基于复杂控制流的源代码内存泄漏静态检测

姬秀娟[1] 孙晓卉[2] 许 静[3]
(南开大学滨海学院 天津 300270)[1] (京东数字科技 北京 100176)[2]
(南开大学人工智能学院 天津 300350)[3]

摘 要 C/C++源代码因其手动分配堆内存的特点,造成大量内存泄漏的问题。对于多分支的控制流结构,其内存分配点和释放点具有不确定性,使得内存泄漏检测的难度增大。针对这种复杂控制流中的内存泄漏问题,定义了一种基于路径抽象的内存泄漏分类方法,提出了一种基于投影的模型检测静态分析算法。该算法采用投影技术将原控制流图进行规约和简化;同时,在进行函数间的分析时,融合了 Cloning Expands the ICFG 和 Expanded Supergraph 两种方法,构建了一个函数间的内存定义-使用控制流图(Interprocedural Memory Control Flow Graph,IMCFG)。实验表明,该检测方法在有效性和准确率方面明显取得了较好的结果。

关键词 内存泄漏,复杂控制流,投影,静态分析,C/C++
中图法分类号 TP311 **文献标识码** A

Source Code Memory Leak Static Detection Based on Complex Control Flow

JI Xiu-juan[1] SUN Xiao-hui[2] XU Jing[3]
(Binhai College, Nankai University, Tainjin 300270, China)[1]
(Jingdong Digital Technology, Beijing 100176, China)[2]
(College of Artificial Intelligence, Nankai University, Tianjin 300350, China)[3]

Abstract C/C++ source code has a lot of memory leaks because of its manual allocation of heap memory space. Regarding complex control flow with multiple branches in the control flow graph, it is more difficult to detect memory leaks because of the uncertainty of memory allocating and releasing. A memory leak classification method was defined based on path abstraction in complex control flow. A projection-based model detection based on the analysis algorithm was proposed, where the original control flow graph is projected, thus simplifying and regulating the control flow graph. Meanwhile, in the inter-procedural analysis, by combining Cloning Expands the ICFG approach and Expanded Supergraph approach, a Inter-procedural Memory def-use Control Flow Graph(IMCFG) was built. At last, this algorithm is proved to be effective and precise by experiments.

Keywords Memory leak, Complex control flow, Projection, Static detection, C/C++

1 引言

1.1 内存泄漏相关研究背景

随着软件的设计和功能变得越来越复杂,对性能尤其是安全方面的要求也越来越高。软件的代码安全直接影响软件的可靠性及可维护性等,其中最为突出的安全问题之一是内存泄漏。首先,内存泄漏是一种很常见的漏洞。根据标记漏洞的 US-CERT 数据库的报告显示[1],自 1991 年以来 30%以上的漏洞均来自内存泄漏或损坏;其次,按照 CMM5 中的定义规范[2],内存泄漏对应 BUG 体系中的"非常严重"等级,是致命错误;最后,相对其他错误来说内存泄漏很难再现,比较隐蔽,编译器不容易察觉。系统发生内存泄漏后的具体表现:内存资源耗尽,机器失去响应;进程 ID 耗尽,无法创建新的进程;硬盘耗尽,内存交换和日志的使用功能受到限制[3]。

1.2 C/C++语言源代码中的内存泄漏

本文主要研究 C/C++源代码中的堆内存泄漏。堆内存是在程序中由程序员申请,使用后必须释放的内存。在 C 语言中,程序通过调用标准库的内存分配函数来分配内存,而且需要在使用结束后调用 free()函数释放相应的内存空间。内存分配函数主要包括 calloc(),malloc(),realloc();其中 calloc()和 malloc()的作用是从堆内存空间中任意分配大小、连续的内存单位,并返回指向首地址的指针,而 realloc()函数是在已分配的内存空间上重新分配内存。在C++语言中,程序使用 new 和 delete 来动态分配和释放内存。C/C++在管理内存时,没有合适的内存回收机制,这就很容易产生内存泄漏。尤其在嵌入式系统中,由于内存的限制,频繁地分配不定大小的内存会引起很大问题[4]。

本文第 1 节介绍了内存泄露的研究背景;第 2 节介绍了目前流行的静态检测方法与技术等的相关工作;第 3 节从路径抽象的角度对内存泄漏的情况进行分类,在分类的基础上,提出了一种基于投影的模型检测方法;第 4 节通过实验对比分析,验证了该方法性能的优越性,同时也发现了该方法的不足之处。

本文受天津市自然科学基金重点项目(17JCZDJC30700),天津市科技职称项目(17YFZCGX00610)资助。
姬秀娟(1977-),博士,副教授,主要研究领域为软件分析、软件工程,E-mail:9682561@qq.com。

2 相关研究工作

2.1 内存泄漏检测方法

国内外内存泄漏检测的研究方法有很多,主要分为动态检测和静态检测[5]。动态检测的缺点是只能检测到测试用例覆盖的漏洞[6]。而静态检测不需要运行源程序和完备的测试用例,可通过直接分析来检查程序,精确地定位漏洞。目前静态检测常采用的方法有符号执行和约束求解、类型推导、基于规则的检查等[7-11]。其中,符号执行和约束求解方法在处理大规模程序时,路径的数目随着程序尺寸的增大呈指数级增长,带来了"路径爆炸"和"无限搜索空间"等问题,使得时间复杂度增大;类型推导虽然适合大规模程序,但与控制流分析无关,不合适复杂控制流问题;基于规则的检查由于规则受到描述机制的局限,导致其扩展性较差。

近年来,许多研究人员采用建模的方法来分析程序源代码。例如,文献[12]提出的堆抽象建模方法,但在实际应用中的精度还有待提高;文献[13]利用模型工具验证断言的可达性来判断内存泄漏,但自动化水平还有待完善;文献[14]采用的内存状态转换图(Memory State Transition Diagram,MSTD)在某些情况下误报率较高。

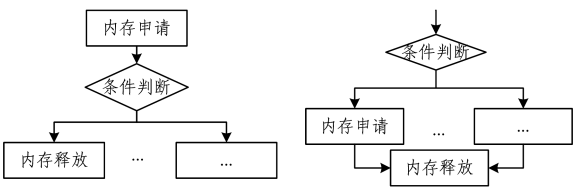
针对复杂控制流下的内存泄漏检测,本文提出了一种基于投影的模型检测方法。该方法通过投影来简化原程序控制流图,采用自底向上遍历查找节点的直接前驱方法来检查内存泄漏。

2.2 基于路径抽象的内存泄漏分类

程序只有一个入口,但由于函数间的调用、条件分支等多种因素的影响,可能会有多个出口[15]。一个复杂的程序通常会出现分支(或循环的)连锁和嵌套的情况。如果一个内存空间被分配后,在其后的所有路径中没有得到相应的释放,那么该内存空间有可能被泄漏。如果内存的分配语句也在条件分支内,而程序需要对条件谓词进行判断,这就大大增加了分析问题的复杂度。本文将连锁和嵌套结构与程序的 3 种控制流相组合,给出了以下 6 种静态检测中内存泄漏的情形。

2.2.1 顺序-分支结构中的内存泄漏

在这种结构中,内存的分配或释放会出现在条件分支中,使得程序在运行时根据条件谓词的取值不同而执行不同的分支,从而造成内存泄漏。具体来说,顺序-分支结构中的内存泄漏分为两种:动态内存在条件分支中释放(见图 1(a))和动态内存在条件分支中分配(见图 1(b))。

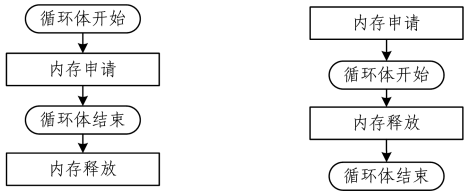


(a)内存在条件分支中释放 (b)内存在条件分支中申请

图 1 顺序-分支结构中的内存泄漏

2.2.2 顺序-循环结构中的内存泄漏

顺序-循环结构是指内存的动态分配与释放中,有且只有一个出现在程序的循环体内。本文只讨论对同一个指针的分配和释放,在图 2(a)中,在循环体内对同一个指针重复申请分配内存,使得指针只能指向最后一次申请的内存空间,造成前面循环中所分配内存的泄漏;同样,在图 2(b)中,内存在循环体中被重复释放,但同一指针只能被释放一次,这种情况会出现编译错误,本文不做详细讨论。

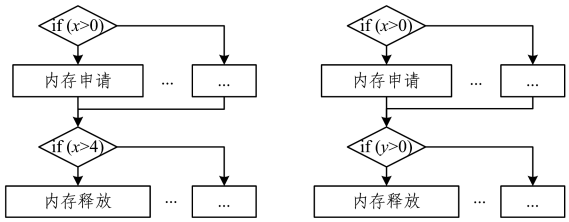


(a)内存在循环体中申请 (b)内存在循环体中释放

图 2 顺序-循环结构中的内存泄漏

2.2.3 连锁-分支结构中的内存泄漏

连锁-分支结构即内存的分配与释放全部出现在条件分支内。这种结构使内存泄漏的静态分析变得更加复杂,因为内存的分配与释放都需要对条件谓词进行判断。按照分支条件是否相同可以将连锁-分支结构中的内存泄漏分为图 3 中的(a)和(b)两种。这两种分类在静态分析中没有本质区别,但可以为实验提供不同的测试用例。

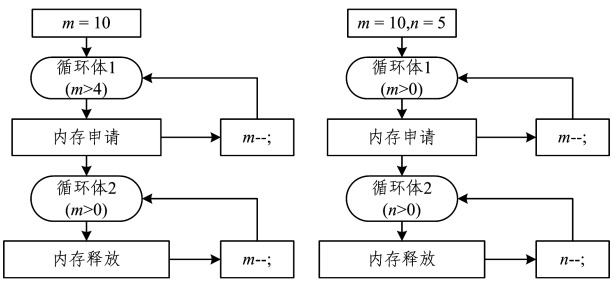


(a)分支内条件变量相同 (b)分支内条件变量不同

图 3 连锁-分支结构中的内存泄漏

2.2.4 连锁-循环结构中的内存泄漏

连锁-循环结构即内存的分配和释放分别出现在连锁结构的循环体内,而内存在循环体内的重复分配和重复释放最终会导致内存空间的泄漏。同理,根据循环体内的循环变量是否相同可将连锁-循环结构中的内存泄漏分为循环变量相同(见图 4(a))和循环变量不同(见图 4(b))。



(a)循环变量相同 (b)循环变量不同

图 4 连锁-循环结构中的内存泄漏

2.2.5 嵌套-分支结构中的内存泄漏

在嵌套-分支结构中,内存泄漏出现在内外相嵌的两层条件分支内,如图 5 所示。如果 $x>0$,即外层条件谓词取值为真,则内存空间被分配,内存释放出现在条件分支内。根据内层条件变量的取值不同(如果 $x=2$),若内存释放所在的分支未被执行,就造成了内存泄漏。

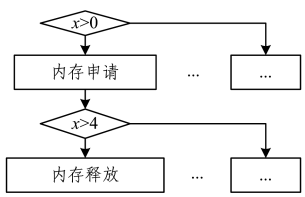


图 5 嵌套-分支结构中的内存泄漏

2.2.6　嵌套-循环结构中的内存泄漏

嵌套-循环结构如图 6 所示,在内外相嵌的两层循环体中对同一个指针分别进行内存的分配和释放。在该图中,变量 m 的初始值为 10,由于两个循环体内变量的取值范围不同,使得内外层循环体执行的次数也不同,且在一个循环体内对同一指针变量进行申请或释放内存本身就造成了内存错误。因此,这种情况也会造成内存泄漏。

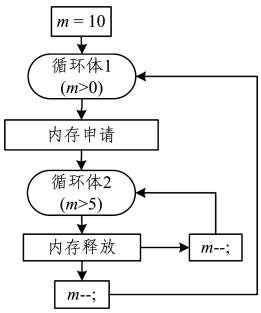


图 6　嵌套-循环结构中的内存泄漏

3　基于投影的内存泄漏静态检测方法

为了简化复杂控制流中内存泄漏的检测,本文提出了基于投影的静态检测方法,以对复杂路径进行规约和简化。该方法在分析词法和语法并生成抽象语法树的过程中,将函数内的程序控制流图投影到简化的控制流图中,然后对程序结构进行分析。

3.1　函数内内存泄漏分析

3.1.1　内存泄漏模型概述

将上述分类进行总结并抽象,给出了下面控制流图中动态内存的泄漏模型。以 C 语言为例,在包含内存动态分配与释放的程序源代码中,如图 7 所示,$Malloc(p)$和 $Free(p)$分别代表对同一指针的内存分配和释放。$C1$ 表示内存分配所在的控制流分支,$C2$ 表示内存释放所在的控制流分支。

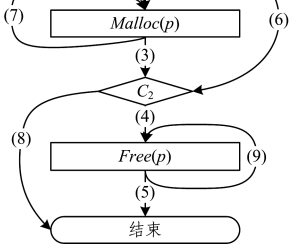


图 7　控制流图中内存泄漏模型

两种情况,即内存未被分配(对应图 7 中的路径(1)(6)(8))和内存分配后在程序结束前被释放(对应图 7 中的路径(1)(2)(3)(4)(5))都不存在内存泄露。其他路径都会由于控制流分支点处的取值不同而造成内存泄漏,具体分析。

1)$C1$=False,$C2$=True:对应图 7 中的路径(1)(6)(4)(5)和(1)(6)(4)(9)(5),即未被分配内存而被释放或重复释放而造成的内存泄漏。

2)$C1$=True,$C2$=False:对应图 7 中的路径(1)(2)(3)(8)和(1)(2)(7)(3)(8),即内存被分配或多次分配后,在程序结束前没有被释放而造成的内存泄漏。

3)$C1$=True,$C2$=True:其中造成内存泄漏的路径包括(1)(2)(3)(4)(9)(5)、(1)(2)(7)(3)(4)(5)和(1)(2)(7)(3)(4)(9)(5),也就是对同一指针重复进行内存分配和释放的过程。

3.1.2　控制流图投影算法

一段程序的控制流图可看作一个有向图 G,控制流图中的语句可看作顶点 n,所有语句的集合为 N,语句之间的执行顺序为 e,即顶点之间的边为 e,E 为所有边的集合。下面给出程序控制流图的投影定义和定理[16]。

定义 1(控制流图投影定义)　$G=(N,E)$和 $G^*=(N^*,E^*)$均为有向图,集合 M 代表程序中与内存管理有关的语句节点(以内存分配与释放为主),集合 B 代表所有控制流分支节点(if,else,while,for 等)。而 G^* 是 G 关于 N^* 的投影,记作 $G^*=P[G,N^*]$,如果满足:

(1)$N^*\subseteq N$;

(2)$\forall m,n\in N^*(m\neq n,m\in\Gamma(n))$($\Gamma(n)$代表 n 的后继节点),那么必存在一条有向通路 $\mu=(m,\cdots,n)\in G$;

(3)在 G^* 中,对于 $n\in N^*\cap B,m\in M$。如果 $m\notin\Gamma(n)$,那么 $\Gamma(n)\leftarrow(n,\Gamma(n))$。

定理 1(控制流图投影定理)　设有向图 $G=(N,E)$,$N^*\subseteq N$,且 $G^*=P[G,N^*]$,那么 G^* 严格保持了 N^* 中的点在 G 中的相互间可达关系。也就是说,$\forall m,n\in N^*$,m 在 G^* 中可到达 n 当且仅当 m 在 G 中可到达 n。

证明:取任意 $m,n\in N^*$。

(必要性)设 m 在 G^* 中可达 n,即存在有向通路 $\mu^*=(e_1^*,e_2^*,\cdots,e_k^*)\in G^*$。那么对于任意 $e_i^*,e_j^*\in\mu^*$ $(1\leqslant i<j\leqslant k)$,同时满足 $e_j^*\in\Gamma(e_i^*)$,必定存在 $\mu=(e_i^*,\cdots,e_j^*)\in G$。对于 $e_j^*,e_t^*\in\mu^*$ $(1\leqslant j<t\leqslant k)$,同时满足 $e_t^*\in\Gamma(e_j^*)$,必定存在 $\mu=(e_j^*,\cdots,e_t^*)\in G$。以此类推,可以推出有向通路 $\mu=(m,\cdots,n)\in G$,即 m 在 G 中可达 n。

(充分性)设 m 在 G 中可达 n,即存在有向通路 $\mu=(m,\cdots,n)\in G$,此时必有:

1)μ 上除了端点 m,n 外,无 N^* 中的点;

2)$\mu=(m,\cdots,n_1,\cdots,n_2,\cdots,n_k,\cdots,n)$,其中 $m,n,n_i\in N^*$ $(i=1,2,\cdots,k)$,其余节点不属于 N^*。

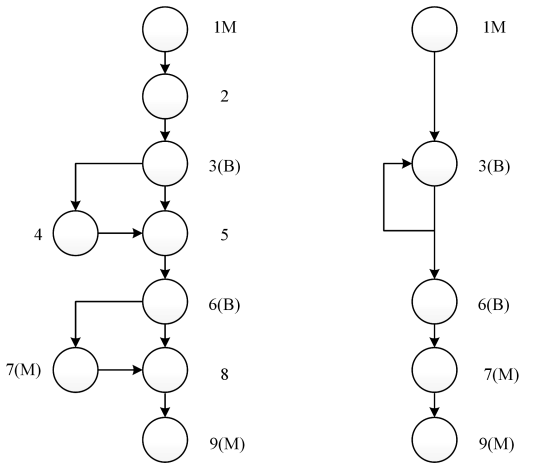
对 1),根据控制流图投影定义,可知 m 在 G^* 中可达 n,同时 m,n 在 G 中为相邻关系。由以上两点可推出 $n\in\Gamma(m)$。

对 2),可设 $\mu=\mu_1\cdot\mu_2\cdots\mu_{k+1}$,其中 $\mu_1=(m,\cdots,n_1)$,$\mu_i=(n_{i-1},\cdots,n_i)$ $(2\leqslant i\leqslant k)$,$\mu_{k+1}=(n_k,\cdots,n)$,且各个子路上除了端点外皆非 N^* 中的点。对每个子路的分析同 1),可知在 G^* 中有向通 $\mu^*=(e_1^*,\cdots,e_{k+1}^*)$,即在 G^* 中 $n\in\Gamma(m)$。

根据控制流图投影定理可知:G^* 其实是由从 G 中抽取的部分节点 N^* 组成,而且严格保持了 N^* 中的点在 G 中的相互间可达关系的图。特别地,对 $N^*=N$,有 $P[G,N^*]=G$,即对原图的所有节点投影将保持原图不变。

图 8 给出了一个控制流图投影的示例。图 8(a)展示了一段程序在进行投影前的控制流图 G,其中 1,7,9 为内存分配与释放的相关节点(即 $M=\{1,7,9\}$),3 和 6 为控制流分支

点(即 $B=\{3,6\}$),而其他的为普通节点。图 8(b)为该段程序关于 $N^*=\{M,B\}$的投影图 G^*。



(a)投影前的控制流图 G (b)投影后的控制流图 G^*

图 8 控制流图投影示例

下面给出了根据已知控制流图和投影节点求投影图的算法。设 $G=(N,E)$为程序的控制流图,N 为程序中所有语句的集合。对于内存中可以存储的数据,红黑树的搜索效率较高,同时避免了二叉树退化成链表,因此算法 1 采用了红黑树来存储节点集合 N。E 为语句之间执行的先后关系,具体实现用集合 $Map()$来表示;$Map()$中的元素为 N 中的节点 n 与其后继节点 $\Gamma(n)$的映射关系。$G^*=P[G,N^*]$是原控制流图 G 关于 N^* 的投影,$N^*=\{A,B,F\}$,其中 A 代表所有动态内存分配语句所在节点的集合,F 代表所有动态内存释放语句所在节点的集合。在算法 1 中,$n.$ind 代表节点 n 的入度,$n.$outd 代表节点 n 的出度。同时说明,对于循环节点,其出度和入度都将增加 1。

算法 1 控制流图投影算法

```
输入:G,N*,N=Red_Black_Tree,E=Map(N)
输出:G*,E=Map(N*)
Step1(以 N 为节点,按照语句执行顺序建立最小堆)
  DEFINE Red_Black_Tree(N);
INIT Red_Black_Tree(N);
  CREATE Red_Black_Tree(N);
  (建立集合 Map(N),集合中存储 N 中所有节点与其后继节点
  Γ(N)的映射)
  ∀n∈N,Map(n)←(n,Γ(n));
Step2(折叠通路)
  WHILE(Red_Black_Tree.hasNext)
  DO BEGIN
    IF(n.ind≥2 ‖ n.outd≥2) //即 n 属于 B
    THEN [
      IF(A.a∈Γ(n) ‖ F.f∈Γ(n))
      THEN [Γ(n)←A.a ‖ Γ(n)←F.f]
      ]
    ELSE
    THEN [Γ(n)←(n∪Γ(n)∩N*)]
    END
Step3(调整红黑树与映射集合,得出最终的投影 G*)
  DELETE n (n∉N*);
  Red_Black_Tree(N) ← Red_Black_Tree(N*)
  Map(N) ← Map(N*);
```

算法 1 的复杂度分析:Step1,Step2 和 Step 中的时间复杂度均为 $O(\log(N))$,空间复杂度均为 $O(N)$。因此,该算法时间复杂度为 $O(\log(N))$,空间复杂度为 $O(N)$。

$G^*=P[G,N^*]$是程序控制流图 G 的投影。对于 G^*,给出以下两个定义。

定义 2(闭合节点) 对于 $n\in N^*$ 且 $n\notin\Gamma(n)$,那么节点 n 为闭合节点。

定义 3(直接前驱节点) 从后向前的第一个非闭合节点称为节点 n 的直接前驱节点。

投影过程的输出会作为检测过程的输入。在检测过程中,本文采用栈消除的思想。以 A 或 F 中的节点作为栈中的基准元素 *base*,查找 *base* 的直接前驱节点。如果 *base* 的直接前驱节点为 B,那么判定为疑似泄漏。查找结束后,检查栈是否为空,如果不为空则判定为内存泄漏。其过程如下。

过程 1 内存泄漏检测过程

```
DEFINE Stack stack;
DEFINE Node base,current;
Bool MemLeak_Check(Tree Red_Black_Tree,Set Map){
WHILE (Red_Black_Tree.hasNext)
DO BEGIN
  base←null;
  current=Red_Black_Tree.current_Ele;
  IF(base==null)
  THEN[
    IF(base !=A && base ! =F)
    THEN [CONTINUE];
    ELSE THEN[
      base← current;
      stack.push(base);]
    ]
  ELSE THEN[
    IF(current ==B)
    THEN[
      IF(B.ind < 2 ‖ B.outd < 2)
          THEN [RETURN false;]
      ]
      ELSE IF(current ==base)
      THEN [stack.push(current);]
      ELSE IF(current ! =base)
      THEN [stack.pop();]
  ]
END
  IF(stack!=null)
  THEN [RETURN false;]
  ELSE THEN
  [RETURN true;]
}
```

过程 1 的复杂度分析:该检测过程共循环一次,即对红黑树的遍历。因此,该过程时间复杂度为 $O(\log(N))$,空间复杂度为 $O(N)$。

3.1.3　函数内分析方法评价及补充

该方法利用图的概念对程序控制流图进行投影,使得分析过程简单而直观。但由于在控制流分支的判断中没有对谓词条件的值做具体分析,导致对数据流的敏感度低,所以该方法可能会存在较高的误报。因此,该算法在执行过程中,可以结合符号执行和约束求解来提高分析结果的精度。本文使用 SAT4j 约束求解器,它是一个开源的布尔可满足性问题的约束求解器,它在符号执行过程中将问题转化为合取范式,最后求出满足路径条件的布尔值。代码如下。

```
S1:int * p;
S2:p=(int * )malloc(sizeof(10 * int));
S3:p=10;
S4:if(p!=null)
S5:  free(p);
```

在上述代码中,在分析时 S4 会被标识为 B(控制流分支语句),按照简单的投影检测方法,该段代码会被判定为疑似内存泄漏。但在对 S4 的条件谓词(p!=null)进行分析时,该条件判定结果为真,即内存被分配后,使用 *free* 函数进行了释放。该段代码对内存的管理比较规范,因此,符号执行和约束求解减少了误报。

3.2　函数间内存泄漏分析

在程序的静态分析中,对于复杂的源程序代码,函数间的分析需要建立一个统一而又实用的分析模型。在函数间的分析中,本文融合了 Cloning Expands the ICFG(Interprocedural Control Flow Graph)和 Expanded Supergraph 两种方法[17-18],通过记录每个调用点处内存指针的值,并将相关的调用点的调用主体克隆,构建了一个 IMCFG。IMCFG 分析顺序为由底至顶,即被调用的函数被总是先于调用的函数被处理,分析方法如内存泄漏在函数内的分析。

3.2.1　传统方法分析

在传统 Cloning 方法中,对每个调用点的调用主体进行复制,并在主体内单独传播信息,即构建一个函数间控制流图;同时,调用主体与内联的调用点相对应。而 Cloning Expands the ICFG 则是在此基础上根据调用嵌套的深度扩大控制流图。其中,调用嵌套的深度可使用快速排序的思想来求,具体如图 9 所示。但这种方法对于调用嵌套深度指数较大的情况下会导致代码信息的爆炸。

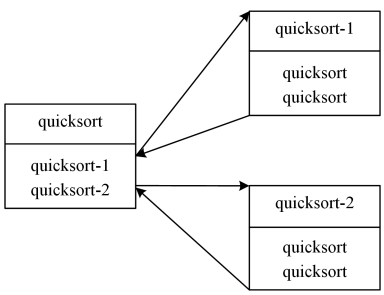


图 9　调用嵌套的深度计算

在传统 Supergraph 方法中,在分析时通过每次调用的 call 和 return 将过程内单独的控制流图连接成整个程序的控制流图。而 Expanded Supergraph 是复制每个调用点的抽象值,并将值的正确形式连接到调用点。即连接时考虑了调用的深度,其深度可通过调用串的长度来表示。每个调用点的抽象值可以表示为:

$$mult(P_i)=k_i * n$$

其中,P_i 表示第 i 个被调用的过程,k_i 代表第 i 个调用串,n 代表调用串的长度,即调用的深度。

3.2.2　函数间分析流程

图 10 为函数间分析流程图。在函数调用点处分析其参数及返回值,如果该调用过程中包含堆内存指针的修改,那么记录该调用点的抽象值 $mult(P_i)$,然后进行函数内的分析;如果不包含,则直接进行函数内分析。分析后记录该函数的分析结果,然后将分析结果及抽象值(如果记录了抽象值)与对应的调用点映射。

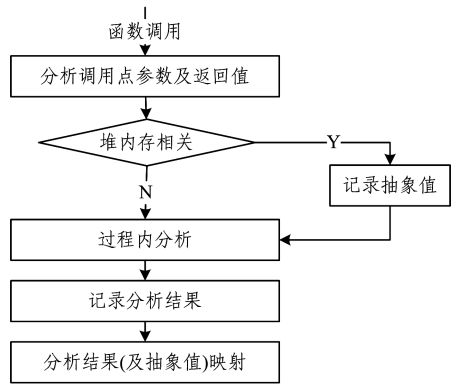


图 10　函数间分析流程图

在 IMCFG 的构建过程中,程序将复制带有抽象值的函数主体。该函数连接了与内存的分配与释放相关的函数体,并形成了一个函数间的控制流图,这样大大减少了复制的函数主体的数量,同时也减小了部分调用嵌套的深度,提升了函数间分析的效率。

4　实验分析

为了验证本文方法的有效性和精度,将提出的投影模型检测方法与另外几种常用的程序检测方法进行对比,其中包括类型推导检测工具 CppCheck[19]、基于程序风格的检测工具 Splint[20]、基于资源精简切片构建方法的 RL_Detector[21]。

首先,给出相关术语与指标的定义。

(1)KLOC:所检测源代码的行数,以千行来计数;

(2)TC:所列出泄漏类型测试用例的数量总和;

(3)HC:TC 中所检测出的内存泄漏数量;

(4)TW:报告出的内存泄漏个数的总和;

(5)TL:程序中包含的内存泄漏数量总和;

(6)FP:误报的漏洞数量;

(7)MLF:TW 与 FP 的差值,代表检测结果中的属于内存泄漏缺陷的数量;

(8)Cov_Rate:覆盖率,用来衡量有效性的指标,其计算公式为 $Cov\_Rate=HC/TC$;

(9)FP_Rate:误报率,用来测量精度的指标,其计算公式为 $FP\_Rate=FP/TW$;

(10)FN_Rate:漏报率,用来测量精度的指标,其计算公式为 $FN\_Rate=1-TW/TL$。

4.1　实验 1

该实验为第 3 节基于路径抽象的内存泄漏的每一种分类提供了测试用例,并将几种方法在同一环境下进行了检测,检测结果如表 1 所列。

表 1 4 种方法对路径抽象的内存泄漏测试用例的检测结果

检测方法	CppCheck	Splint	RL_Detector	本文方法(人工检测)
内存在条件分支中释放	YES	NO	YES	YES
内存在条件分支中申请	—	—	—	—
内存在循环体内释放	NO	NO	YES	YES
内存在循环体内申请	YES	NO	YES	YES
分支变量相同的连锁-分支结构	NO	NO	YES	YES
分支变量不同的连锁-分支结构	NO	NO	YES	YES
循环变量相同的连锁-循环结构	YES	YES	NO	YES
循环变量不同的连锁-循环结构	YES	YES	NO	YES
嵌套-分支结构中的内存泄漏	NO	NO	YES	YES
嵌套-循环结构中的内存泄漏	NO	NO	YES	YES

经实验分析,本文给出的所有内存泄漏分类中,第 2 种即内存在条件分支中申请的泄漏在编译环境(例如 Visual Studio)中会发生编译中断,因此该类型在静态检测中不作讨论。根据表中的结果比较,其中类型推导检测方法的检测覆盖率 Cov_Rate(CppCheck)为 5/9≈0.56,基于程序风格和注释的检测覆盖率 Cov_Rate(Splint)为 2/9≈0.22,资源精简切片构建方法的检测覆盖率 Cov_Rate(RL_Detector)为 7/9≈0.78,而本文针对复杂控制流提出的投影模型检测方法的检测覆盖率为 1。从实验 1 可以看出,针对复杂控制流的内存泄漏类型,投影模型检测方法的覆盖率高于其他方法,体现了该方法的有效性。

下面对其他 3 种检测方法做具体分析。CppCheck 将控制流中的节点按照一定的规则分组,按对应的规则进行检测,是控制流不敏感的分析,因此对复杂的控制流并不敏感;Splint 从改善程序风格和发现潜在的影响程序移植性错误的角度来提高软件质量,内存泄漏对于该方法的严重等级并不高,因此覆盖率较低;RL_Detector 针对 C 语言的内存泄漏进行检测,同时在函数间采用了上下文敏感的检测方法,但该方法对于循环内的内存申请和释放没有做详细讨论,因此在实验 1 给出的测试用例中没有覆盖连锁-循环结构。

4.2 实验 2

该实验使用的数据集为 SPEC CPU 2000 benchmark[22] 的 15 个 C 语言源代码,通过比较 4 种不同方法对同一测试集的测试结果的漏报率来验证投影模型的检测精度。具体实验结果如表 2 所列。

表 2 4 种检测方法对 SPEC2000 的测试结果

程序集名称	KLOC	资源精简切片构建(RL_Detector)		类型推导(CppCheck)		基于程序风格和注释的检测(Splint)		投影检测模型(人工检测)	
		TW	FP	TW	FP	TW	FP	TW	FP
164.gzip	7.8	1	1	1	1	1	1	3	1
175.vpr	17.0	0	0	0	0	0	0	4	1
176.gcc	205.8	35	0	1	0	46	24	22	1
177.mesa	49.7	4	2	1	0	9	5	17	5
179.art	1.3	1	0	1	0	0	0	3	0
181.mcf	1.9	0	0	0	0	5	2	1	1
183.equake	1.5	0	0	0	0	0	0	8	0
186.crafty	18.9	0	0	0	0	0	0	0	0
188.ammp	13.3	20	0	12	0	22	4	22	2
197.parser	10.9	0	0	0	0	0	0	0	0
253.perlbmk	58.2	4	1	2	1	0	0	2	0
254.gap	59.5	0	0	0	0	0	0	0	0
255.vortex	52.7	0	0	0	0	0	0	0	0
256.bzip2	4.6	0	0	1	0	2	1	1	0
300.twolf	19.7	0	0	0	0	0	0	0	0
合计	581	65	6	19	2	85	37	83	11

在 4 种检测方法对这 15 个 C 语言源程序文件的测试结果中,RL_Detector 共报告出 65 个漏洞,其中 6 个经人工检查后判定为误报,其误报率 FP_Rate(RL_Detector)为 6/65≈0.093;CppCheck 共报告出 19 个漏洞,其中 2 个经人工检查为误报,其误报率 FP_Rate(CppCheck)为 2/19≈0.11;Splint 共报告出 85 个漏洞,其中 37 个经人工检查为误报,其误报率 FP_Rate(Splint)为 37/85≈0.44;而本文提出的投影模型检测方法共报告出 83 个漏洞,其中 11 个经人工检测为误报,其误报率为 11/83≈0.22。从 TW 值即各个检测方法所报告的漏洞总数来看,Splint 报告的漏洞数量最多,本文的检测方法其次,而 CppCheck 报告的漏洞数量最少;但从误报率来看,Splint 的误报率最高,而其他 3 种方法检测出的误报率在可接受范围内。

除此之外,可以通过比较检测结果的 MLF 值来判断投影模型检测方法的有效性。表 2 可以抽象为 15 个四元组的稀疏矩阵,对其进行压缩。具体操作为:对每个程序集求各个方法的 MLF,然后得到 15 个四元组,忽略掉值为 0 的四元组,得到最终矩阵。

$$D=\begin{bmatrix}0&0&35&2&1&0&0&20&3&0\\0&0&1&1&1&0&0&12&1&1\\0&0&22&4&0&3&0&18&0&1\\2&3&21&12&3&0&8&20&2&1\end{bmatrix}^{\mathrm{T}}$$

图 11 中的横坐标为矩阵中每一行所对应的程序集名称,纵坐标为对应 MLF 值。该图反映了各个检测方法对 SPEC2000 的 15 个程序集的检测结果中,经人工检查后确认的内存泄漏个数。从图 11 中可以看出,除了 176.gcc,对于其他测试集,投影模型检测法的检测效果均较好。而 176.gcc 相对其他程序而言,代码量太大。因此,投影模型检测方法对

于规模较大的测试对象的可伸缩性稍差一些。

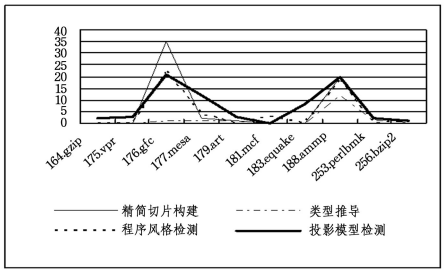


图 11　各个方法的 MLF 值的比较

综上，可以得出结论：针对复杂控制流的投影模型检测方法，对内泄漏的检测是有效的。同时，该方法在检测小规模程序时，精度较高，但对大规模程序检测的可伸缩性较差。鉴于此，我们将进一步完善算法，同时增强程序中对数据流的分析。

结束语　本文根据 C/C++语言中的路径特点，提供了一种基于路径抽象的内存泄漏分类。分类的主要对象为复杂控制流下的内存泄漏检测。在分类的基础上，提出了投影模型的内存泄漏检测方法；同时，在函数间分析中，结合了 Cloning Expands the ICFG 和 Expanded Supergraphs 两种方法，构造了 IMCFG，从而在内存泄漏的分析中，简化了一些无关分析；最后，通过两组实验，验证了投影模型检测方法的有效性和准确性。但对于规模较大的源代码，该方法还有待改善，其可伸缩性有待提高。

参 考 文 献

[1] United States Computer Emergency Readiness Team (US-CERT) [OL]. https://www.us-cert.gov/ 2016,7,5

[2] cmm5 定义 bug 等级[OL]. http://www.docin.com/p-1535244861.html.

[3] LEAK M. [OL]. http://www.baike.com/wiki/%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F.

[4] 王喆. C/C++代码内存泄漏缺陷检测方法研究[D]. 大连：大连理工大学，2012.

[5] LI M C, CHEN Y J, WANG L Z, et al. Dynamically validating static memory leak warnings[C]// International Symposium on Software Testing and Analysis Conference. Lugano, Switzerland, 2013:112-122

[6] LEE S H, JUNG C H, RAMAN E, et al. Automated memory leak detection for production use[C]// International Conference on Software Engineering. Hyderabad, India, 2014:825-836.

[7] 杨宇，张健. 程序静态分析技术与工具[J]. 计算机科学，2004，31(2)：171-174.

[8] SOR V, TREIER T, Srirama S N. Improving statistical approach for memory leak detection using machine learning[C]// International Conference on Software Maintenance. Eindhoven, Netherlands, 2013:22-28.

[9] SUI Y L, YE D, XUE J L. Detecting Memory Leakd Statically with Full-Sparse Value-Flow Analysis[J]. IEEE Transactions on Software Engineering, 2014, 40(2):107-122.

[10] LIM W, PARK S, HAN H. Memory leak detection with context awareness[C]// Reliable and Autonomous Computational Science Conference. San Antonio, USA, 2012:276-281.

[11] JOY M M, MUELLER W, RAMMIG F J. Source code annotated memory leak detection for soft real time embedded systems with resource constraints[C]// International Conference on Dependable, Autonomic and Secure Computing. Dalian, China, 2014:166-172.

[12] KANVAR V, KHEDKER U P. Heap abstractions for static analysis[J]. ACM Computing Surveys, 2016, 49(2):1-29.

[13] 付晓毓，朱利，顾伟. 基于模型检测的内存泄露静态测试方法[J]. 微电子学与计算机，2010，27(10)：170-173.

[14] XU Z B, ZHANG J, XU Z X. Memory leak detection based on memory state transition Graph[C]// Asia-Pacific Software Engineering Conference. Ho Chi Minh, Vietnam, 2011:33-40.

[15] 张仕金. 基于 Cppcheck 软件缺陷模式的研究与定位[D]. 重庆：重庆大学，2014.

[16] 侯洛明. 程序静态分析的通用方法[J]. 计算机学报，1987，2：74-81.

[17] ALT M, MARTIN F. Lecture Notes in Computer Science[M]. US, Berlin: Springer, 2005.

[18] 李平华. 过程间数据流分析技术研究[D]. 南京：东南大学，2004.

[19] CppCheck[OL]. trac.cppcheck.net/wiki.

[20] Splint[OL]. http://www.splint.org/.

[21] 姬秀娟. 资源泄漏故障静态分析的关键技术的研究[D]. 天津：南开大学计算机与控制工程学院，2014.

[22] Standard Performance Evaluation Corporation(SPEC)[OL]. http://www.spec.org/cpu/.