方法,其效果可以视为指针赋值语句 p=null. 如果 free(p)是到 PLL 的路径上的最后一次对 p 的值的修改,那么该 PLL($p=q$)是不会导致内存泄漏错误的. 因此逆向分析的路程可以返回一个 ok,表明当前的路径上是不可能产生内存泄漏的.

4 实例分析与实验

4.1 程序实例

本文采用的程序实例代表了两种典型的内存泄漏类型. $leak_1$ 表现的是在指向内存块 M 的局部指针变量在退出其作用域之前没有被释放而导致 M 被泄漏的情况, 而 $leak_2$ 中给出的则是对一个指向某一内存块 M'的唯一指针被更新,导致 M'被泄漏.

$leak_1$ 经过解码后的汇编程序形式如图 4 所示.

```
f:
  pushl %ebp
  movl %esp,%ebp
  pushl $50
  call malloc
  popl %edx
  xorl %edx,%edx
  test1 %eax,%eax
  je .L2
  movb $99,10(%eax)
  movb $1,%dl
.L2:
  movl %edx,%eax
  leave
  ret

main:
  pushl %ebp
  movl %esp,%ebp
  subl $8,%esp
  andl $-16,%esp
  subl $16,%esp
.L7:
  call f
  jmp .L7
```

图 4 $leak_1$ 的反汇编结果

Fig. 4 Disassembly result of $leak_1$

在图 4 中汇编的控制流恢复过程中,首先建立程序的函数调用图. 对 f 中的 malloc 调用,

LiveIn={}

LiveOut={%eax}

这里的 LiveOut 信息说明寄存器 eax 的内容被作为返回值返回. 于是 malloc 就被确定为

%eax=call malloc <[%esp],<type>>

根据库函数 malloc 的类型信息,推断出%eax 在当前活跃周期内的类型是 char *. 函数 f 的返回指令在这里被确定为 PLL. 逆向分析过程可以得到泄漏的路径.

示例程序 $leak_2$(见图 5)描述的是在指向某段内存的指针被赋值语句更新后使得其原本指向的内存块成为泄漏的内存. $leak_2$ 中对 malloc 函数的第一次调用,其参数为 5,在%eax 中的返回值被转存到%ebx,分析所得结果为 %ebx=call malloc 5. 第二处的 malloc 函数调用的参数是 7,最后的返回值被存放到%esi 中,%esi=call malloc 7. %ebx, %esi 在当前活跃周期内的类型为 char *. %ebx=%esi 因此被判定为指针赋值语句,因此是一个 PLL.

4.2 实验分析

基于工具的原型实现,本文进行了实验. 表 1 给出了对二进制程序进行控制流分析的时间数据(测试平台:Pentium 2.0, Mandrake Linux 9.0,内核版本 2.4.20). 表 1 给出了控制流图恢复的时间,程序的类型恢复所需要的时间以及内存泄漏所需要的时间.

```
main:
  pushl %ebp
  movl %exp,%ebp
  pushl %esi
  pushl %ebx
  andl $-16,%esp
  subl $28,%esp
  pushl $5
  call malloc
  movl %eax,%ebx;;%ebx=s
  movl $7,(%esp)
  call malloc
  movl %eax,%esi;%esi=q
  movl %esi,%ebx;s=q
  testl %ebx,%ebx;s=null?
  setne %al
  leal -8(%ebp),%esp
  popl %ebx
  popl %esi
  leave
  ret
  .ident "GCC:(GNU)3.4.6 20060404(red hat 3.4.6-3)"
```

图 5　$leak_2$ 的反汇编结果

Fig. 5　Disassembly result of $leak_2$

表 1　示例程序分析结果

Tab. 1　Result of the analysis on the sample programs

实例	CFG 恢复时间/s	类型分析时间/s	TMC 分析时间/s
$leak_1$	0.32	0.05	0.11
$leak_2$	0.31	0.05	0.10

基于嵌入式测试程序集 MiBench[12] 中的部分程序(network, security 相关)进行了进一步的实验,实验结果如表 2 所示. 表 2 给出了对程序进行控制流图和数据流分析的时间、内存泄漏分析的时间. 针对目标语言的控制流分析时间是针对源语言的分析时间的 1.1～1.5 倍左右,这是由于目标语言分析需要从低级语言中恢复程序的控制流结构所致. 实验对基于类型的泄漏分析方法(TC)和基于 TMC 算法的泄漏分析方法进行了比较. 由于 TC 方法和 TMC 算法采取的都是排除安全的 PLL 方法,因此他们给出的结果都是内存泄漏错误的超集,其中包含了所有实际存在的内存泄漏错误. 结果表明,TMC 能够将错误范围限制得很小,便于对内存泄漏错误的鉴别.

表 2　针对 MiBench 的分析结果

Tab. 2　Result of the analysis on MiBench

测试程序	控制流/数据流分析时间/s		内存泄漏分析时间/s		疑似泄漏数目	
	源语言	目标语言	TC	TMC	TC	TMC
anagram	2.61	3.75	2.61	4.75	10	4
rijndael	5.14	6.68	2.06	3.43	6	2
sha	1.15	1.86	0.31	0.42	6	3
disjkstra	1.29	1.78	0.42	0.55	4	1
patricia	1.36	2.02	2.73	4.55	18	5

5　结论

本文提出了一种针对二进制程序的内存泄漏分析方法,设计并实现了相应的程序分析框架 MLAB. MLAB 无需依赖程序与源代码相关的任何信息,仅仅依据程序的二进制代码形式恢复其控制流和数据流信息. MLAB 有着良好的可扩展性,是对二进制代码进行属性分析的一种通用方法,可以依据恢复的控制流和数据流信息对常见的程序属性进行分析. MLAB 方法还可以拓展到对 Java 字节码的分析领域,在提供了对 Java 字节码的控制流和数据流结构重建的方法后,MLAB 便可以对 Java 字节码进行安全属性分析.

针对内存泄漏问题,MLAB 方法提供了对程序中的变量类型进行推断的方法,着重分析了与内存访问相关的操作和数据,并基于程序的自动机抽象,运用模型检测算法对二进制程序中的内存泄漏问题进行了分析,实验证明了 MLAB 的有效性.

参考文献(References)

[1] Jump M, McKinley K S. Cork: dynamic memory leak detection for Java[R]. Technical Report TR-06-07, Department of Computer Science, The University of Texas at Austin, Austin, TX, 2006.

[2] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation[J]. ACM Sigplan Notices, 2007, 42(6): 89-100.

[3] Heine D L, Lam M S. Static detection of leaks in polymorphic containers[C]// Proceeding of the 28th International Conference on Software Engineering. Shanghai: ACM Press, 2006:252-261.

[4] Heine D L. Static memory leak detection [D]. Department of Electrical Engineering, Stanford University, 2004.

[5] Regehr J, Cooprider N, Archer W, et al. Efficient type and memory safety for tiny embedded systems [OB/OL]. http://www.cs.utah.edu/～regehr/papers/plos06a.pdf.

(下转第 203 页)

[4] 刘少辉,盛秋戬. Rough 集高效算法的研究[J]. 计算机学报, 2003,26(5): 524-529.
[5] 徐章艳,刘作鹏,杨炳儒,等. 一个复杂度为 max($O(|C||U|$, $O(|C|2|U/C|)$的快速属性约简算法[J]. 计算机学报, 2006, 29(3): 391-399.
[6] Bazan J, Nguyen H S, Nguyen S H, et al. Rough set algorithms in classification problem[C]// Rough Set Methods and Applications: New Developments in Knowledge Discovery in Information System. Heidelberg: Physica-verlag, 2000: 49-88.
[7] Bazan J. A comparison of dynamic non-dynamic and rough set methods for extracting laws from decision tables[C]// Rough Set in Knowledge Discovery. Heidelberg: physica-verlag, 1998: 321-365.
[8] Deng D Y, Huang H K. A new discernibility matrix and function[C]// Proceedings of the Rough Set and Knowledge Technology. Berlin: Springer-Verlag, 2006: 114-121.
[9] Shan N, Ziarko W. An incremental learning algorithm for constructing decision rules[C]// Proceedings of the RSKD'93. London: Springer-Verlag, 1993: 326-334.
[10] 王亚英,邵惠鹤. 一种两类决策系统的递增式粗集归纳学习算法[J]. 信息与控制, 2002, 29(6): 521-525.
[11] Guan J W, Bell D A. Rough computational methods for information system[J]. Artificial Intelligence, 1998, 105(1-2): 77-103.
[12] Guan J W, Bell D A, Guan Z. Matrix computation for information systems[J]. Information Sciences, 2001, 131(1-4): 129-156.
[13] 文香军,蔡云泽,谭天乐,等. 基于粗糙属性向量树的规则提取快速矩阵算法[J]. 电子学报, 2006, 34(1): 65-70.
[14] 谭天乐,李平,宋执环. 基于粗糙集的逻辑故障树方法及其应用[J]. 仪器仪表学报, 2004, 25(1): 18-22.
[15] 郝丽娜,徐心和. 粗糙集神经网络系统在故障诊断中的应用[J]. 控制理论与应用, 2001, 18(5): 681-685.
[16] Skowron A, Rauszer C. The discernibility matrices and functions in information systems[C]// Intelligent Decision Support, Handbook of Applications and Advances of the Rough Sets Theory. Dordrecht: Kluwer Academic Publishers, 1992: 331-362.
[17] Hu X H, Cercone N. Learning in relational database: a rough set approach[J]. Computational Intelligence, 1995, 11(2): 323-337.
[18] 程玉胜,张佑生,胡学钢. 基于决策类分割的动态数据环境下的归纳学习[J]. 系统仿真学报, 2007, 19(12): 2 864-2 867, 2 871.
[19] 程玉胜. 基于粗糙集理论的知识不确定性度量与规则获取方法研究[D]. 合肥工业大学, 2007.

(上接第 195 页)

[6] Splint. LCLint [OB/OL]. http://lclint.cs.virginia.edu/.
[7] Gimpel Software. PCLint [OB/OL]. http://www.gimpel.com/.
[8] 易宇,金然. 基于符号执行的内核级 Rootkit 静态检测[J]. 计算机工程与设计, 2006, 27(16): 3 064-3 068.
[9] Harris L C, Miller B P. Practical analysis of stripped binary code[J]. Workshop on Binary Instrumentation and Applications, 2005, 33(5): 63-68.
[10] Brumley D, Newsome J, Song D, et al. Towards automatic generation of vulnerability-based signatures [C]// Proceedings of the 2006 IEEE Symposium on Security and Privacy. Washington: IEEE Computer Society, 2006: 2-16.
[11] Henzinger T A, Jhala R, Majumdar R, et al. Extreme model checking [C]// International Symposium on Verification: Theory and Practice, Lecture Notes in Computer Science. Springer-Verlag, 2003: 332-358.
[12] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: a free, commercially representative embedded benchmark suite[C]// 4th IEEE Annual Workshop on Workload Characterization. Austin: IEEE Press, 2001: 3-14.