

# 一种混合式内存泄漏静态检测方法

胡燕<sup>1</sup>, 龚育昌<sup>2</sup>, 孙伟峰<sup>1</sup>, 赵振西<sup>2</sup>

<sup>1</sup>(大连理工大学 软件学院, 辽宁 大连 116620)

<sup>2</sup>(中国科学技术大学 计算机科学技术系, 安徽 合肥 230027)

E-mail: yhu6@mail.ustc.edu.cn

**摘要:** 内存泄漏是导致系统性能降低的重要问题, 提出一种基于模型检测算法的内存泄漏静态检查方法 TMC. 该方法依据程序的控制流图构建对应于程序执行的有限状态自动机, 进而在此基础上应用模型检测算法分析程序中可能存在的内存泄漏. 论文利用几个典型的程序实例详细说明了 TMC 的工作原理, 并通过基于内存操作密集测试程序集 PtrDist 的实验对 TMC 进行了验证. 实验结果表明, TMC 能够显著提升内存泄漏分析的精度.

**关键词:** 模型检测; 内存泄漏; 别名分析; 自动机

中图分类号: TP311

文献标识码: A

文章编号: 1000-1220(2008)10-1935-05

## Hybrid Static Method for Memory Leak Detection

HU Yan<sup>1</sup>, GONG Yu-chang<sup>2</sup>, SUN Wei-feng<sup>1</sup>, ZHAO Zhen-xi<sup>2</sup>

<sup>1</sup>(School of Software, Dalian University of Technology, Dalian 116620 China)

<sup>2</sup>(Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027 China)

**Abstract:** Memory leak is the key factor that causes system performance degradation. A static memory leak detection method based on model checking (TMC) is proposed in this paper. In the method, an abstract finite state automata expressing control flow graph of program is constructed first, then a model checking based analysis algorithm is applied to check memory leaks. Several sample programs are analyzed to show the details of how the TMC method works. Finally the pointer-intensive benchmark suite PtrDist is used to test and verify the TMC method. The experimental results show that TMC can distinctly improve the precision of memory leak analysis.

**Key words:** model checking; memory leak; alias analysis; automata

## 1 引言

在使用 C/C++ 等支持显式内存分配的语言所编写的程序中, 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.

内存泄漏的检测方法分为两大类, 一类是动态检测方法<sup>[1-4]</sup>, 另一类是静态检测方法<sup>[5,6]</sup>. 动态检测方法是在程序中进行动态内存分配时, 在堆中作为标记. 程序退出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. 动态检测方法便于实现, 但是内存泄漏往往发生在某条代码路径上, 要找出能够覆盖特定的代码路径的程序输入并不容易, 这往往需要依靠经验. 在程序变得庞大和

复杂时, 使用动态检测方法很难满足应用需求. 动态分析方法的另一个缺点是它只能给出泄漏的堆上内存的在程序中的初始分配位置, 而不能给出导致内存泄漏的操作的准确位置.

内存泄漏的静态检查方法基于类型进行分析, 文献[7]中详细描述了利用类型分析保障程序的内存安全性的方法. Lint 工具<sup>[8]</sup>同样是利用类型分析方法, 这类工具需要手工对访问操作进行注释以辅助内存分析. 文献[5,6]给出了能够自动进行内存类型推断的类型分析方法, 减少了对指针注释的数量, 但是仍然需要手工添加一定数量的类型注释. 另外, 由于类型分析方法中没有利用控制流信息, 使得基于类型的方法所得到结果的精度有一定局限.

模型检测算法<sup>[9]</sup>是被广泛应用的一种程序静态分析方法, 它能够很好地利用程序中的控制流和信息流进行更为精确的程序属性分析. 基于 BDD (Binary Decision Diagram) 的状态表示方法使得模型检测工具的效率已经得到大大提高<sup>[10]</sup>, 因此使用模型检测方法对程序进行属性检查已经变得切实可行.

本文针对类型分析方法的不足, 提出了基于模型检测技

术对内存泄漏问题进行分析的方法 TMC (Type and Model Checking). TMC 方法将类型分析与模型检测算法相结合,利用模型检测方法对类型分析得到的结果进行更为细致的检查. TMC 提升了内存泄漏分析的精度,并且可以为内存泄漏错误的诊断提供更为精确的信息. 文中详细介绍了 TMC 方法的原理,并利用具有代表性的 Benchmark 程序说明方法的可操作性与实用性.

## 2 内存泄漏检测方法

### 2.1 TMC 的工作流程

本文中为 TMC 方法设计的分析框架的基本工作流程如图 1 所示. 分析工具接受源程序作为输入,经过语法分析得到程序的抽象语法树(Abtract Syntax Tree, AST)表示形式. 程序的 AST 形式经过控制流分析过程形成程序的控制流图(Control Flow Graph, CFG).

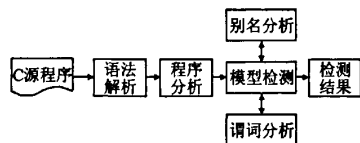


图 1 检测程序工作流程

Fig. 1 Memleak check workflow

TMC 模型检测算法基于 CFG 以及程序相关的数据流信息(如别名分析)等进行分析. 该方法基于程序的 CFG 构建程序的抽象表示形式,并基于类型分析获得模型检测方法的分析对象.

谓词分析模块(Predicate Analysis)用于为路径可达性分析建立谓词条件. 别名分析(Alias Analysis)用于提供程序中的内存块的指针别名信息.

### 2.2 PLL 与类型分析

图 1 中使用的模型检测算法的作用对象是潜在泄漏位置(Potential Leak Location, PLL),定义 1 给出的 PLL 的定义.

定义 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  $pc$  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  $p$  的赋值导致 Mem 成为被泄漏的内存区域. 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  $p$  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,从而导致内存的泄漏的发生.

每个 C 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的程序控制位置,  $lhs$  表示左端表达式,  $rhs$  则表示右端.

```

01 int f(void){
02   char *s;
03   s=malloc(50); /* get memory */
04   if(s==NULL)
05     return 1;
06   else
07     return 0; /* leak here */
08 }
09
10 int main(void){
11   while(1) f();
12   return 0;
13 }
  
```

图 2 示例程序

Fig. 2 Example program

程序中的 PLL 集合通过类型分析方法构建. 程序的返回语句对应的 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. 在分析程序中所有的赋值语句的过程中,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. 图 2 是一个存在 return 语句所导致内存泄漏的实例. 其中第 5、7、12 行都是 PLL. 第 3 行的表达式左端的变量  $s$  然是指针类型的,但是这个表达式的右端  $rhs$  却是对 malloc 函数的调用,因此该语句不是一个 PLL.

### 2.3 TMC 方法的工作原理

在通过类型分析获得程序中的 PLL 集合之后,模型检测算法针对每一个 PLL,对从程序的入口到 PLL 所在程序位置的可达性进行分析.

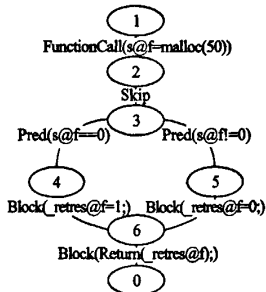


图 3 图 2 中函数  $f$  的 CFA

Fig. 3 CFA of function  $f$  in fig. 2

为了实施模型检测算法,首先应为程序构建一个抽象模型. 本文采取的抽象方法是依据 CFG 构建的有限状态自动机 CFA (Control Flow Automata). 图 3 给出的是图 2 中的程序中函数  $f$  的 CFA. 自动机中的状态是程序中的关于程序中的变量的状态的谓词,状态转换的条件则是程序中的语句. 图 3 中路径  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$  代表程序进入状态  $S_5 = (s@f \neq \text{NULL})$ ,而  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  则对应着状态  $S_4 = (S@f = \text{NULL})$ .

各个 CFA 通过程序的过程调用图联系起来,形成一个完整的程序的状态转换描述.

## 3 TMC 算法描述

TMC 方法中所采用的算法(称为 TMC 算法)是对基于谓词抽象的模型检测算法的拓展. 本节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集合构成了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:

(1) 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。例如图2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。

(2) 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,将经过缩减之后的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ppl\_set中的每个PLL,应用模型检测方法逐个进行分析。算法形式如下:

```
Foreach ppl in ppl_set do
  If(modelCheck(ppl) = NotSafe)
    Leak-list.add (ppl)
Foreach leak in leak-list
  Print_error-path (leak)
```

modelCheck(ppl)的目标是分析ppl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ppl是否安全,该ppl就会被添加到结果的集合leak-list中。与ppl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ppl的路径,用以辅助程序员对程序错误的来源进行分析。与leak-list中的LL相关联的错误路径最终被输出,可以作为对程序中可能存在的内存泄漏的分析与测试的依据。

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 $p=(S_0,op_1,S_1,\dots,op_n,S_n)$ ,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。

上述的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的分析,产生大量的谓词。过多的谓词会导致算法需要检测的状态数大大增加,降低程序的执行效率。针对内存泄漏这一实际问题,提出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

为了有效的进行内存泄漏分析,本小节对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。

逆向分析的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ppl是否安全。

在内存状态的描述中,每个程序含有一组内存块 $M=\{m_1,m_2,\dots,m_k\}$ ,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。

```
M ← ∅
FOREACH op IN path
  IF 操作是指针赋值形式 p=q THEN
    M ← M ∪ {m_p}
    m_p 的指针集合初始化为 {p,q}
  IF PLL.op 是 p=q 的形式 THEN
    IF m_p 的别名集合的元素数目 > 1 THEN
      return LEAK-FALSE
    ELIF 操作 op 的形式是 p=malloc(n) THEN
      M ← M ∪ {m_p}
      将 m_p 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 m_p 指向的指针中只有一个元素 THEN
        return LEAK-TRUE;
    ELIF 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
```

图4 逆向分析算法

Fig. 4 Reverse analysis algorithm

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。在逆向分析的过程中,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中,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.对逆向分析中的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,算法流程见上页图4.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的时候,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时的处理方法如下:

#### (1) 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之后, $p$ 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.对于返回语句所对应的PLL,则需要记录最近一次对内存变量的使用.

#### (2) 对 malloc 函数的调用

到达PLL的路径形式为 $p=malloc(\dots); \dots; p=q$ ;

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.

#### (3) 对 free 函数的调用

到达PLL的路径形式为 $Free(p); \dots; p=q$ ;

用于释放 $p$ 所指向的内存的方法的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.因此逆向分析的路径可以返回一个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

利用图2给出的内存泄漏问题的典型实例作为分析实例.其中,第3行通过调用 malloc 函数进行了动态内存分配,分配得到的内存被赋予 $f$ 中的指针变量 $p$ .如果内存分配不成功( $P=NULL$ ),那么函数 $f$ 返回1;否则,函数 $f$ 返回0.

根据TMC算法,为了处理堆上分配的内存存在函数结束前没有被释放的情形,每个进行了动态内存分配的函数的返回语句都应该被视为一个PLL.对于这一类情形,可分为3种情况:

1. 在函数返回点没有未释放的内存对象;
2. 在此函数内申请的内存对象,并且相应的指针没有被函数返回;
3. 指向函数的指针被作为返回值返回.

情况1和3都不会导致内存泄漏,而情况2则会导致内存

存泄漏的发生.图2中的第7行就是属于情况2的PLL实例.

图5给出了图2中实例程序中存在内存泄漏的程序执行路径.路径条件为 $(s@f) \neq 0$ ,由于 $s@f$ 是一个局部的指针变量,并且在 $s@f$ 没有被作为返回值返回,因此从对函数 $f$ 的调用返回之后, $s@f$ 指向的内存被泄漏.

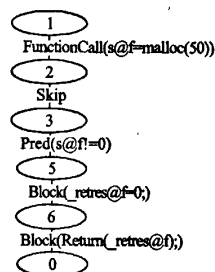


图5 图2中实例程序的泄漏路径

Fig.5 Memleak error trace of code in fig.2

另一例子是图6中求链表逆序的程序.该程序中,第8、10、11、12、13行都是PLL,而第8行可以通过预处理过程排除.要证明第12行的语句 $y=x$ 不会导致内存泄漏,首先假设在这个地方发生了内存泄漏.初始假设是 $y$ 是指向相应内存对象的唯一的指针.但是在第11行,这个假设就被否定,

因为实际上 $x \rightarrow next$ 和 $y$ 指向同一个内存对象.因此第12行的指针赋值不会导致内存泄漏.第13行的PLL也可以通过类似的分析方法排除.对第11行,跟第12行的分析类似,初始假设 $x \rightarrow next$ 是指向的唯一指针,而第10行的代码立刻否定了这一假设.因此第11行也不会导致内存泄漏.对第10行的代码,逆向分析的路径可能会有两种情况,一种是从reverse函数的入口基本块进入while循环的第1个循环体, $t$ 的初始值是NULL,这种情况不会产生泄漏问题;另一种情况则从循环的上一个循环体而来,这种情况下第13行的地址赋值语句会导致 $x$ 和 $t$ 指向同一个内存对象,从而使得原始的假设不成立.从而第10行也是安全的.

```

01 Typedef struct list{
02     Int data;
03     Struct list * next;
04 }List;
05
06 List * reverse(List * x){
07     List * y, * t;
08     y=NULL;
09     while(x!=NULL){
10         t=x->next;
11         x->next=y;
12         y=x;
13         x=t;
14     }
15     return y;
16 }

```

图6 示例程序:链表逆序

Fig.6 Example program: list reversal

经过上述的分析,可知过程reverse的执行不会导致内存泄漏问题的产生.

### 4.2 实验分析

本文采用了内存操作比较密集的PtrDist测试程序<sup>[11]</sup>作

为实验的对象.表 1 给出了使用我们的算法进行内存泄漏检测时各个主要模块的执行时间(测试平台:Pentium 2.0, Mandrake Linux 9.0,内核版本 2.4.20).

表 1 检测算法的执行时间(时间单位:毫秒)

程序名	语法规析	TMC 算法		错误路径	
	时间	执行时间	分析时间	分析时间	
anagram	80	500	190	80	
backprop	70	450	100	36	
bc	250	845	500	87	
ft	180	2950	460	840	
ks	260	3170	530	740	
yacr2	860	6800	430	3120	

图 7 则给出了各个模块执行时间占整体执行时间的比例,其中列出的是 MC 算法实现中各个主要模块的执行时间占总的模型检测时间的比例.(checkerror 是逆向分析潜在错误路径所使用的时间百分比,predAnalysis 则是对路径谓词进行分析所使用的时间百分比,coverage 表示建立程序的函数调用图所使用的时间,other 中包括了源程序的语法规析,以及数据流分析等所耗费的时间).

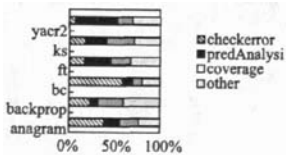


图 7 MC 算法各模块执行时间比例

Fig. 7 Time Consumption of MC algorithm

表 2 描述了对程序进行内存泄漏分析的结果.表中包括了每个程序中的 PLL 数目,和单纯的类型方法给出的内存泄漏检测的结果(Non-MC),以及使用基于模型检测的方法的处理结果(TMC).实验数据表明,与单纯基于类型的检测算法(Non-MC)相比,基于模型检测算法的内存泄漏检测方法能够将疑似内存泄漏点控制在较小的范围内.两种方法(Non-MC 和 TMC)给出的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.在存在内存泄漏错误的测试程序 bc 和 ft 中,使用 Non-MC 方法的结果给出的错误

表 2 内存泄漏的相关分析数据

测试程序	PLL 数目	Non-MC	TMC	leaks
anagram	19	10	4	0
backprop	10	6	5	0
Bc	57	13	3	1
Ft	42	19	4	1
Ks	103	37	10	0
Yacr2	46.	7	1	0

警告数目分别是 13,19,相对庞大,而且没有可供查看的错误路径供程序对内存泄漏错误原因进行分析.而使用 TMC 方法给出的警告数则分别为 3 和 4,同时给出导致内存泄漏的错误路径,这些错误路径为查找内存泄漏的原因提供了充分的信息.

5 结束语

本文在充分分析了内存泄漏问题的成因以及已有分析方法的基础上,提出了一种基于模型检测技术的内存泄漏的静态分析方法 TMC.该方法结合了类型分析与模型检测方法各自的优点,具有比类型分析方法更为精确的分析结果.文中通过 Benchmark 程序实例验证了 TMC 方法的有效性.

进一步的工作则是增强分析工具的功能,使其能够应用于大型的软件的属性检查.

References:

[1] Maria Jump, Kathryn S McKinley. 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] Nicholas Nethercote, Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]. Proceedings of PLDI 2007, San Diego, California, USA, June 2007.

[3] Huang Yao. The research and implementation of memory leak detection and analysis of C/C++ program[D]. Software Engineering Institute of Beijing University of Aeronautics and Astronautics, 2004.

[4] Gao Hai-chang, Feng Bo-qin, He Hang-jun, et al. Dynamic memory testing based on source code instrumentation on linux platform[J]. Mini-Micro Systems, 2006, 27(9): 49-53.

[5] David L Heine, Monica S Lam. Static detection of leaks in polymorphic containers[C]. Proceeding of the 28th international conference on Software engineering, May 20-28, 2006, Shanghai, China.

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

[7] Regehr J, Coopriider N, Archer W, et al. Efficient type and memory safety for tiny embedded systems[C]. In Proceedings of the PLOS 2006 Workshop on Linguistic Support for Modern Operating Systems, San Jose, California, October 2006.

[8] Barker C. Static error checking of C applications ported from UNIX to WIN32 systems using LCLint[R]. Senior Thesis, Dept. Computer Science, University of Virginia, Charlottesville, 2001.

[9] Henzinger T A, Jhala R, Majumdar R, et al. Extreme model checking [C]. In: International Symposium on Verification, Theory and Practice, LNCS 2772, Springer, 2003, 332-358.

[10] Yang B, Bryant R E, O'Hallaron D R, et al. A performance study of BDD-based model checking[C]. In Proceedings of the Formal Methods on Computer-Aided Design, November 1998, 255-289.

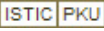
附中文参考文献:

[3] 黄 瑶. C/C++ 程序内存泄漏检测和分析技术的研究与实现 [D]. 北京航空航天大学软件学院, 2004.

[4] 高海昌,冯博琴,何杭军,等. Linux 平台下基于源代码插桩的动态内存检测[J]. 小型微型计算机系统, 2006, 27(9): 49-53.

作者：[胡燕](#), [龚育昌](#), [孙伟峰](#), [赵振西](#), [HU Yan](#), [GONG Yu-chang](#), [SUN Wei-feng](#), [ZHAO Zhen-xi](#)

作者单位：[胡燕, 孙伟峰, HU Yan, SUN Wei-feng \(大连理工大学, 软件学院, 辽宁, 大连, 116620\)](#), [龚育昌, 赵振西, GONG Yu-chang, ZHAO Zhen-xi \(中国科学技术大学, 计算机科学技术系, 安徽, 合肥, 230027\)](#)

刊名：[小型微型计算机系统](#) 

英文刊名：[JOURNAL OF CHINESE COMPUTER SYSTEMS](#)

年, 卷(期)：2008, 29(10)

被引用次数：0次

参考文献(12条)

1. Maria Jump. [Kathryn S McKinley Cork:dynamic memory leak detection for java](#)[Technical Report TR-06-07] 2006
2. Nicholas Nethercote. [Julian Seward Valgrind:a framework for heavyweight dynamic binary instrumentation](#) 2007
3. Huang Yao [The research and implementation of memory leak detection and analysis of C/C++ program](#) 2004
4. Gao Hai-chang. [Feng Bo-qin. He Hang-jun Dynamic memory testing based on source code instrumentation on linux platform](#)[期刊论文]-[Mini-micro Systems](#) 2006(09)
5. David L Heine. [Monica S Lam Static detection of leaks in polymorphic containers](#) 2006
6. Heine D L [Static memory leak detection](#) 2004
7. Regehr J. Coopridner N. Archer W [Efficient type and memory safety for tiny embedded systems](#) 2006
8. Barker C [Static error checking of C applications ported from UNIX to WIN32 systems using LCLint](#) 2001
9. Henzinger T A. Jhala R. Majumdar R [Extreme model checking](#) 2003
10. Yang B. Bryant R E. O'Hallaron D R A [performance study of BDD-based model checking](#) 1998
11. 黄瑶 [C/C++程序内存泄漏检测和分析技术的研究与实现](#)[学位论文] 2004
12. 高海昌. 冯博琴. 何杭军 [Linux平台下基于源代码插桩的动态内存检测](#)[期刊论文]-[小型微型计算机系统](#) 2006(09)

相似文献(2条)

1. 期刊论文 [龚育昌. 胡燕. 张晖. 赵振西. GONG Yuc-hang. HU Yan. ZHANG Ye. ZHAO Zhen-xi 一种针对可执行代码的内存泄漏静态分析方案 -中国科学技术大学学报](#)2009, 39(2)

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.

2. 学位论文 [胡燕 编译过程安全性基础研究](#) 2007

程序安全性已成为现代软件开发中必须重视的关键问题.在软件开发流程中,从设计到编码到编译为最终的可执行代码,任何一个阶段的安全隐患都可能导致最终软件的安全性问题.因此,对软件开发流程的各个阶段进行安全性测试是保证软件质量的一个必要部分.在软件开发过程中,编译器负责从软件源代码到目标代码的变换这一重要阶段,因此编译过程的正确性和安全性对软件的安全性有着非常大的影响.目前,针对软硬件系统的形式化验证技术发展迅速,相关工具的工具也逐步成熟,这些进步使得对编译器这类复杂的软件进行验证成为可能.由于编译验证在安全软件开发中的重要意义,它已经成为当前研究的一个热点.

本文以编译过程验证为主要研究方向,针对编译验证的核心问题,对编译器安全性的验证方法进行了探索研究,其中包括对程序属性的描述与分析方法的研究,以及对基本的编译验证方案的探索.文中提出了一种基于程序分析的编译验证框架,该框架基于对编译过程中个阶段的编译中间表示形式的分析验证编译过程是否保证了特定的安全特性.文中研究了对源语言程序和中间代码的安全属性分析方法,提出了一种新颖的算法,并将利用其对重要的软件安全属性进行分析.另外,文中还提出了一种新颖有效的二进制代码的分析方案,将对高级语言程序的属性检查方法扩展到了可执行程序领域,因此使得能够采用统一的分析方法完成程序在编译各个阶段的各种形式的安全属性验证,从而为本文提出的编译验证框架提供了有效的安全属性分析手段.主要研究内容包括:

(1) 在深入分析编译验证的根本问题和研究方法的基础上, 提出了一种基于程序分析的编译验证框架。在该验证框架中, 编译验证与分析作为一个独立的大模块集成到一个已有编译器实现中, 通过对编译各个阶段中的程序形式的安全属性检查, 验证编译器实现的正确性与安全性。

(2) 针对基于程序分析的编译验证框架的基本需求, 在深入分析程序基本属性的形式化描述方法的基础上, 研究了内存安全属性与信息流安全属性这两种重要的安全属性的描述方法, 提出了基于类型精化方法的内存安全性的统一表示方法。类型精化是基于对已有类型进行扩展的一种方法, 这种方法对于描述程序的安全属性有着重要的意义。另外, 还将类型扩展技术应用在信息流安全属性描述方面, 并以SSA中间语言作为载体进行了信息流安全属性描述的一个实例研究。(3) 在分析了编译器的典型实现技术的基础上, 提出了对编译几个主要阶段的正确性验证方法。在分析了parse(从具体语法到抽象语法的语法规则)与unparse(从抽象语法到具体语法, 将抽象语法树线性化)之间的关系, 提出基于parse-unparse的编译前端验证方案。基于对代码实现算法的分析与研究, 提出了基于对树重写条件检查的正确性验证方案, 对树模式匹配的条件进行检验, 从而验证代码生成算法的正确性。基于对编译优化算法实现的深入分析, 提出了一种能够有效地表示程序中的数据流属性以及控制流属性的扩展属性文法, 并基于此提出了基于属性文法计算程序属性并生成程序的抽象模型, 进而利用模型检测工具检验编译优化正确性的验证方案。

(4) 在深入研究了程序分析方法之后, 提出了基于类型分析与模型检测方法相结合的混合式分析方法TCMC(Type Checking and Model Checking), 并将其应用于程序的内存安全属性以及信息流安全性分析。其中特别针对内存泄漏的静态分析方法, 说明了TCMC算法的应用, 及其相对于类型分析方法的劣势。

(5) 在深入分析底层代码的验证问题及其研究方法的基础上, 提出了一种新的目标机器代码分析方法。该方法基于反编译技术恢复程序中的控制流, 并基于恢复的控制流程图对程序进行数据流分析和类型分析, 也可以将控制流程图转换为SSA形式, 进而利用TCMC算法进行程序安全属性的分析。

本文做出的主要贡献如下:

(1) 提出一个基于程序分析的编译验证框架。该框架建立在一个具有良好模块化结构的编译器实现之上, 各个主要验证模块作为对编译器的扩展很方便地集成到编译器的编译过程中。该框架利用SSA形式作为统一的编译中间表示形式, 便于程序属性分析, 也便于向框架中添加自定义的分析模块。

(2) 提出基于类型精化的内存安全性统一表示, 同时提出了在SSA中间表示形式上基于类型扩展进行信息流安全编码的方法。这部分程序属性描述方法的研究为验证框架中的属性检测方法研究打下了基础。

(3) 提出了类型分析与模型检测方法相结合的TCMC算法, 并将其应用于程序安全属性的分析过程中。该算法避免了类型分析算法对控制流不敏感的不足, 利用模型检测算法对类型分析的结果进行分析, 提高了分析的精度, 同时也避免了单纯使用模型检测算法进行分析可能造成的较大时间开销。

(4) 提出了基于反编译技术对汇编码和二进制程序进行分析的方法。该方法不需要编译器提供额外的调试符号信息, 能够直接从二进制代码中恢复其控制流与数据流结构。这种方法的提出为底层代码安全属性提供了有效的方法, 与基于源语言、中间语言的属性分析一起, 为基于程序分析的编译验证框架提供了一整套有效的分析工具。

本文链接: [http://d.wanfangdata.com.cn/Periodical\\_xwxjsjxt200810038.aspx](http://d.wanfangdata.com.cn/Periodical_xwxjsjxt200810038.aspx)

授权使用: 大连理工大学图书馆(d11lg), 授权号: 551af36a-3904-49f3-ba50-9e510116a82a

下载时间: 2010年12月19日