软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(2):357−372 [doi: 10.13328/j.cnki.jos.004532]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于区域内存模型的 C 程序静态分析*

董玉坤[1,2], 金大海[1], 宫云战[1], 邢 颖[1]

[1](网络与交换技术国家重点实验室(北京邮电大学),北京 100876)

[2](中国石油大学(华东) 计算机与通信工程学院,山东 青岛 266580)

通讯作者: 董玉坤, E-mail: dongyk@upc.edu.cn

摘 要: 为了提高程序的静态分析精度,提出了一种应用基于区域的符号化三值逻辑(region-based symbolic three-valued logic,简称 RSTVL)的静态分析方法.RSTVL 能够描述 C 程序运行时内存中数据结构的形态信息与变量的存储状态,以及可寻址表达式间的各种关系,包括指向关系、层次关系与取值逻辑关系.为了提高静态分析的精度,提出了一种基于 RSTVL 的流敏感、域敏感的过程内分析与基于符号化函数摘要的上下文敏感的过程间分析,能够精确地分析出每个程序点上的形态信息、数据流信息与指针指向关系.实验结果表明,相对于基于符号化三值逻辑的方法,该分析方法在保证一定分析效率的前提下,能够实现较高准确度的分析.

关键词: 可寻址表达式;内存模型;静态分析;符号化函数摘要;缺陷检测

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 董玉坤,金大海,宫云战,邢颖.基于区域内存模型的 C 程序静态分析.软件学报,2014,25(2):357−372. http://www.jos.org.cn/1000-9825/4532.htm

英文引用格式: Dong YK, Jin DH, Gong YZ, Xing Y. Static analysis of C programs via region-based memory model. Ruan Jian Xue Bao/Journal of Software, 2014,25(2):357−372 (in Chinese). http://www.jos.org.cn/1000-9825/4532.htm

Static Analysis of C Programs via Region-Based Memory Model

DONG Yu-Kun[1,2], JIN Da-Hai[1], GONG Yun-Zhan[1], XING Ying[1]

[1](State Key Laboratory of Networking and Switching Technology (Beijing University of Posts and Telecommunications), Beijing 100876, China)

[2](College of Computer and Communication Engineering, China University of Petroleum, Qingdao 266580, China)

Corresponding author: DONG Yu-Kun, E-mail: dongyk@upc.edu.cn

Abstract: In order to improve the precision of static analysis for C procedures, this paper introduces a static analysis method applying region-based symbolic three-valued logic (RSTVL). RSTVL can describe shape of data structures, all kinds of memory states and relations of addressable expressions including alias relations, hierarchical relations and logic relations. To improve precision, a RSTVL-based analysis method is proposed to analyze the shape, dataflow and point-to relationship at every procedure point. The method facilitates flow-sensitive and field-sensitive intra-procedure, and context-sensitive inter-procedure analysis based on symbolic function summary. Experimental results validate that the porposed static analysis method offers higher precision on the precondition with no efficiency loss.

Key words: addressable expression; memory model; static analysis; symbolic function summary; defect detection

C 语言能够作为嵌入式软件开发的主导语言,主要是因为 C 语言作为高级语言提供了指针与位操作,又具有可以直接对硬件进行操作等低级语言的特性,但这些特性难免也会造成程序中存在一些代码级别的缺陷.代码级别缺陷会导致运行时程序状态违背程序安全属性的要求,例如数组越界、空指针引用、非法计算、缓冲区

* 基金项目: 国家自然科学基金(91318301, 61202080); 国家高技术研究发展计划(863)(2012AA011201)
收稿时间: 2013-05-06; 修改时间: 2013-09-29; 定稿时间: 2013-12-05

溢出等,引发不可预测的异常.为了检测程序的代码级别的缺陷,通常进行静态分析,静态地获得程序运行时状态,判断运行时状态是否满足程序安全属性从而判断是否存在缺陷.但静态分析无法分析程序的所有非平凡属性,只能获得近似的结果,而静态分析的精度决定着缺陷检测结果的准确度.尽管过去的研究提出了一些静态分析的方法并开发了相应的工具,但依然有两个核心问题影响分析的精度:一是如何准确地表示内存对象的存储以及全面描述变量间的各种关联;二是如何用抽象的方法精确地处理各种程序语义.

C 程序包含指针、结构体、数组等复杂数据类型,引发可寻址表达式[1]间主要有指向关系、层次关系、取值逻辑关联这 3 种关系.全面地静态分析需要考虑分析这 3 种关系,否则会导致分析精度的下降.对 C 程序的分析研究主要有数据流分析、指针分析、形态分析.数据流分析主要用于分析程序运行时变量的取值情况;指针分析用于分析程序运行时的指针指向信息;形态分析主要用于判断程序运行时内存中数据结构的形态信息,能够精确地分析出变量间的指向关系与层次关系.为了全面地表示程序中可寻址表达式间的 3 种关联关系,首先需要以某种方式对内存存储进行抽象.基于不同的静态分析目的与精度的需要,一种方法是只描述内存对象间的一部分关系,例如,大数组模型[2]难以描述内存对象间的层次关系,形态图[3]与 TVLA[4]未考虑内存单元上存储信息间的关联,指向图[5]只描述指向关系,基于区域的模型[6]是未考虑取值的逻辑关联,STVL[7]考虑了取值的逻辑关联但不能描述复合数据类型的层次关系;另一种方法是对被测程序进行限制[8],例如,要求被分析程序没有动态内存分配,并且是完全的类型安全.

为了保证静态分析的精度,需要进行敏感的分析.例如,流敏感[9]考虑语句的执行顺序,域敏感[10]考虑复合数据类型与成员的关系,上下文敏感[5]考虑函数调用上下文信息,路径敏感[11]考虑路径条件.敏感的分析方法会提高分析的精度,但分析的代价随之升高,分析的效率也随之下降.为了平衡分析的精度与效率,大多数静态分析方法是流敏感与上下文敏感的.

因为静态分析的两个核心问题的困扰,导致目前对 C 程序的静态分析主要存在 3 点不足:(1) 未能全面地描述变量的存储状态;(2) 对域敏感分析考虑得不够,导致对复合数据结构的处理不够精确;(3) 未将数据流分析与指针分析、形态分析有效地结合起来.

鉴于上述问题,本文试图从 3 个方面进一步提高静态分析的精度:一是给出一个能够更准确地描述运行时程序状态的内存模型;二是有机地结合各种敏感分析方法;三是在一个统一的框架下进行数据流分析、指针分析与形态分析.为此,本文提出了一种应用 RSTVL(region-based symbolic three-valued logic)的 C 程序静态分析方法.RSTVL 能够全面地描述内存对象的存储状态以及可寻址表达式间的各种关联,基于 RSTVL 的分析将对变量的操作映射到对区域的操作.基于 RSTVL 的过程内分析是流敏感与域敏感的,过程间分析是上下文敏感与域敏感的.

本文第 1 节介绍 RSTVL.第 2 节介绍基于 RSTVL 的过程内分析.第 3 节介绍基于 RSTVL 的过程间分析.第 4 节通过实验对本文方法进行评估.第 5 节介绍相关工作.第 6 节对全文进行总结.

1 基于区域的符号化三值逻辑

静态分析的主要任务是分析程序的具体语义,即,获得程序的存储状态.存储状态是内存对象在程序点上状态的集合,存储状态包括左值表示的内存单元的地址信息与右值表示的值信息.

一个程序 P 可表示为六元组($V,L,S,\tau,init,end$),其中,$V\in Vars$,为程序 P 的变量集合;L 为程序点集合;$S\in Stmts$,为程序语句集合;$\tau\in L\times S\times L$,表示迁移关系;$init\in L$,为程序 P 的起点;$end\in L$,为程序 P 的终点.Worklist[12]算法作为迭代算法的一种实现,从程序的起点 $init$ 开始,基于迁移关系 τ,分析出在每个程序点 $l\in L$ 上的变量 $v\in V$ 的状态信息,直至程序的终点 end 结束.如果有一种静态分析方法 A 对一种语言 L 开发的任一程序 P,如果在 P 的任一程序点 l 上计算出的内存单元取值都包括实际运行时可能的取值,则称 A 为可靠的.

C 程序包含指针、结构体、数组等复杂数据类型,可寻址表达式间可能存在指向关系、层次关系、取值逻辑关联等多种关系,而且提供了指针算术、动态内存分配、强制类型转换等操作,这都增加了对 C 程序进行静态分析的难度.

1.1 引 例

图1为3个C程序的例子.对于程序片断图1(a),L9语句处的指针引用表达式*$pst[i]\rightarrow m$的引用路径上还隐含着3个表达式:pst,$pst[i]$,$pst[i]\rightarrow m$;而L7语句处的表达式$pst[i]$与L9处的表达式$pst[i]$虽然名称一样,但其实是两个不同的表达式.

对于程序片断图1(b),当L6语句处的取地址操作执行完以后,L7语句处的$sp\rightarrow p\rightarrow d$与$data.d$具有别名关系,导致L7语句处的赋值操作后$data.d$的值也为−10.而经过L9的分支语句,L10语句处的j的值为1或0,若采用路径不敏感的分析,则被赋值的$a[j]$可能是$a[1]$,也可能是$a[0]$,只能进行弱更新操作.

对于程序片断图1(c),L10语句处的实参s将会受到被调用函数f_3的副作用影响,$s.a$的指向信息将会发生改变.而调用函数f_3的返回值也由传递的参数决定,需要分析实参与形参的对应关系才能获得更精确的结果.

```
L1: struct st {
L2:     int *m;
L3: } **pst;
L4: void f1(struct st **sst){
L5:     int i=0;
L6:     pst=sst;
L7:     pst[i]→m=&i;
L8:     for (; i<9; i++){
L9:         int j=*pst[i]→m;
L10:    }
L11: }
```

(a)

```
L1: struct s1 {    int d;    }data;
L2: struct s2 {    struct s1 *p;    }*sp;
L3: void f2(int i) {
L4:     int j, a[6]={0,0,0,0,0,0};
L5:     sp=malloc(sizeof(struct s2));
L6:     sp→p=&data;
L7:     sp→p→d=−10;
L8:     for (j=0; j<6; j++)    a[j]=j;
L9:     if (i>0) i=1; else i=0;
L10:    a[i]=data.d;
L11: }
```

(b)

```
L1: typedef struct{ int *a; }str;
L2: int f3(str *ps, int **p){
L3:     ps→a=*p;
L4:     return **p+2;
L5: }
L6: void f4(·){
L7:     int x=1;
L8:     int *x0=&x;
L9:     str s;
L10:    x=f3(&s,&x0);
L11: }
```

(c)

Fig.1 Motivating examples

图1 程序示例

由图1的3个程序片断可以看出:C程序中具有复杂的表达式,而且这些表达式间可能存在多种关联,函数调用更加剧了分析的复杂性.为了实现对C程序较精确的分析,静态分析方法需要能够对C语言的各种类型的表达式进行准确的描述,并能对程序运行时的内存状态进行描述,以获得相关变量间的各种关联,对函数调用需要考虑调用点的上下文环境.

1.2 可寻址表达式

在程序中,表达式是一个由操作符与操作数组合的序列,规定了对一个值的计算.表达式的值有左值与右值之分,表达式的左值是指它代表的内存对象的地址,表达式的右值是指对表达式求值所得到的值.

定义1(内存对象). 程序运行时所分配内存对应的表达式,包括顶级变量v、复合类型内存对象的成员、动态分配的内存块.C99所支持的内存对象var的语法可归纳为

$$var::=v|var.f|var[n]|malloc(exp),$$

其中,v为顶级变量,$\underline{exp}$表示参数.

定义2(可寻址表达式). 内存对象与具有左值且可被赋值的表达式.

对可寻址表达式进行操作,将影响内存单元的存储状态.C99所支持的可寻址表达式e的语法可归纳为:

- $e::=var|e.f|e\rightarrow f|e[exp]|(e)|*e|m(\underline{exp})$,其中,$m$为函数指针,$\underline{exp}$表示参数;
- $*e::=*e'|*(++e')|*(--e')|*(e'++)|*(e'--)|*(e'\ \mathrm{op}\ exp')$,其中,$e'$为指针,op=+|−,$exp'$为整型表达式.

C程序中的数据类型可分为标量类型与复合类型:标量类型包括基本类型、指针类型;复合类型包括数组、结构体、联合.因为联合已较少使用,本文不再对联合类型进行分析.下面给出父变量的概念.

定义3(父变量). 复合类型变量为其成员的父变量.e为$e.f$,$e[exp]$的父变量.

虽然每个可寻址表达式在运行时会对应一块内存区域,但是某些可寻址表达式在不同的执行上下文环境下,它们所对应的内存区域可能会不一样.例如程序片断图1(a)中的L9语句处的$pst[i]$.程序运行时只为内存对

象分配区域,内存对象包括声明的顶级变量、复合类型变量的成员、动态分配的内存块等.在程序片断图 1(a)中的可寻址表达式中,*sst*,*i*,*pst*,*j* 等是内存对象,但**sst*,***sst*,*pst*[*i*]→*m*,**pst*[*i*]→*m* 等不是内存对象.

可寻址表达式因为内存单元间的左值或右值间的关联而存在以下 3 种关联:

(1) 左值与右值间的关联,称为指向关系,是由指针与所指向的变量产生的关系.指向关系会引发多个可寻址表达式对应同样一块内存区域,这些可寻址表达式之间具有别名关系.

(2) 左值间的关联,称为层次关系,是指复合类型可寻址表达式与其成员的关系.

(3) 右值间的关联,称为取值逻辑关系,是指基本类型的可寻址表达式在取值上的线形关系或逻辑关系.

1.3 基于区域的符号化三值逻辑

要全面并精确地分析出所有内存对象的存储状态,需要考虑可寻址表达式间的各种关联.文献[6]构建了基于区域的三元模型⟨*Var*,*Region*,*Value*⟩,考虑了指向关系与层次关系,但 *Value* 只能表示具体的值或区域编号,未考虑取值逻辑关系,而且该模型只适用于单路径的分析.文献[7]的 STVL 是一个三元模型⟨*Var*,S_{Exp},*Domain*⟩,考虑了取值逻辑关系,但未能很好地处理层次关系与指向关系.本文结合文献[6]基于区域的三元模型与文献[7]的 STVL 的优点,提出了基于区域的符号化三值逻辑——一个基于区域的内存模型.

定义 4. 基于区域的符号化三值逻辑 RSTVL 定义为四元组,*RSTVL*=⟨*Var*,*Region*,S_{Exp},*Domain*⟩,其中,*Var* 表示内存对象,*Region* 表示区域,S_{Exp} 表示符号表达式,*Domain* 表示取值区间.

基于 RSTVL 的静态分析构成一个三值逻辑命题 $3v\ell$,即⟨*Var*,*Region*,S_{Exp},*Domain*⟩→{0,1,1/2},$3v\ell\to 0$ 表示 *Var* 取值为未知区间,$3v\ell\to 1$ 表示 *Var* 取值为具体区间值,$3v\ell\to 1/2$ 表示 *Var* 取值为符号值.

四元组 RSTVL 用来描述标量类型的内存对象.复合类型的内存对象可分解为标量类型成员的组合,用三元组⟨*Var*,*Region*,χ⟩表示.χ的含义由 *Var* 的类型决定:如果 *Var* 是数组类型,则χ是{⟨*i*,*Region*⟩},$i\in\mathbf{N}$ 是数组 *Var* 的下标;如果 *Var* 是结构体,则χ是{⟨*f*,*Region*⟩},*f* 是结构体 *Var* 的成员.

对不同类型的内存对象,RSTVL 用不同类型区域对其存储状态进行抽象描述:*PrimitiveRegion* 描述基本类型的内存对象,*PointerRegion* 描述指针,*ArrayRegion* 描述数组,*StructRegion* 描述结构体.每个区域都有唯一的编号:*PrimitiveRegion* 的编号形式为 *bm_i*($i\in\mathbf{N}$),*PointerRegion* 的编号形式为 *pm_i*,*ArrayRegion* 的编号形式为 *am_i*,*StructRegion* 的编号形式为 *sm_i*.对动态分配的无名内存,用 *mxm_i_n*(*x* 表示区域的类型,取值为“*b*”,“*p*”,“*a*”,“*s*”)编号的区域描述,其中,*n* 为该无名内存的字节数.空地址的区域编号为 *null*,野地址的区域编号为 *wild*.规定参数与全局变量的区域编号额外设置首字母“u”与“g”.

定义 5. 将对内存对象分配的区域称为安全区域,将动态分配的区域称为动态区域.对这两种区域统称为可操作区域.将 *null* 与 *wild* 标志的区域称为不可操作区域.

动态区域经过非空判断后变为安全区域,动态区域经过是空判断后变为不可操作区域.

定义 6. 符号表达式 S_{Exp} 由符号通过数学运算与关系操作构成,递归定义如下:

$$\begin{cases} S_{Exp}\to Rel_{Exp}\mid\neg Rel_{Exp}\mid Rel_{Exp}\ell S_{Exp},\ell\in\{\&\&,\|\} \\ Rel_{Exp}\to Exp\mid Exp\Re Rel_{Exp},\Re\in\{<,\leqslant,>,\geqslant,==,\neq\} \\ Exp\to Term\mid Term\pm Exp \\ Term\to Power\mid Power\times Term\mid Power\div Term \\ Power\to Factor\mid Factor^{Power} \\ Factor\to Constant\mid Symbol\mid(S_{Exp}) \end{cases},$$

其中,符号表达式 S_{Exp} 由逻辑表达式 Rel_{Exp} 通过关系操作构成;Rel_{Exp} 由数学表达式 *Exp* 通过逻辑操作构成;*Exp* 由项 *Term* 通过加减运算组成;*Term* 由多个因子 *Power* 通过乘除运算组成;每个 *Power* 由 1 个或多个原子 *Factor* 通过幂运算组成;原子 *Factor* 是符号表达式的最基本元素,它可以是一个数值常量 *Constant*、符号变量 *Symbol* 或者符号表达式,其中,每个 *Symbol* 均对应着一个可寻址表达式.

对于取值区间,采用区间抽象域[13]的方法,每个 *Symbol* 的取值用区间表示.区间分为数值型区间与指针型区间两大类,其中,指针型区间 *PointerDomain* 是指向集合 *PointTos* 标识所指向的区域,*PointTos* 的元素即为区域

的编号.RSTVL 的区间及其上的操作构成完备格$\langle L,\leqslant,\sqcup,\sqcap,\bot,\top\rangle$,其中,$\bot$为空集,数值型区间的$\top$为$[-\infty,+\infty]$,指针型区间的$\top$为空地址、野地址与所有可操作区域编号的并,$\sqcup$为集合的并运算,$\sqcap$为集合的交运算.虽然区间抽象域是非关系型的,但通过符号表达式可实现逻辑关联描述.基于 RSTVL 的静态分析可被映射为在格上的操作.

RSTVL 可描述可寻址表达式间的 3 种关联,可满足流敏感、域敏感、路径不敏感的静态分析.因为路径不敏感,导致一个可寻址表达式可能对应多个区域.

RSTVL 将内存划分为离散的区域,能够描述区域间的指向关系、层次关系以及取值的逻辑关联.基于 RSTVL 的区域抽象包括:

(1) 对每个程序点 l,R^l 表示在 l 处能够被访问的区域集合,$S^l=\langle s,domain\rangle$表示在 l 处使用的符号集合.

(2) 对每个程序点 l,有一个抽象存储: $\rho^l=(\rho_v^l,\rho_r^l,\rho_f^l)$. 其中,

- $\rho_v^l: Var\rightarrow R^l$,映射一个内存对象到一个区域;
- $\rho_r^l: R^l\rightarrow R^l$,表示区域间的指向关系;
- $\rho_f^l: (R^l\times F)\rightarrow R^l$,映射一个复合类型变量的成员到一个区域;

(3) *RSTVL* 描述的抽象存储的信息间关联可表示为一个四元组$\sigma=\langle\sigma_r,\sigma_s,\sigma_f,\sigma_d\rangle$,其中,

- $\sigma_r: V\rightarrow R$,表示可寻址表达式与区域的关系,是一个多对多的关联,一个可寻址表达式可能对应多个区域,一个区域也可能描述多个可寻址表达式的抽象存储;
- $\sigma_s: R\rightarrow S_{Exp}$,表示区域与符号表达式映射;
- $\sigma_f: S_{Exp}\rightarrow S$,表示符号表达式与符号的关系,是一个多对多的关联,一个符号表达式由若干个符号通过逻辑运算及数学运算构成;
- $\sigma_d: S\rightarrow D$,表示符号与区间的映射,每个符号都有一个区间.

对可寻址表达式的各种操作,都首先需要获得所对应的区域.定义 $R^l[\![e]\!]$表示在程序点 l 上,抽象存储ρ^l 中,可寻址表达式 e 对应的区域集合.下面给出获得各种类型的可寻址表达式对应的区域的策略:

- $R^l[\![var]\!]=\rho_v^l(var)$;
- $R^l[\![e.f]\!]=\bigcup\limits_{r\in R^l[e]}\rho_f^l(r,f)$;
- $R^l[\![e[i]]\!]=\bigcup\limits_{r\in R^l[e]}\rho_f^l(r,i)$;
- $R^l[\![*e]\!]=\bigcup\limits_{r\in R^l[e]}\rho_r^l(r)$;
- $R^l[\![(e)]\!]=R^l[\![e]\!]$;
- $R^l[\![e\rightarrow f]\!]=\bigcup\limits_{r\in R^l[e]}\left(\bigcup\limits_{r'\in\rho_r^l(r)}\rho_f^l(r',f)\right)$.

下面给出关于 RSTVL 的其他相关操作:

- $C^l[\![v]\!]$:在程序点 l 上,为内存对象 v 生成一个区域 r.若 v 为标量类型,为 v 生成一个符号 s,并根据 v 的数据类型为 s 指定一个初始区间(若 v 是基本类型,初始区间设置为$[-\infty,+\infty]$;若 v 是指针类型,指向集合为$\{wild\}$),v 的区域 r 的符号表达式 s_{exp} 即由单一符号 s 构成;若 v 为结构体或数组的成员,父变量是 f_v,建立 f_v 的区域与 r 的层次关系.
- $V_r^l[\![r]\!]$:在程序点 l 上,获得区域 r 的符号化表达式.每个 *PrimitiveRegion* 与 *PointerRegion* 类型的区域均有唯一的符号化表达式.
- $V_e^l[\![e]\!]$:在程序点 l 上,获得可寻址表达式 e 的符号化表达式,$V_e^l[\![e]\!]=\mathop{\|}\limits_{r\in R^l[\![e]\!]}V_r^l[\![r]\!]$(||表示或操作).
- $D_s^l[\![s]\!]$:在程序点 l 上,获得符号 s 的取值区间;

- $D_{se}^{l}[\![s_{Exp}]\!]$:在程序点 l 上,获得符号化表达式 s_{Exp} 的取值区间.根据 s_{Exp} 中每个符号的取值进行区间运算[13]得到.
- $D_{e}^{l}[\![e]\!]$:在程序点 l 上,获得标量类型的可寻址表达式 e 的取值区间. $D_{e}^{l}[\![e]\!]=\bigcup_{r\in R^{l}[\![e]\!]} D_{se}^{l}[\![V_{r}^{l}[\![r]\!]]\!]$.
- $R_{n}^{l}[\![name]\!]$:在程序点 l 上,获得编号为 $name$ 的区域.
- $S[\![S_{Exp}]\!]$:获得符号表达式 s_{Exp} 中出现的所有符号.
- $E_{r}[\![r]\!]$:获得区域 r 对应的内存对象表达式.
- $E_{s}[\![s]\!]$:获得符号 s 对应的可寻址表达式.
- $N[\![r]\!]$:获得区域 r 的编号.

2 过程内分析

本文的过程内分析采用经典的数据流迭代算法[12],基于控制流图进行流敏感的分析,对复合数据类型变量进行域敏感的分析.对控制流图上的每个节点 n,定义两个程序点:$\bullet n$ 表示 n 前的程序点,$n\bullet$表示 n 后的程序点.数据流计算方程如下:

$$in(n)=\begin{cases} init(n), & n=entry \\ \bigcup_{p\in pred(n)} out(p), & \text{otherwise} \end{cases} \tag{1}$$

$$out(n)=gen(n)\cup(in(n)-kill(n)) \tag{2}$$

其中,n 为控制流图当前节点,p 为 n 的前驱节点.公式(1)中 $in(n)$为计算的 n 的汇入信息,也就是程序点$\bullet n$ 的抽象存储 $R^{\bullet n}$; $init$ 表示对函数入口处的全局变量与参数进行初始化操作.公式(2)中 $out(n)$为计算的 n 的出口信息,也就是程序点 $n\bullet$的抽象存储;$gen(n)$表示 n 中新产生的数据流信息;$kill(n)$表示 n 中注销或被改变的数据流信息; $pred(n)$表示 n 的所有前驱节点集合.

相对于流不敏感分析,本文采用的流敏感分析能够实现强更新操作,$kill(n)$能够计算出更多的数据流信息.相对于域不敏感的分析,本文采用的域敏感分析将会更准确地计算复合类型变量及其成员的数据流信息,本文的 RSTVL 也能更全面地描述抽象存储,这都将保证本文的分析所获得的每个程序点上的抽象存储更接近真实运行时的内存状态.因为本文对变量的取值通过区间抽象域进行抽象表示,满足在格上的操作.例如,在进行循环处理时,可采用加宽/收窄算子,通过迭代快速地达到不动点.

2.1 赋值语句的迁移操作

程序语句中最核心的语句是赋值语句,C 程序中有各种类型的赋值语句,包括拷贝赋值($x=y$)、加载赋值($x=*y$,$x=y.f$,$x=y\rightarrow f$)、存储赋值($*x=y$,$x.d=y$,$x\rightarrow d=y$)、取地址赋值($x=\&y$)等各种基本赋值操作以及更加复杂的赋值操作.本文将赋值操作统一表示为 $e_l=e_r$,并在一个统一的分析框架进行分析.如果 e_r 是$\&e$、数组 arr、指针算术 $p+i$,其对应的区域分别表示为:

- $R^{l}[\![\&e]\!]=r$,其中, $D_{e}^{l}[\![E_{r}[\![r]\!]]\!]=\bigcup_{r'\in R^{l}(e)} N[\![r']\!]$;
- $R^{l}[\![arr]\!]=R^{l}[\![\&arr[0]]\!]$,其中,$arr$ 是数组;
- $R^{l}[\![p+i]\!]=\bigcup_{r\in R^{l}[\![p]\!]} \rho_{f}^{l}(r',j)$, 其中,$r'$是 $E_{r}[\![r]\!]$的父变量的区域,j 是 $E_{r}[\![r]\!]$下标值与 i 的和.

C 语言通过 $malloc$,$free$ 进行动态内存分配与释放的操作.对 $malloc$ 分配内存的赋值语句:

$$e_l=malloc(n),\ R^{l}[\![malloc(n)]\!]=C^{l}[\![v]\!],$$

其中,v 是临时变量,类型为指针 e_l 所指向的类型.当在程序点 l 上对动态分配的内存通过 $free$ 进行释放时,假设描述该动态内存的区域编号 mx,从 R^{l} 获得取值区间的指向集合 $PointTos$ 包含 mx 的 $PointerRegion$,将 mx 除去;如果指向集合 $PointTos$ 不包括 $wild$,则添加 $wild$,标识相应的指针指向一个野地址.

因为任何一个表达式都规定了对一个值的计算,每个表达式也都具有一种数据类型.如果一个表达式是指针、数组或结构体,则该表达式肯定为可寻址表达式,可在所在的程序点上基于RSTVL获得该表达式对应的区域.如果一个基本类型的表达式不是可寻址表达式,则本文直接用符号表达式表示该表达式的值.

假定赋值操作语句 $e_l=e_r$ 对应着控制流图的节点 n,分析时首先需要确定在程序点$\bullet n$ 上 e_l 对应的区域数目.如果 $R^n[\![e_l]\!]$是一个单例集,且为一个可操作的区域 r,则对区域 r 进行强更新操作.如果 $R^n[\![e_l]\!]$对应多个区域,则对每一个可操作的区域 r 进行弱更新操作.基于RSTVL的赋值语句的迁移操作如算法1所示.

算法 1. 赋值语句的迁移操作.

1. **if** $|R^{\bullet n}[\![e_l]\!]|==1$ && $r\in R^{\bullet n}[\![e_l]\!]$ **then**
2. $\quad V_r^{n\bullet}[\![r]\!]=\perp$;
3. **if** r is *PrimitiveRegion* or *PointerRegion* **then**
4. $\quad$ **for** each $r\in R^{\bullet n}[\![e_l]\!]$
5. $\quad\quad V_r^{n\bullet}[\![r]\!]=V_r^{n\bullet}[\![r]\!]\,||\,V_e^{\bullet n}[\![e_r]\!]$;
6. **else**
7. $\quad$ **for** each $r_1\in R^{\bullet n}[\![e_l]\!]$
8. $\quad\quad$ **for** each $r_2\in R^{\bullet n}[\![e_r]\!]$
9. $\quad\quad\quad$ *combine*(n,r_1,r_2);

作为被赋值的可寻址表达式,e_l 不可能是数组.辅助函数 *combine*(n,r_1,r_2),在控制流图节点 n 上,如果 r_1 是为标量类型分配的区域,则将区域 r_1 与区域 r_2 的值进行合并,合并后的值赋给区域 r_1.如果 r_1 是为复合类型分配的区域,则分解为标量类型再进行 *combine* 操作.

2.2 分支语句与合并语句的分析

如果条件判断语句节点为 n,则进入 n 的其中一个出边 e 的条件约束通过符号集$\zeta=\langle s,domain\rangle$描述,流向 e 的抽象存储表示为 $R^{n\bullet\mapsto e}$.$R^{n\bullet\mapsto e}$ 的运算规则如算法2所示.

算法 2. 分支语句迁移操作.

1. $R^{n\bullet\mapsto e}=R^{n\bullet}$;
2. **for** each $\langle s,domain\rangle\in\zeta$
3. $\quad$ update $\langle s,D_s^{n\bullet\mapsto e}[\![s]\!]\rangle$ with $\langle s,D_s^{n\bullet\mapsto e}[\![s]\!]\cap domain\rangle$ in $S^{n\bullet\mapsto e}$;

汇聚节点的抽象存储为各个入边流入的抽象存储的并.如果程序点 l_1 的抽象存储是程序点 l_2 与 l_3 的并,合并操作 $R^{l_1}=R^{l_2}\cup R^{l_3}$ 如算法3所示.

算法 3. 合并操作 $R^{l_1}=R^{l_2}\cup R^{l_3}$ 的迁移操作.

1. **for** each $r\in R^{l_2}$;
2. $\quad$ let $r_1=C^{l_1}[\![E_r[\![r]\!]]\!]$;
3. $\quad$ **if** $\exists r'\in R^{l_3}$ && $E_r[\![r]\!]==E_r[\![r']\!]$ **then**
4. $\quad\quad V_r^{l_1}[\![r_1]\!]=V_r^{l_2}[\![r]\!]\,||\,V_r^{l_3}[\![r']\!]$;
5. $\quad$ **else** $V_r^{l_1}[\![r_1]\!]=V_r^{l_2}[\![r]\!]$;
6. **for** each $r\in R^{l_3}$
7. $\quad$ **if** !$(\exists r'\in R^{l_2}$ && $E_r[\![r]\!]==E_r[\![r']\!])$ **then**
8. $\quad\quad$ let $r_1=C^{l_1}[\![E_r[\![r]\!]]\!]$;
9. $\quad\quad V_r^{l_1}[\![r_1]\!]=V_r^{l_3}[\![r]\!]$;

2.3 循环语句的分析

本文对循环语句的分析采用加宽/收窄算子理论[14,15].对循环语句的抽象存储迭代求精,直到达到循环内外的抽象存储的不动点.该技术通过加宽算子使得迭代运算快速收敛,得到精确解的一个上界,通过收窄算子使得加宽后的结果尽量逼近精确解.基于 RSTVL 的加宽/收窄算子运算规则见表 1.

Table 1 Widening/narrowing operator's rules

表 1 加宽/收窄算子运算规则

Domain	Widening operator	Narrowing operator
Numeric domain	$[a_1,b_1]\nabla[a_2,b_2]=[L_1,L_2]$ **if** ($a_2<a_1$) **then** $L_1=-\infty$; **else** $L_1=a_1$; **if** ($b_2>b_1$) **then** $L_2=+\infty$; **else** $L_2=b_1$;	$[a_1,b_1]\Delta[a_2,b_2]=[L_1,L_2]$ **if** ($a_1=-\infty$) **then** $L_1=a_2$; **else** L_1=MIN(a_1,a_2); **if** ($b_1=+\infty$) **then** $L_2=b_2$; **else** L_2=MAX(b_1,b_2);
Pointer domain	$pt_1\nabla pt_2=pt_3$ **if** $\exists r_{Name}\in pt_2$ && $r_{Name}\notin pt_1$ $pt_3=pt_1\cup${"*wild*"};	$pt_1\Delta pt_2=pt_3$ **if** "*wild*"$\in pt_1$ && "*wild*"$\notin pt_2$ delete "*wild*" from pt_1; $pt_3=pt_1\cup pt_2$;

假定一个循环将被执行若干次,循环块的直接前驱节点为 n_1,包含循环条件的循环头节点为 n_2,循环语句块真分支内的第 1 个节点为 n_3,循环语句块真分支内的最后一个节点为 n_4.n_1 与 n_4 都是 n_2 的前驱节点,每次迭代运算过程中通过 $R^{n_1\bullet}$ 与 $R^{n_4\bullet}$ 的并是否发生变化来判断抽象存储是否稳定.应用加宽/收窄算子理论的循环语句分析如算法 4 所示:先执行一遍循环体,再迭代执行加宽操作(第 4 行~第 7 行),最后迭代执行收窄操作(第 9 行~第 12 行).

算法 4. 循环语句的分析.

1. $R^{\bullet n_2}=R^{n_1\bullet}$;
2. $R^{\bullet n_3}=R^{n_2\bullet\mapsto T}$; analysis the body of the loop; $R^{\bullet n_2}=R^{n_4\bullet}\cup R^{n_1\bullet}$;
3. $R'=R^{\bullet n_2}$;
4. **while true**
5. $R^{\bullet n_3}=R^{n_2\bullet\mapsto T}$; analysis the body of the loop; $R^{\bullet n_2}=R^{n_4\bullet}\cup R^{n_1\bullet}$;
6. $R^{\bullet n_2}=R'\nabla R^{\bullet n_2}$;
7. **if** $R'\neq R^{\bullet n_2}$ **then** $R'=R^{\bullet n_2}$; **else break**;
8. $R'=R^{\bullet n_2}$;
9. **while true**
10. $R^{\bullet n_3}=R^{n_2\bullet\mapsto T}$; analysis the body of the loop; $R^{\bullet n_2}=R^{n_4\bullet}\cup R^{n_1\bullet}$;
11. $R^{\bullet n_2}=R'\Delta R^{\bullet n_2}$;
12. **if** $R'\neq R^{\bullet n_2}$ **then** $R'=R^{\bullet n_2}$; **else break**;

2.4 实例分析

考虑程序片断图 1(b),L5 处通过 *malloc* 动态分配了一块内存,根据被赋值对象 *sp* 的数据类型,生成了一个类型为 struct s_2 的临时变量 m_mVar_1,为其分配一个编号为"*msm*_8_4"的区域.指针 *sp* 对应的指针型区间的 *PointTos* 的值为"*msm*_8_4",表明 *sp* 指向了 m_mVar_1.

处理语句 L6 前,*data* 的区域的编号为"*sm*_2".分析 L6 的赋值语句时,对右边的取地址操作,首先生成一个类型为 struct s_1*类型的临时指针 m_mVar_2,为其分配一个编号为"*mpm*_9_10"的区域,并设置其对应的指针型区间的 *PointTos* 为{"*sm*_2"}.对于被赋值表达式 $sp\rightarrow p$,首先分析得到指针 *sp* 对应的指针型区间的 *PointTos*,*PointTos* 为单例集,只有 1 个元素"*msm*_8_4",即,m_mVar_1 的区域编号.因为采取"晚初始化"(lazy initialization)的处理方式,在 L5 处并未对 m_mVar_1 的成员 *p* 生成一个临时变量,而 L6 的被赋值变量为 $m\_mVar_1.p$,这种情况下,需要识别出可寻址表达式 $m\_mVar_1.p$,并为其分配一个编号为"*mpm*_10_4"的区域.因为 $sp\rightarrow p$ 只对应 1 个区域,对其进

行强更新操作,即,将编号是“*mpm*_10_4”的区域的符号化表达式,更新为编号是“*mpm*_9_10”的区域的符号化表达式.

对于循环语句L8,经过加宽/收窄的处理,循环体内 j 的取值区间为[0,5],数组 a 的所有成员的取值区间都被更新为[0,5],该区间是实际取值区间的上近似.处理语句 L10 时,因为本文采取的是路径不敏感的分析,i 的取值区间为[0,1],$a[i]$对应的区域可能是 $a[0]$的区域,也可能是 $a[1]$的区域,需要执行弱更新操作.即,$a[0]$,$a[1]$的值为其原来值与 $data.d$ 的值的并,均为$[-10,-10]\cup[0,5]$.

3 过程间分析

函数调用对调用点上下文抽象存储的影响可分为3种:(1) 函数副作用,函数调用对全局变量与指针类型参数抽象存储的更改;(2) 函数返回值;(3) 控制流中断.

同一函数在不同的调用点被调用,上下文环境不同,将导致函数返回结果及对上下文环境的影响不同.为适应不同上下文环境的函数调用,本文的过程间分析采用基于符号化函数摘要的方法[16,17],实现在不同上下文环境下对同一函数调用得到不同结果,达到上下文敏感分析.本文的符号化函数摘要的创建以及实例化将充分利用 RSTVL 的优点,主要关注函数副作用与返回值对调用点抽象存储的影响.

本文的符号化函数摘要的基本思想是:首先,对全局变量与形参进行初始化;然后,根据过程内分析的结果,计算返回语句的符号表达式;最后,将函数退出节点的各个前驱节点的全局变量及形参的抽象存储信息添加到函数摘要中.符号化的函数摘要存放于一个全局环境中,在每个函数调用点,取出被调用函数的符号化函数摘要,并根据调用点处的上下文信息对符号化函数摘要进行实例化,实现对调用点处的抽象存储更新.

3.1 符号化函数摘要生成

同一函数在不同的上下文环境下被调用,指针类型的实参与全局变量的指向信息可能不同.在对被调函数分析时,为了能使不同调用点的指针指向信息方便地映射到被调函数上,引入扩充变量[18]来抽象地表示指针类型的形参和全局指针表达式的指向信息.扩充变量的引入规则为:(1) 对于 n 级指针 p,扩充出$*p$,$**p$,$***p$,…共 n 个变量;(2) 对于结构体 s,如果对应的结构体类型有 $f_1,f_2,\ldots,f_n$ 共 n 个成员,则扩充出 $s.f_1,s.f_2,\ldots,s.f_n$ 共 n 个变量.例如,对程序片断图1(c)函数 f_3 的形参 ps,ps 为一级指针,引入扩充变量$*ps$,$*ps$ 为 str 类型可寻址表达式,再扩充出变量$(*ps).a$.对一个函数基于控制流图进行分析时,首先获得该函数的参数,基于定义-使用链获得该函数使用的全局变量,对指针类型的参数与全局变量引入扩充变量,对参数、该函数内使用的全局变量及生成的扩充变量,均创建一个区域.

定义 7. 基于 RSTVL 的符号化函数摘要 SPS_{RSTVL}(symbolic procedure summary using RSTVL),应用 RSTVL 描述一个函数的行为,共包括4类信息:

$$\begin{cases} VarS_{Exp}Set=\{\langle Var,S_{Exp}\rangle\} \\ RetS_{Exp}=S_{Exp} \\ SymbolDomainSet=\{\langle Symbol,Domain\rangle\} \\ RegionVarSet=\{\langle Region_{Name},Var\rangle\} \end{cases}.$$

对参数与全局变量及它们的扩充变量初始化之后,便进行过程内分析.根据过程内分析的结果,计算返回语句的符号表达式,并将函数退出节点的各个前驱节点的全局变量及形参的符号化表达式添加到函数摘要中.生成的函数摘要将作为函数的一个属性.算法5是生成函数摘要的算法.

算法 5. 函数摘要生成算法.

1. $S_{Exp}List=\varnothing$; $SExp_{Ret}=\perp$; $VarS_{Exp}Set=\varnothing$; $SymbolDomainSet=\varnothing$; $RegionVarSet=\varnothing$;
2. **let** V_{Set} as the set of global variables, parameters and extended variables
3. **for** each $n_{Ret}\in pre(G_{exit})$
4. $\quad RetS_{Exp}=RetS_{Exp}\|getS_{Exp}(n_{Ret})$;
5. $S_{Exp}List=\{RetS_{Exp}\}$;

6. **for** each $v \in V_{Set}$
7. **if** $V_e^{G_{exit}\bullet}[\![v]\!] \neq V_e^{\bullet G_{entry}}[\![v]\!]$ **then**
8. $VarS_{Exp}Set=VarS_{Exp}Set \cup \{\langle v, V_e^{G_{exit}\bullet}[\![v]\!]\rangle\}$;
9. $S_{Exp}List=S_{Exp}List \cup \{V_e^{G_{exit}\bullet}[\![v]\!]\}$;
10. **for** each $s \in S_{Exp}List$
11. $SymbolDomainSet=SymbolDomainSet \cup \{\langle s, D_s^{G_{exit}\bullet}[\![s]\!]\rangle\}$;
12. **if** $D_s^{G_{exit}\bullet}[\![s]\!]$ is *PointerDomain* **then**
13. let *ptList* as *PointTos* of $D_s^{G_{exit}\bullet}[\![s]\!]$;
14. **for** each $r_{Name} \in ptList$
15. $RegionVarSet=RegionVarSet \cup \{\langle r_{Name}, E_r[\![R_n^{G_{exit}\bullet}[\![r_{Name}]\!]]\!]\rangle\}$;
16. add $S_{Ret}Set$, $VarS_{Exp}Set$, *SymbolDomainSet*, *RegionVarSet* to SPS_{RSTVL};

3.2 符号化函数摘要实例化

在函数调用点:首先,获得被调用函数的函数摘要;其次,基于形参与实参及其父变量的关系,获得每个形参对应的实参可寻址表达式集合;然后,将函数摘要中指针类型的区间进行实例化,将 *PointTos* 的区域编号映射为调用点处的区域编号;再将符号化表达式中的符号用调用点处的符号替代;对于被函数副作用影响的全局变量与实参,基于实例化后的符号表达式对其进行过程内的赋值语句分析;最后,基于实例化后的符号计算出函数返回值.

假定函数 p 在其控制流图节点 n 调用函数 q,即 $p \xrightarrow{n} q$, 函数摘要的实例化操作如算法 6 所示.其中,

- *mappingToArguments*(*var*):获得形参 *var* 与其对应的实参集合;
- *initializingPointerDomain*(*d*):将指针型区间 *d* 的 *PointTos* 的区域编号实例化为调用点处对应的区域编号;
- *fVarAVarsSet*(*fVarAVarsSet*,var_p):从描述形参与实参对应关系的 *fVarAVarsSet* 中获得 var_p 对应的实参集合.

算法 6. 函数摘要实例化算法.

1. $sps=getSummary(method)$;
2. get $VarS_{Exp}Set$, $RetS_{Exp}$, *SymbolDomainSet*, *RegionVarSet* from *sps*;
3. $fVarAVarsSet\langle var,\{\langle var\rangle\}\rangle=\varnothing$;
4. **foreach** $\langle var,S_{Exp}\rangle \in VarS_{Exp}Set$
5. $varsList=mappingToArguments(var)$;
6. $fVarAVarsSet=fVarAVarsSet \cup varsList$;
7. **foreach** $\langle s,d\rangle \in SymbolDomainSet$ and *d* is *PointerDomain*
8. *initializingPointerDomain*(*d*);
9. **foreach** $\langle var_p,S_{Exp}\rangle \in VarS_{Exp}Set$
10. $tempS_{Exp}=s_{Exp}$;
11. **foreach** $s \in S[\![s_{Exp}]\!]$
12. **let** $e=E_s[\![s]\!]$, $newS_{Exp}=\perp$;
13. **foreach** $r \in R^{\bullet n}[\![e]\!]$
14. $newS_{Exp}=newS_{Exp} \cup V_r^{\bullet n}[\![r]\!]$;
15. replace *s* in $tempS_{Exp}$ with $newS_{Exp}$;
16. $var_as=fVarAVarsSet(fVarAVarsSet,var_p)$;

17. **if** $|R^{\cdot n}[\![var_a s]\!]|==1$ **then** $strongUpdate(tempS_{Exp},var_a)$;
18. **else** $weakUpdate(tempS_{Exp},var_a)$;
19. **if** $SExp_{Ret}\neq\perp$ **then**
20. **foreach** $s\in S[\![SExp_{Ret}]\!]$
21. **let** $e=E_s[\![s]\!]$, $newS_{Exp}=\perp$;
22. **foreach** $r\in R^{\cdot n}[\![e]\!]$
23. $newS_{Exp}=newS_{Exp}\cup V_r^{\cdot n}[\![r]\!]$;
24. replace s in $SExp_{Ret}$ with $newS_{Exp}$;

3.3 实例分析

考虑程序片断图1(c),函数f_3的函数摘要中:

- $VarS_{Exp}Set$={⟨*ps*,*ps*_0⟩,⟨(**ps*).*a*,(**ps*).*a*_1⟩,⟨*(**ps*).*a*,*(**ps*).*a*_2⟩,⟨*p*,*p*_3⟩,⟨**p*,**p*_4⟩,⟨***p*,***p*_5⟩;
- $S_{Ret}Set$={⟨2+***p*_5⟩};
- *RegionVarSet*={⟨"*usm*_1",**ps*⟩,⟨"*upm*_4",**p*⟩,⟨"*ubm*_5",***p*⟩};
- *SymbolDomainSet*={⟨(**ps*).*a*_1,{"*usm*_5"}⟩,⟨**p*,{"*upm*_4"}⟩,⟨***p*,{"*ubm*_5"}⟩}.

对于函数调用 $f_4\xrightarrow{\text{L10}}f_3$, 形参 *ps*,(**ps*).*a*,*p*,**p*,***p* 分别对应着实参集合{$annoyPtr_0$},{*s*.*a*},{$annoyPtr_1$},{x_0},{*x*}.$annoyPtr_0$ 为生成的临时指针变量,指向 *s*;$annoyPtr_1$ 为生成的临时指针变量,指向 x_0.*SymbolDomainSet* 中的"*upm*_4","*ubm*_5"被实例化为"*bm*_2","*bm*_1",其中,"*bm*_2"是为 x_0 分配的区域编号,"*bm*_1"是为 *x* 分配的区域编号.而(**ps*).*a*_1 是为(**ps*).*a* 生成的符号,(**ps*).*a* 只对应着 *s*.*a*,则 *s*.*a* 对应的指针型区间的 *PointTos* 被更新为{"*bm*_1"},表明经过函数调用的副作用,*s*.*a* 指向了 *x*.对于 $S_{Ret}Set$ 中的***p*_5 是为***p* 生成的符号,***p* 只对应着 *x*,而 *x* 取值为常量1,则符号表达式 2+***p*_5 被实例化为3,即,函数返回值为[3,3].

4 实验结果及其分析

基于RSTVL的静态分析方法已经在缺陷检测工具DTSC中实现,DTSC的执行步骤包括:抽象语法树生成、符号识别、全局调用函数分析、生成控制流图、生成定义-使用链、数据流分析、缺陷检测.为了验证本文方法的效果,本文选取了5个C开源工程进行分析,并与STVL方法进行了对比.实验采用的硬件环境为:双核1.33G U5600处理器,3GB物理内存,Windows XP操作系统.

4.1 可寻址表达式间关联

通过本文方法识别出5个C工程的可寻址表达式及分析出的关联关系,见表2.从表2可以看出:将近50%的可寻址表达式存在层次关系,将近40%的可寻址表达式存在指向关系,将近10%的可寻址表达式间具有取值的逻辑关联关系.在进行静态分析时对这些关联关系考虑得不全面,势必导致分析精度的下降.

Table 2 Association of addressable expressions

表2 可寻址表达式间关联统计

Benchmark	File count	LOC	Addressable expression	Relations			
				Hierarchical	Point-To	Logic	Total
antiword-0.37	67	24 215	1 717	1 171	979	105	2 255
uucp-1.07	251	52 595	2 810	966	779	142	1 887
sphinxbase-0.3	62	22 517	972	451	430	201	1 082
optipng-0.6	68	27 075	979	471	355	189	1 015
barcode-0.98	14	3 409	300	187	126	24	337
Total	462	129 811	6 778	3 246	2 669	661	6 576

4.2 效率分析

为了验证分析方法的效率,通过实验与基于 STVL 的静态分析方法[7,17]的 DTSC 进行对比.因为 RSTVL 比 STVL 能够描述更多的关联信息,在分析时将不可避免地导致分析效率的下降.为此,本文选取了 5 个 C 开源工程进行分析,比较两种分析方法在进行数据流分析时分析时间的差异.分析效率结果对比见表 3.

Table 3 Efficiency comparision of two strategies

表 3 两种策略的效率对比

Benchmark	STVL		RSTVL		Time increment (%)
	Time (s)	Speed (L/s)	Time (s)	Speed (L/s)	
antiword-0.37	163	158	243	106	49
uucp-1.07	3 539	29	4 664	22	32
sphinxbase-0.3	232	91	289	73	25
optipng-0.6	355	76	492	55	39
barcode-0.98	67	133	75	119	12
Total	4 356	43	5763	32	32

由表 3 可见:STVL 方法的分析效率是 30 行代码/秒;RSTVL 方法的分析效率是 23 行代码/秒,即,万行代码大约需要 7 分钟.RSTVL 方法比 STVL 方法的分析效率略有下降,主要有 3 个方面的原因:

(1) RSTVL 采用的是四元组模型,比 STVL 增加了区域的概念,在对可寻址表达式进行分析时,映射到对应的区域上进行分析,增加了计算量.

(2) RSTVL 方法对指针类型与复合类型的参数与外部变量生成了扩充变量,导致参与运算的变量比 STVL 方法增多.

(3) 实现了域敏感的过程间分析,在函数摘要生成与实例化时都考虑了域成员.其中,工程 antiword-0.37 只有 30%的可寻址表达式是基本类型的内存对象,另 70%的可寻址表达式是指针、复合类型或复合类型的成员,分析时需要考虑区域间的关联耗时较多,从而导致基于 RSTVL 的分析比基于 STVL 的分析时间增加了 49%.

STVL 方法与 RSTVL 方法对循环均采取了加宽/收窄的分析策略,为了分析得到不动点,循环体将会分析若干次.因此,当被分析的程序循环过多,特别是嵌套循环过多时,将会严重影响分析的效率.uucp-0.98 工程的分析效率偏低,主要原因是个别文件具有复杂的循环,例如分析 uucp\uuconf\hsinfo.c 文件,STVL 方法耗时 968s,RSTVL 方法耗时 1342s.原因是*_uuconf_ihdb_system_internal* 方法内有多个循环,特别是行号在 111~356 间的循环,而且该循环又嵌套多个循环,需要经过若干次迭代才能达到数据流的稳定,严重影响了分析效率.

因为本文的分析是以预编译后的中间文件为分析单元,数据流分析是以函数为分析单元的,分析时间与被分析程序的规模具有线性关系,所以本文的方法也适用于对大规模程序的分析.

4.3 空指针引用检测

将本文静态分析的结果应用于空指针引用缺陷的检测.对于一个指针 p,当其在程序点 l 上被引用时,令 p 对应的 *PointTos* 为集合 $List_{pt}$.如果 $List_{pt}$ 只包括"*null*"一个元素,则判为肯定空指针引用;如果 $List_{pt}$ 的元素均为安全区域的编号,则为安全指针引用;其他情况判为可疑指针引用.空指针引用包括肯定空指针引用与可疑指针引用.对部分库函数,人工生成了摘要;对于没有函数摘要的函数,采取保守的策略,其返回值为抽象域的最大区间.表 4 为应用 STVL 方法与 RSTVL 方法检测空指针引用缺陷的结果,其中,缺陷检测点(inspection point,简称 IP)为经人工确认得到的有效缺陷(BUG).

经过人工判定,RSTVL 方法所检测出的 BUG 包括 STVL 方法所检测出的所有 BUG.因为 RSTVL 考虑了复合类型变量的层次关系,且过程间分析时实现了域敏感的过程间分析,因此 RSTVL 方法与 STVL 方法相比能够检测出更多的对复合类型变量的成员引用的缺陷.图 2 是 RSTVL 方法检测出的一个空指针引用,STVL 方法漏报了该缺陷.被调函数 *usGetNextByte* 可能在 530 行为 *pInfoCurrent→pBlockCurrent* 赋值为空,当函数 *usGetNextChar* 在 595 行调用 *usGetNextByte* 后,*pReadinfo→pBlockCurrent* 受函数调用的副作用影响可能为空,

这将导致在602行对 *pReadinfo→pBlockCurrent* 的引用可能是空指针引用.

Table 4 Detecting results of null pointer dereference defects

表4 空指针引用检测结果

Benchmark	STVL		RSTVL		BUG increment (%)
	IP	BUG	IP	BUG	
antiword-0.37	42	17	67	24	41
uucp-1.07	101	58	144	68	17
sphinxbase-0.3	38	19	51	25	32
optipng-0.6	37	23	48	27	17
barcode-0.98	11	4	10	4	0
Total	229	121	320	148	22

```
文件: antiword\blocklist.c
486行的被调函数:usGetNextByte(*,readinfo_type *pInfoCurrent,*,*,*,*)
530:pInfoCurrent→pBlockCurrent=NULL;

557 行的主调函数:usGetNextChar:
595:usLSB=usGetNextByte(pFile,pReadinfo,pAnchor,pulFileOffset,pulCharPos,pusPropMod);
602:if (pReadinfo→pBlockCurrent→tInfo.bUsesUnicode);   //空指针引用
```

Fig.2 A null pointer dereference defect detected by our strategy

图2 本文方法发现的空指针引用缺陷

在表4的检测结果中,RSTVL方法比STVL方法所报IP稍有增多,主要有3点原因:(1) 对于没有函数摘要的函数,RSTVL采取了保守的策略,默认其返回值为返回类型对应的最大格值;(2) 如果被调用函数没有函数摘要,对于所传递的指针类型实参,如果不是安全指针则判断为空指针引用;(3) 对于过程间引用的指针,因为没有考虑路径条件,导致对一些复合类型变量的指针类型成员的引用被误报为空指针引用,而STVL方法不对此类指针的引用进行检测.

图3是RSTVL方法产生的一个空指针引用误报.对于 *uucp\uuconf\strip.c* 第44行被引用的指针 *qglobal→qprocess*,通过分析实际是对扩充变量(**pglobal*).*qprocess* 的引用,因此,将(**pglobal*).*qprocess* 不能为空作为函数 *uuconf_strip* 的前置约束.而对于 *uucp\uuchk.c* 的函数调用 $main \xrightarrow{243} uuconf\_strip$,根据被调函数 *uuconf_strip* 的前置约束,需要判断 *puuconf→qprocess* 是否为空.而 *puuconf* 与 *puuconf→qprocess* 是通过函数调用 $main \xrightarrow{145} uuconf\_init$ 的副作用被赋值的,如果 *puuconf→qprocess* 为空指针,将导致 *uucp\uuchk.c* 第146行的条件 *iret*!=*UUCONF_SUCCESS* 成立,程序将终止,这样将保证在 *uucp\uuchk.c* 第146行传递给 *uuconf_strip* 的 *puuconf→qprocess* 不为空,不会产生空指针.而RSTVL方法未考虑 *puuconf→qprocess* 与 *uuconf_init* 返回值的关系,导致误判了传递给 *uuconf_strip* 的 *puuconf→qprocess* 可能为空,产生了误报.STVL方法虽然未产生该误报,原因是其对 *qglobal→qprocess* 既未进行检测也未建立为前置约束.

```
文件: uucp\uuchk.c
145:   iret=uuconf_init (&puuconf, (const char *) NULL, zconfig);
146:   if (iret!=UUCONF_SUCCESS)
147:       ukuuconf_error(puuconf,iret);
243:   iret=uuconf_strip(puuconf,&iint);   //误报puuconf→qprocess为空指针引用

文件: uucp\uuconf\strip.c
36:   int uuconf_strip(pointer pglobal,int *pistrip)
41:   struct sglobal *qglobal=(struct sglobal *) pglobal;
44:   if (qglobal→qprocess→fstrip_login)
```

Fig.3 A null pointer dereference false positive caused by our strategy

图3 本文方法产生的空指针引用误报

5 相关工作

基于不同的静态分析方法以及针对不同应用,已提出了若干种描述运行时内存状态的模型.最简单的是名-值对[19],未考虑内存单元间的任何关联.较直观的是数组模型,但难以表示内存单元间的层次关系.别名对集合、指向关系集合、二元决策图[20]能够表示别名关系,适用于指针分析.形态图与 TVAL 适用于形态分析,主要用于判断程序运行时内存中数据结构的"形态"信息.区域的概念应用于内存的管理[21,22],在静态分析领域,基于区域的模型是用一个区域表示为一个内存对象分配的一块连续内存块[6,23].文献[23]提出了基于区域的形态分析,将全局的堆空间分解为独立的地址集合,并应用于内存泄露的检测,但不支持数据流分析.文献[6]提出了基于区域的三元模型,未用抽象域描述内存中的值,未考虑取值的逻辑关联关系,且只适用于单路径的分析.在静态分析中,用抽象域的域元素近似地表示变量取值,抽象域分为关系型与非关系型、八边形抽象域[24]、多面体抽象域[25]等,本文采用了非关系型的区间抽象域,但通过符号表达式可表示变量间的逻辑关联关系.

为了提高静态分析的精度,通常采用敏感的分析方法,主要包括流敏感、域敏感、上下文敏感、路径敏感.流敏感的分析在每个程序点计算并保存程序状态.其中,Worklist 算法作为最典型的迭代算法,是最典型的一种流敏感分析算法.为了提高分析的效率,有些研究针对指针分析提出了稀疏的流敏感分析[26,27].对于 C 程序中存在的复合类型变量,域敏感能够保证分析的精度,域敏感分析可分为基于数值偏移[10]与基于符号偏移[4],基于数值偏移的方法因为能够描述连续内存,能够更准确地描述强制类型转换、指针算术,但分析的效率也会下降.上下文敏感的过程间分析主要采用函数摘要[16,17]的方法,基于函数摘要的方法可以避免在每个调用点处重复分析该函数的行为,具有较高的分析效率.

大部分的函数摘要采用基于⟨输入,输出⟩映射表[28],也有一些符号化函数摘要[16,17],符号化的函数摘要在捕获信息时将具体的程序状态表示为符号化抽象取值,具有较高的分析效率.

6 结束语

本文可被看作以前工作[7,13,17]的进一步扩展,更深入、全面地研究了 C 程序的静态分析.一方面,在对内存对象存储的描述上,本文提出了 RSTVL,通过抽象区域模拟实际分配的内存区域,能够更全面地表示运行时内存对象的存储信息,并可以描述内存对象间的指向关系、层次关系和取值逻辑关联信息;另一方面,在基于 RSTVL 的流敏感、域敏感和上下文敏感的分析时,因为能将对可寻址表达式的值修改反映到对抽象区域的状态修改,在分析的各个环节采取了保守的分析策略,这能保证分析结果是实际可能运行情况的上近似,进而保证了分析的可靠性.应用本文分析方法进行代码级别的缺陷检测,将会大大降低检测的漏报率.

因为本文方法是路径不敏感的,在提高分析效率的同时也降低了分析的精度,也导致了对空指针引用检测的可疑指针引用比例偏高.本文的下一步工作是考虑路径的约束条件,特别是生成函数摘要时考虑路径的约束信息,并将静态分析的结果应用于内存泄露、缓冲区溢出、非法计算等更多源代码级别的缺陷的检测.

References:

[1] Dong YK, Xing Y, Jin DH, Gong YZ. An approach to fully recognizing addressable expression. In: Proc. of the 13th Int'l Conf. on Quality Software. Washington: IEEE Computer Society, 2013. 149−152. [doi: 10.1109/QSIC.2013.64]

[2] Zhang J. Symbolic execution of program paths involving pointers and structure variables. In: Proc. of the 4th Int'l Conf. on Quality Software. Washington: IEEE Computer Society, 2004. 87−92. [doi: 10.1109/QSIC.2004.32]

[3] Mooly S, Thomas R, Reinhard W. Solving shape-analysis problems in languages with destructive updating. ACM Trans. on Programming Languages and Systems, 1998,20(1):1−50. [doi: 10.1145/271510.271517]

[4] Lev-Ami T, Sagiv M. TVLA: A system for implementing static analyses. In: Proc. of the 7th Int'l Static Analysis Symp. LNCS 1824, Berlin: Springer-Verlag, 2000. 280−301. [doi: 10.1007/978-3-540-45099-3_15]

[5] Robert P. Wilson, Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In: Proc. of the ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1995. 1−12. [doi: 10.1145/207110.207111]

[6] Xu ZX, Kremenek T, Zhang J. A memory model for static analysis of C programs. In: Proc. of the Leveraging Applications of Formal Methods, Verification, and Validation. LNCS 6415, Berlin: Springer-Verlag, 2010. 535−548. [doi: 10.1007/978-3-642-16558-0_44]

[7] Zhao YS, Wang YW, Gong YZ, Chen HH, Xiao Q, Yang ZH. STVL: Improve the precision of static defect detection with symbolic three-valued logic. In: Proc. of the 8th Asia-Pacific Software Engineering Conf. Washington: IEEE Computer Society, 2011. 179−186. [doi: 10.1109/APSEC.2011.23]

[8] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. A static analyzer for large safety-critical software. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. San Diego: ACM Press, 2003. 196−207. [doi: 10.1145/780822.781153]

[9] Choi JD, Burke M, Carini P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1993. 232−245. [doi: 10.1145/158511.158639]

[10] Miné A. Field-Sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the 2006 ACM SIGPLAN/SIGBED Conf. on Language, Compilers, and Tool Support for Embedded Systems. New York: ACM Press, 2006. 54−63. [doi: 10.1145/1134650.1134659]

[11] Hampapuram H, Yang Y, Das M. Symbolic path simulation in path-sensitive dataflow analysis. In: Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM Press, 2005. 52−58. [doi: 10.1145/1108792.1108808]

[12] Kam JB, Ullman JD. Global data flow analysis and iterative algorithms. Journal of the ACM, 1976,23(1):158−171. [doi: 10.1145/321921.321938]

[13] Wang YW, Gong YZ, Xiao Q, Yang ZH. A method of variable range analysis based on abstract interpretation and its applications. Acta Electronica Sinica, 2011,39(2):296−303 (in Chinese with English abstract).

[14] Cousot P, Cousot R. Static determination of dynamic properties of programs. In: Proc. of the 2nd Int'l Symp. on Programming. 1976. 106−130. http://www.di.ens.fr/~cousot/COUSOTpapers/ISOP76.shtml

[15] Li MJ, Li ZJ, Chen HW. Program verification techniques based on the abstract interpretation theory. Ruan Jian Xue Bao/Journal of Software, 2008,19(1):17−26 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/17.htm [doi: 10.3724/SP.J.1001.2008.00017]

[16] Gulwani S, Tiwari A. Computing procedure summaries for interprocedural analysis. In: Proc. of the 16th European Conf. on Programming. LNCS 4421, Berlin: Springer-Verlag, 2007. 253−267. [doi: 10.1007/978-3-540-71316-6_18]

[17] Zhao YS, Gong YZ, Liu L, Xiao Q, Yang ZH. Context-Sensitive interprocedural defect dection based on a unified symbolic procedure summary model. In: Proc. of the 11th Int'l Conf. on Quality Software. Barcelona: IEEE Computer Society, 2011. 51−60. [doi: 10.1109/QSIC.2011.15]

[18] Huang B, Zang BY, Wei JY, Zhu CQ. Context-Sensitive interprocedural pointer analysis. Chinese Journal of Computers, 2000, 23(5):477−485 (in Chinese with English abstract). [doi: 10.3321/j.issn:0254-4164.2000.05.005]

[19] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394. [doi: 10.1145/360248.360252]

[20] Berndl M, Lhoták O, Qian F, Hendren L, Umanee N. Points-To analysis using BDD. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2003. 103−114. [doi: 10.1145/780822.781144]

[21] Berger ED, Zorn BG, McKinley KS. Reconsidering custom memory allocation. In: Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 2002. [doi: 10.1145/583854.582421]

[22] Lattner C, Adve V. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2005. [doi: 10.1145/1064978.1065027]

[23] Hackett B, Rugina R. Region-Based shape analysis with tracked locations. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 2005. 310−323. [doi: 10.1145/1047659. 1040331]

[24] Mine A. The octagon abstract domain. Higher-Order and Symbolic Computation, 2001,19(1):31−100. [doi: 10.1007/s10990-006-8609-1]

[25] Clarisó R, Cortadella J. The octahedron abstract domain. In: Proc. of the ACM SIGPLAN Conf. on Int'l Static Analysis Symp. LNCS 1824, London: Springer-Verlag, 2004. 312−327. [doi: 10.1007/978- 3-540-27864-1_23]

[26] Hardekopf B, Lin C. Semi-Sparse flow-sensitive pointer analysis. In: Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 2009. 226−238. [doi: 10.1145/1594834.1480911]

[27] Oh H, Heo KH, Lee WC, Lee WS, Yi KK. Design and implementation of sparse global analyses for C-like languages. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2012. 229−238. [doi: 10.1145/2345156.2254092]

[28] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. In: Proc. of the Program Flow Analysis: Theory and Applications. Upper Saddle River: Prentice-Hall, 1981. 189−233.

附中文参考文献:

[13] 王雅文,宫云战,肖庆,杨朝红.基于抽象解释的变量值范围分析及应用.电子学报,2011,39(2):296−303.

[15] 李梦君,李舟军,陈火旺.基于抽象解释理论的程序验证技术.软件学报,2008,19(1):17−26. http://www.jos.org.cn/1000-9825/19/17.htm [doi: 10.3724/SP.J.1001.2008.00017]

[18] 黄波,臧斌宇,韦俊银,朱传琪.上下文敏感的过程间指针分析.计算机学报,2000,23(5):477−485. [doi: 10.3321/j.issn:0254-4164.2000.05.005]

董玉坤(1981－),男,山东济宁人,博士生,讲师,CCF 学生会员,主要研究领域为软件测试,程序静态分析.
E-mail: dongyk@upc.edu.cn

宫云战(1962－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,软件测试.
E-mail: gongyz@bupt.edu.cn

金大海(1974－),男,博士,副教授,CCF 会员,主要研究领域为软件工程,软件测试.
E-mail: jindh@bupt.edu.cn

邢颖(1978－),女,讲师,CCF 学生会员,主要研究领域为软件测试,软件工程.
E-mail: lovelyjamie@yeah.net