软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(5):1276−1293 [doi: 10.13328/j.cnki.jos.005949]

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

面向顺序存储结构的数据流分析*

王淑栋, 尹文静, 董玉坤, 张 莉, 刘 浩

(中国石油大学(华东) 计算机科学与技术学院,山东 青岛 266580)

通讯作者: 董玉坤, E-mail: dongyk@upc.edu.cn

摘 要: C 程序中数组、malloc 动态分配后的连续内存等顺序存储结构被大量使用,但大多数传统的数据流分析方法未能充分描述其结构及其上的操作,特别是在利用指针访问顺序存储结构时,传统的分析方法只关注了指针的指向关系,而未讨论指针可能发生偏移的数值信息,且未考虑发生偏移时可能存在越界的不安全问题,导致了对顺序存储结构分析不精确.针对以上不足,首先对顺序存储结构进行抽象建模,并对顺序存储结构与指针结合使用时的指向关系与偏移量进行有效表示,建立了用于顺序存储结构的抽象内存模型 SeqMM;其次,归纳总结 C 程序中顺序存储结构涉及的指针相关迁移操作、谓词操作及遍历顺序存储结构的循环操作,提出了安全范围判别保证操作安全性;之后,针对函数调用时形参指针引用顺序存储结构与实参的映射过程进行过程间推导规则设计;最后,基于上述分析,提出了一种内存泄漏缺陷检测算法,对 5 个开源 C 工程的内存泄漏缺陷进行检测.实验结果表明,所提出的 SeqMM 能够有效地刻画 C 程序中的顺序存储结构及其涉及的各种操作,其数据流分析结果能够用于内存泄漏的检测工作,同时在效率和精度之间取得合理的权衡.

关键词: 顺序存储结构;数据流分析;抽象内存模型;过程间分析;内存泄漏

中图法分类号: TP311

中文引用格式: 王淑栋,尹文静,董玉坤,张莉,刘浩.面向顺序存储结构的数据流分析.软件学报,2020,31(5):1276−1293. http://www.jos.org.cn/1000-9825/5949.htm

英文引用格式: Wang SD, Yin WJ, Dong YK, Zhang L, Liu H. Data flow analysis for sequential storage structures. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1276−1293 (in Chinese). http://www.jos.org.cn/1000-9825/5949.htm

Data Flow Analysis for Sequential Storage Structures

WANG Shu-Dong, YIN Wen-Jing, DONG Yu-Kun, ZHANG Li, LIU Hao

(College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China)

Abstract: Sequential storage structures such as array and continuous memory block allocated dynamically by malloc are widely used in C programs. But traditional data flow analysis fails to adequately describe their structures and operations. When pointers are used to access the sequential storage structures in C programs, existing data flow analysis methods basically pay attention to only points-to information and do not consider the numerical properties offset. In addition, it does not consider the unsafe problem caused by out of bounds when offset occurs, which leads to inaccurate analysis for sequential storage structure. To improve the precision for analyzing sequential storage structures, an abstract memory model SeqMM is proposed to describe sequential storage structures, which can effectively describe points-to relationships and offset. Then there are three operations are summarized, such as the pointer-related transfer operation, predicate operation, and loop operation traversing sequential storage structures, and it is also considered that whether the index is out of bounds to ensure the security of operation execution when analyzing these operations. After that, mapping rules are introduced

* 基金项目: 中央高校基本科研业务费专项资金(19CX02028A); 国家自然科学基金(61873281)

Foundation item: Fundamental Research Funds for the Central Universities (19CX02028A); National Natural Science Foundation of China (61873281)

本文由“系统软件构造与验证技术”专题特约编辑赵永望副教授、刘杨教授、王戟教授推荐.

收稿时间: 2019-08-31; 修改时间: 2019-10-24; 采用时间: 2019-12-24; jos 在线出版时间: 2020-04-07

for parameters referencing sequential storage structure to corresponding arguments. Finally, a memory leak detection algorithm is proposed to detect memory leak in 5 open-source projects. The experimental results indicate that SeqMM can effectively describe sequential storage structure and various operations in C programs, and the results of data flow analysis can be used to detect memory leaks when a reasonable balance between accuracy and efficiency occurs.

Key words: sequential storage structure; data flow analysis; abstract memory model; inter-prcedural analysis; memory leak

数据流分析(data flow analysis)一直是程序分析、缺陷检测、测试用例生成等方面关注的重要问题,它是静态分析中的关键技术,通过分析程序状态信息在控制流图中的传播来计算每个静态程序点(语句)在运行时可能出现的状态.通过数据流分析,测试人员能够在未实际执行程序的情况下发现程序运行时的行为与相关信息.

数组、*malloc*(·)动态分配后的连续内存等顺序存储结构在 C 程序中被大量使用,其结构的复杂性使得无法利用传统的数据流分析方法对其进行准确分析.传统的数据流分析方法基于分析和效率折中的考虑,较少对顺序存储结构进行建模;其次,在程序利用指针访问顺序存储结构时,大多数现有的数据流分析方法基本只关注了指针的指向信息,并未考虑指针在连续内存中可能发生偏移的数值性质,更未考虑发生偏移时可能存在越界的不安全问题.因此,对 C 语言中的顺序存储结构进行有效刻画与分析,是数据流分析领域的难点.

为了说明本文对顺序存储结构的分析,首先定义本文研究的顺序存储结构,如图 1 所示.在图 1 中,m_1 为物理内存,其中,矩形框表示部分内存块,斜线标注部分为数组或 *malloc*(·)动态分配成功的连续内存;m_2 为对 m_1 处斜线标注内存内部结构的细致描述,假设此连续内存已存储了 int 型数据,其中每一个 int 型数据在内存中占据 4 个字节,图中每一最小矩形块代表一个字节;m_3 为本文定义的顺序存储结构.本文研究的顺序存储结构不考虑具体内存存储数据的类型,即不考虑存储数据在内存中所占字节数.同时,将存储一个数据的内存大小记为一个单元(斜线标注部分为其中一个单元),并按照每个单元与首单元的相对位置,从 0 开始依次对内存单元进行编号.

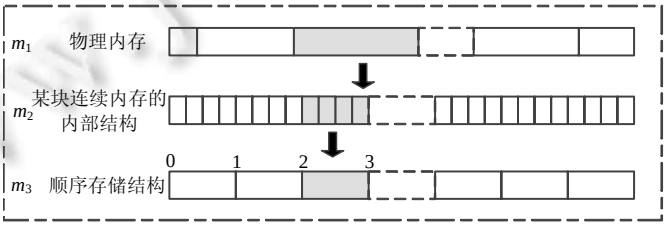


Fig.1 Sequential storage structure studied in this paper

图 1 本文研究的顺序存储结构

基于定义的顺序存储结构,本文提出了一种用于 C 语言顺序存储结构的数据流分析方法.首先,本文以基于区域的符号化三值逻辑(RSTVL)[1]模型为基础,从顺序存储结构的整体与单个元素两层面实现对顺序存储结构的完整建模,提出了描述顺序存储结构的抽象内存模型 SeqMM;之后,本文将 C 程序中顺序存储结构与指针结合使用时的操作进行归纳,利用指针指向关系分析和数值性质分析相结合的方法对顺序存储结构的访问过程进行细致的分析与总结;其次,本文对形参指针引用顺序存储结构元素与实参的映射进行了推导规则设计,实现了过程间分析;最后,基于上述分析,本文实现了 C 程序的内存泄漏缺陷检测算法,并在 5 个开源 C 工程及植入的内存泄漏缺陷中验证了该算法的有效性.同时,实验结果表明:SeqMM 模型可有效表示 C 程序中的顺序存储结构,准确刻画了连续内存上的指针访问顺序存储结构时的相关操作,其分析结果能够用于内存泄漏的检测工作,同时,在效率和精度之间取得合理的权衡.

本文第 1 节对现有抽象内存模型、数据流分析方法进行总结.第 2 节定义一种简化的 C 语言,确定本文研究范围,并提出描述顺序存储结构的抽象内存模型 SeqMM.同时,在第 3 节归纳 C 程序中指针访问顺序存储结构相关操作并依次进行分析.第 4 节对本文提出的方法在内存泄漏的缺陷检测应用中进行细致的讨论.同时,在第 5 节展开一系列的统计与实验对比讨论,在精度和效率上验证本方法的有效性.第 6 节对本文工作进行总结,同时展望下一步的研究工作.

1 相关工作

1.1 现有的抽象内存模型

对程序进行数据流分析,首先应该确定程序中的变量在不同程序点处的存储状态,即构建描述变量的存储模型.目前,基于不同的精度、效率及应用的考虑,研究者们已提出了多种刻画变量存储信息的抽象内存模型.

最初被提出的抽象内存模型是名-值对[2],其简单、直观,但无法直接对指针、数组、结构体等复杂结构进行有效的描述.为此,Xu 等人[3]引入了区域层次结构,它在内存对象的识别、跟踪及内存对象间的关系推理方面具有重要作用,未考虑多路径的分析.Zhang 等人[4]提出了一种直观的数组模型,可处理指针与结构体,未能表示内存单元之间的层次关系.Hackett 等人[5]提出了一种用局部堆位置推理代替全局堆抽象推理的内存抽象模型,将形状抽象分解为一组独立的地址集合,每个地址描述一个堆位置,但该模型不支持数据流分析.董龙明等人[6]提出了一种堆操作程序中的内存安全性域敏感的 *k*-limit 抽象存储模型,能够描述指针之间的别名与局部可达关系.STVL[7]可统一抽象运行时的数据流信息,能表示变量之间的逻辑关系,却不能表示层次关系,而 RSTVL 恰恰解决了这一问题.

以上大部分对变量的存储状态研究基本上都只关注了指针的指向信息,却未处理指针可能发生偏移后的数值性质.

文献[8]在考虑指针指向关系与偏移量的基础上提出了一种指针内存模型,并基于该模型设计了一个可以描述指针指向关系与偏移信息的抽象域,但在刻画指针偏移量时,使用大范围区间来表示偏移数值范围,导致指针偏移量分析结果误差较大.

1.2 数据流分析方法

为了提高数据流分析的精度,研究人员通常采用敏感的分析方法,主要包括流敏感、上下文敏感、路径敏感、域敏感.

近些年,研究人员在流敏感、上下文敏感两方面针对 Steensgaard[9]或 Andersen[10]算法的改进与扩展进行了大量的研究工作,并主要集中于指针分析[11−15].文献[16]引入了一种需求驱动的强更新指针分析方法 SUPA,它可以计算 C 程序数据流的指向信息,并能在一定分析预算的约束下对预先计算不精确的数值流进行精化,从而实现上下文敏感.

路径敏感分析增加了对代码中路径条件的考虑.一些缺陷检测工具如 Prefix[17]在某些选定的路径中寻找 bug,或使用量化的递归公式对许多程序属性敏感的路径和上下文条件进行精确地表示[18];Gutzmann 等人[19]讨论了在数据流中插入特定于路径的过滤器操作来保留控制流信息,但没有考虑捕获路径相关性,也没有排除不可行的路径.Sui 等人[20]提出一种可伸缩路径敏感框架 SPAS,可获得程序中所有指向集合,但同时增大了空间开销.文献[21]通过识别最小不可行路径段的簇来区分沿着可行和不可行控制流路径流动的数据,引入了对 MFP 解的部分路径敏感性,实现了变量的定义分析和值范围分析.

域敏感分析通过考虑结构体等复合数据类型与成员之间的关系,保证对复合类型变量分析的准确性.Litvak 等人[22]专注处理大型复合结构变量的程序,并为这类程序计算域敏感程序依赖关系的问题提供了一个有效和精确的解决方案.

针对现有工作对抽象内存模型以及数据流分析研究的不足,本文提出了 SeqMM 模型,从顺序存储结构的整体与单元素两个层面实现对顺序存储结构的完整建模.本文通过归纳分析访问顺序存储结构的操作与过程间分析实现流敏感、上下文敏感、域敏感的数据流分析,同时将分析结果应用于内存泄漏的检测工作.

2 面向顺序存储结构的抽象内存模型

2.1 一种简化的C语言

指针记录了程序执行过程中内存中的某个地址,同时在指针指向的地址上存储了某种类型的数据[8].通过

对源代码进行分析可得,C 语言中的指针可以被赋值为某个变量的地址、数组变量及某个数组元素的地址等.

定义 1(地址表达式). 基于源代码级指针指向地址的考虑,将指针变量、取地址表达式、数组变量及三者分别与整型表达式进行加减操作后的表达式及 *null* 统称为地址表达式.

下面给出由真实代码改编的一段示例代码,如图 2 所示.代码中的 *ptr*,*buf*,*line*,*more*,&*buf*[*x.data*+2],++*line*,*more*-*x.data*+4,*x.result* 等均属于地址表达式.

```
L1  #define bufsize 8
L2  typedef struct Wl{
L3        int result[8];
L4        float data;
L5  } wl;
L6  unsigned char *upc_a_to_e(wl x){
L7            int i,value=1;
L8            int *fmt=&i,*ptr;
L9            unsigned char *buf,*line,*more;
L10           x.data=3.0;
L11           buf=malloc(bufsize*sizeof(char));
L12           line=&buf[x.data+2];
L13           more=++line;
L14           if (line<=buf+(bufsize−2)){
L15                 line+=x.data;
L16                 buf=more-x.data+4;
L17           }
L18           free(buf);
L19           if (i>0)
L20                 ptr=x.result;
L21           else
L22                 ptr=x.result+5;
L23           i=*ptr;
L24           i=ptr[value+2];
L25           i=ean_make(ptr);
L26           return line;
L27     }
L28     int ean_make(int*spc){
L29           spc[2]=30;
L30           return spc[2]+1;
L31     }
```

Fig.2 The sample program

图 2 程序示例

为了形式化描述本文提出的模型,定义本文所支持的 C 语言文法,如图 3 所示.程序 P 是变量 V 与语句 S 的集合,地址表达式集合为 *AER*,可寻址表达式[1]集合为 *AE*.

programs: $prog \in Prog = P \rightarrow (V^n \times S)$

procedures: $p \in P$

statement: $s \in S$, $s ::= e_0 \leftarrow e_1 | e \leftarrow malloc | free(e) | \textbf{call}\ p(e_1,\ldots,e_n) | s_0;s_1 | \textbf{if}\ (e)\ s_0\ \textbf{else}\ s_1$

expressions: $e \in E$,

$e ::= constant | uop\ aexp | aexp\ uop | aexp_1\ bop\ aexp_2$,

where $uop ::= ++|--$, $bop ::= =>|<|>=|<=|==|!=|+|-$

addressable expression: $aexp \in AE$,

$aexp ::= var | aexp.f | aexp->f | aexp[n] | (aexp) | {}^*aexp | addrexp$

address expression: $addrexp \in AER$,

$addrexp ::= null | pointer | \&aexp | array | addrexp_1\ op_1 | op_1\ addrexp_1 | addrexp_1\ op_2\ iexp$

where $op_1 ::= =>|<|>=|<=|==|!=$, $op_2 ::= +|-$, $op_3 ::= ++|--$

variables: $var \in V$

fields: $f \in F = \{f_1,\ldots,f_m\}$

Fig.3 Grammar of C language

图 3 C 语言文法

2.2 面向顺序存储结构的抽象内存模型

在之前的工作中,我们已提出了 RSTVL 模型[1],用来描述可寻址表达式间的指向关系、层次关系与取值逻辑关系,但未涉及对偏移的处理,导致对顺序存储结构相关的缺陷,如数组越界、内存泄漏、缓冲区溢出等检测不精确.因此,本文改进了 RSTVL 模型,使其可以有效刻画顺序存储结构,并精确描述指针访问顺序存储结构操作中的偏移性质.

在图 2 所示代码中,L8 行的指针 *ptr* 及 L9 行的指针 *buf*,*line*,*more* 均为野指针,不确定其指向;在 L12 行指针 *line* 指向 *buf*[*x.data*+2],其指向基址为 *malloc*(·)动态分配成功后顺序存储结构的首地址即 *buf*[0],偏移量为 *x.data*+2;在 L13 行 *more* 指向基址为 *buf*[0],其指向 *buf*[*x.data*+2]单元,偏移量为 *x.data*+2.指针 *buf* 初始指向 *malloc*(·)动态分配成功后顺序存储结构的首地址,L16 行执行成功后,其指向内存单元 *more*[1],偏移量为 7.

定义 2(面向顺序存储结构的抽象内存模型(abstract memory model oriented to sequential storage structure,简称 SeqMM). SeqMM 被定义为四元组:$SepMM=\langle Var,Region,S_{Exp},Domain\rangle$.其中,*Var* 表示内存对象,*Region* 表示内存对象分配的内存块;对每一个描述区域 *Region* 用唯一的区域编号进行标识,区域编号对应区域 *Region* 的左值,符号表达式 S_{Exp} 表示区域 *Region* 的右值,*Domain* 表示取值区间.内存对象、符号表达式等概念参见文献[1].

对于 C 程序中的一个变量 *p*,*p* 的类型不同,SeqMM 有不同的表示.

(1) 当 *p* 为数组变量时,其模型表示为$\langle Var,Region,Domain\rangle$,此时取值区间 $Domain=\{r_1,r_2,\ldots,r_n\}$为各数组元素区域信息,其中,$r_i=\langle elementIndex,elementRegion\rangle$由某个数组元素索引与其对应区域构成.

(2) 当 *p* 为指针变量时,$SepMM=\langle Var,Region,S_{Exp},Domain\rangle$,$Domain=\{d_1,d_2,\ldots,d_n\}$为 *p* 指向区域与对应偏移的集合,其中,$d_i=\langle PtRegion,Offset\rangle$,*PtRegion* 为指针 *p* 指向的内存对象 *Var* 对应区域,且 *Offset* 是一个整型表达式,表示偏离指向基址的偏移量对应的整型区间(不考虑指针的指向内存对象所占内存字节数大小).

(3) 当 *p* 为结构体变量时,$SepMM=\langle Var,Region,S_{Exp},Domain\rangle$,此时取值区间 $Domain=\{r_1,r_2,\ldots,r_n\}$为结构体各成员的区域信息,其中,$r_i=\langle member,memberRegion\rangle$由结构体某成员及其对应区域构成.

(4) 当 *p* 为基本类型变量时,若 *p* 的取值 *value* 未知,其模型表示为四元组$\langle Var,Region,S_{Exp},Domain\rangle$,此时 $Domain=[-inf,inf]$;若 *p* 的取值已知,其模型则为三元组$\langle Var,Region,S_{Exp}\rangle$,此时将其具体取值记为 *p* 的符号表达式.

对不同类型的内存对象 *Var*,SeqMM 用不同类型的区域对其存储状态进行抽象刻画.*PrimitiveRegion* 刻画基本数据类型的内存对象,*PointerRegion*,*ArrayRegion*,*StructRegion* 分别用来刻画指针、数组或 *malloc*(·)动态分配的内存区域、结构体.每一个描述区域仅有唯一的区域编号,上述区域的编号形式依次为“*bm_i*”,“*pm_i*”,“*am_i*”,“*sm_i*”($i\in\mathbf{N}$,从 0 开始递增).对 *malloc*(·)动态分配的无名内存,用“*mxm_i_n*”(“*x*”表示上述区域类型,取值可为“*b*”,“*p*”,“*a*”,“*s*”)进行区域的抽象描述,其中,“*n*”为该无名内存的字节数.对于空指针,其对应地址的区域编号为“*null*”,野指针为“*wild*”,同时,对于函数参数和全局变量的区域编号分别额外增加首字母“*u*”与“*g*”.

用本文所提出的 SeqMM 模型对图 2 代码中的部分变量进行建模,分析得出 L6 行结构体 *x* 的模型表示为$\langle x,usm_0,\{\cdot\}\rangle$;L7 行变量 *i* 用本模型表示为$\langle i,bm_1,i_01,[-inf,inf]\rangle$,$[-inf,inf]$用来标识函数内数值型变量取值区间未知,同时变量 *value* 可表示为$\langle value,bm_2,1\rangle$(“1”为 *value* 的符号表达式);L8 行指针 *fmt* 指向整型变量 *i*,其模型表示为$\langle fmt,pm_3,fmt_45,\{\langle bm_1,[0,0]\rangle\}\rangle$,其中,指针指向基本类型变量时,其偏移区间固定为[0,0],其次,无法确定指针 *ptr* 的指向,故其模型表示为$\langle ptr,pm_7,ptr_1011,\{\langle wild,[0,0]\rangle\}\rangle$,其中,野指针指向地址的区域编号为“*wild*”,*Offset* 初始区间为[0,0];同样地,L9 行指针 *buf*,*line*,*more* 可分别表示为$\langle buf,pm_8,buf_1213,\{\langle wild,[0,0]\rangle\}\rangle$,$\langle line,pm\_9,line\_1415,\{\langle wild,[0,0]\rangle\}\rangle$,$\langle more,pm\_10,more\_1617,\{\langle wild,[0,0]\rangle\}\rangle$;L10 行在为结构体 *x* 的成员 *data* 赋值后,*x.data* 用模型表示为$\langle x.data,ubm_11,3.0\rangle$(“3.0”为 *x.data* 的符号表达式),同时,变量 *x* 模型更新为$\langle x,usm_0,\{data:3.0\}\rangle$;L11 行执行结束后,变量 *buf* 的模型表示为$\langle buf,pm_8,buf_1819,\{\langle mbm_12_8,[0,0]\rangle\}\rangle$.

3 面向顺序存储结构的数据流分析

C 程序中主要通过指针访问顺序存储结构,本节将针对指针的迁移操作、谓词操作、遍历顺序存储结构循环操作、形参指针引用顺序存储结构时与实参的对应进行分析.

3.1 指针相关迁移操作分析

在 C 程序中,指针主要通过值、地址访问顺序存储结构实现对其他变量的赋值,下面本节将就两种方式分别进行细致的讨论.

假定在程序点 *l* 处,$R^l[\![Var]\!]$表示内存对象 *var* 对应区域 *r*;$P^l[\![p]\!]=\{PtRegion_1,PtRegion_2,\ldots,PtRegion_n\}$为指针 *p* 指向区域的集合;$B^l[\![r]\!]$表示区域 *r* 对应的基址区域,*length* 为该区域单元个数即长度;$O^l[\![r]\!]$表示区域 *r* 偏离基址

区域的偏移区间 $Offset$;$E^l[\![r]\!]$表示区域 r 对应的符号表达式; $D^l_{se}[\![S_{Exp}]\!]$表示符号表达式 S_{Exp} 对应的抽象取值区间 $Domain$.

3.1.1 地址访问迁移操作分析

从源代码级看,C程序中通过地址访问顺序存储结构可统一归结为地址表达式 $addrexp_i$的操作,其在程序中表示多样,本节归纳为如下 7 类.

- $p=q+iexp$; //q 为指针变量
- $p=A+iexp$; //A 为数组变量
- $p=\&A[j]+iexp$;
- p++;
- ++p; //操作与 p++类似,不再单独讨论
- p−−;
- −−p; //操作与 p−−类似,不再单独讨论

本文根据上述常见的地址访问操作对指针的赋值总结如下的迁移操作,见表 1,其中,$newS_{Exp}$ 表示产生的新符号表达式,nR 为区域 r 偏移某个长度后的区域,out of Bounds 表示索引越界警告.

Table 1 Address access migration operation

表 1 地址访问迁移操作

语句	迁移操作
$p=q+iexp$;	$\dfrac{B^l[\![r]\!]=ArrayRegion\wedge 0\leqslant O^l[\![r]\!]+iexp<B^l[\![r]\!].length}{nR\wedge O^l[\![nR]\!]=O^l[\![r]\!]+iexp\wedge B^l[\![nR]\!]=B^l[\![r]\!]\wedge newS_{Exp}}(\exists r\in P^l[\![q]\!])$ $\dfrac{B^l[\![r]\!]=ArrayRegion\wedge(O^l[\![r]\!]+iexp<0\vee O^l[\![r]\!]+iexp\geqslant B^l[\![r]\!].length)}{\text{out of Bounds}}(\forall r\in P^l[\![q]\!])$
$p=A+iexp$;	$\dfrac{R^l[\![A]\!]}{nR\wedge O^l[\![nR]\!]=iexp\wedge B^l[\![nR]\!]=R^l[\![A]\!]\wedge newS_{Exp}}$ $\dfrac{expr<0\vee expr\geqslant R^l[\![A]\!].length}{\text{out of Bounds}}$
$p=\&A[j]+iexp$;	$\dfrac{R^l[\![A[j]]\!]\wedge O^l[\![R^l[\![A[j]]\!]]\wedge B^l[\![R^l[\![A[j]]\!]]}{nR\wedge O^l[\![nR]\!]=j+iexp\wedge B^l[\![nR]\!]=B^l[\![R^l[\![A[j]]\!]]\wedge newS_{Exp}}$ $\dfrac{j+iexp<0\vee j+iexp\geqslant B^l[\![R^l[\![A[j]]\!]]\!].length}{\text{out of Bounds}}$
p++;	$\dfrac{B^l[\![r]\!]=ArrayRegion\wedge 0\leqslant O^l[\![r]\!]+1<B^l[\![r]\!].length}{nR\wedge O^l[\![nR]\!]=O^l[\![r]\!]+1\wedge B^l[\![nR]\!]=B^l[\![r]\!]\wedge newS_{Exp}}(\exists r\in P^l[\![p]\!])$ $\dfrac{B^l[\![r]\!]=ArrayRegion\wedge O^l[\![r]\!]+1\geqslant B^l[\![r]\!].length}{\text{out of Bounds}}(\forall r\in P^l[\![p]\!])$
p−−;	$\dfrac{B^l[\![r]\!]=ArrayRegion\wedge 0\leqslant O^l[\![r]\!]-1<B^l[\![r]\!].length}{nR\wedge O^l[\![nR]\!]=O^l[\![r]\!]-1\wedge B^l[\![nR]\!]=B^l[\![r]\!]\wedge newS_{Exp}}(\exists r\in P^l[\![p]\!])$ $\dfrac{B^l[\![r]\!]=ArrayRegion\wedge O^l[\![r]\!]-1<0}{\text{out of Bounds}}(\forall r\in P^l[\![p]\!])$

以 $p=q+iexp$ 为例,对于右端 q 的每一个指向区域 r,若为数组元素区域,则执行下面的操作.

(1) 获取区域 r 的偏移 $O^l[\![r]\!]$,当 $O^l[\![r]\!]+iexp$ 在 0 与基址区域长度 $length$ 之间,才能执行迁移操作;否则,提示 out of Bounds 警告.

(2) 在执行迁移操作时,生成区域 r 偏移 $iexp$ 后的新区域 nR,该区域偏移区间为 $O^l[\![r]\!]+iexp$,同时为 p 生成新的符号表达式与 nR 对应.

由表 1 可以看出,本文在更新左端指针指向区域的同时,对偏移区间、基址区域、符号表达式执行相应的更新操作.

利用本文归纳的迁移操作,对图 2 中的代码进行分析验证.代码 L12 行执行完成后,指针 *line* 指向发生变化,指向元素 *buf*[5],其模型表示为⟨*line*,*pm*_9,*line*_2021,{⟨*bm*_15,[5,5]⟩}⟩,其中,"*bm*_15"为 *buf*[5]的区域编号.在代码 L13 行,指针 *line* 先执行自加操作,其模型表示为⟨*line*,*pm*_9,*line*_2425,{⟨*bm*_16,[6,6]⟩}⟩,指向区域及偏移区间均发生变化.同样地,在执行对指针 *more* 的赋值操作后,*more* 的模型表示为⟨*more*,*pm*_10,*more*_2829,{⟨*bm*_16,[6,6]⟩}⟩.因此,L13 行部分指针的指向关系如图 4 所示.若满足 if 条件,则在 L15 行执行时,*line* 应指向 *buf*[9],超过连续内存索引上界,提示 out of Bounds 警告.

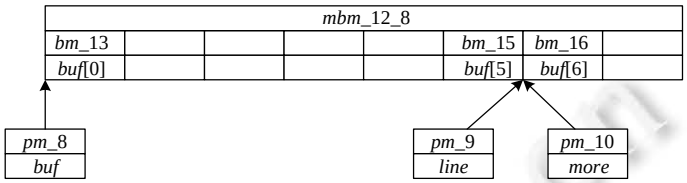


Fig.4 Points-to relationship of partial pointers at L13

图 4 L13 行部分指针指向关系

3.1.2 值访问迁移操作分析

利用指针可方便、灵活、随机访问顺序存储结构中元素的值,设 p 为指针,则归纳其值访问操作如下.

- $a=^{*}p$;
- $a=p[iexp]$. //*iexp* 为整型表达式

根据两种常见值访问操作,可总结如下对应的迁移操作,见表 2.

- 对于 $a=^{*}p$,本文根据指针 p 的每一个指向区域对应符号表达式获得相应的取值区间,并对所有取值区间执行区间并运算,再赋值给左端变量 a 的取值区间 *Domain*,同时生成新的符号表达式与之对应.
- 对于 $a=p[iexp]$,指针 p 应指向数组或 *malloc*(·)动态分配成功的连续内存,且存在指向区域对应的偏移在 0 与基址区域长度 *length* 之间,才可执行迁移操作;否则提示 out of Bounds 警告.$a=p[iexp]$执行迁移操作的过程与 $a=^{*}p$ 类似.

Table 2 Value access migration operation

表 2 值访问迁移操作

值访问	迁移操作
$a=^{*}p$;	$\dfrac{P^l[\![p]\!]}{E^l[\![R^l[\![a]\!]]\!]=newS_{Exp}\wedge D^l_{se}[\![newS_{Exp}]\!]=\cup D^l_{se}[\![E^l[\![r]\!]]\!]}(\forall r\in P^l[\![p]\!])$
$a=p[iexp]$;	$\dfrac{B^l[\![r]\!]=ArrayRegion\wedge 0\leqslant O^l[\![r]\!]+iexp<B^l[\![r]\!].length}{E^l[\![R^l[\![a]\!]]\!]=newS_{Exp}\wedge D^l_{se}[\![newS_{Exp}]\!]=\cup D^l_{se}[\![E^l[\![nR]\!]]\!]}(\exists r\in P^l[\![p]\!])$ $\dfrac{B^l[\![r]\!]=ArrayRegion\wedge(O^l[\![r]\!]+iexp<0\vee O^l[\![r]\!]+iexp\geqslant B^l[\![r]\!].length)}{\text{out of Bounds}}(\exists r\in P^l[\![p]\!])$

在图 2 代码中,在 L23 行语句执行前,指针 *ptr* 指向两个内存对象 *x.result*[0]与 *x.result*[5],故其模型表示为⟨*ptr*,*pm*_7,*ptr*_4445,{⟨*ubm*_19,[0,0]⟩,⟨*ubm*_20,[5,5]⟩}⟩,同时,*x.result* 的模型表示更新为⟨*x.result*,*uam*_18,{⟨0,*ubm*_19⟩,⟨5,*ubm*_20⟩}⟩;当执行 L22 行时,通过指向区域获得对应区域的符号表达式 *x.result*[0]_3637,*x.result*[5]_4041,并依据符号表达式获取当前 *x.result*[0]与 *x.result*[5]的取值均为[−*inf*,*inf*],同时取并集为[−*inf*,*inf*],其次生成新的符号表达式并与[−*inf*,*inf*]之间建立映射关系.在L24行执行时,因区域{"*ubm*_20"}对应偏移为[5,5],[5,5]+*value*+2 后将超出 *x.result* 的索引上界,提示 out of Bounds 警告,因此,该行只根据区域{"*ubm*_19"}进行计算,与 L23 行过程类似.

因此,由表 1、表 2 及示例可得,指针访问顺序存储结构与索引安全性密切相关.本节通过对索引的预先计算检查,对可能超过索引下界或上界的操作进行越界警告,减少之后不必要的分析工作,并有效提高分析精度与效率.

3.2 指针相关谓词操作分析

当分支语句中条件形如⟦$addrexp_1$ $\Re$ $addrexp_2$⟧时,涉及指针对顺序存储结构的访问操作.其中,$addrexp_i$为地址表达式,$\Re\in\{>,<,>=,<=,!=,==\}$.

由地址表达式在 C 代码中的不同形式,可总结为如下 4 类(以$\Re$为“>=”时为例).

- $p+iexp_1>=q+iexp_2$;
- $p+iexp_1>=A+iexp_2$;
- $p+iexp_1>=\&A[j]+iexp_2$;
- $p>=\&a$.

其中,p,q 均为指针,$iexp_i$为整型表达式,a 为基本类型变量.

本文在实现地址表达式的比较时,作如下规定:

(1) 若 $addrexp_1$与 $addrexp_2$存在共同基地址且两者偏移区间满足$\Re$,则谓词一定为真;

(2) 若 $addrexp_1$与 $addrexp_2$存在共同基地址且两者偏移区间不满足$\Re$,则谓词一定为假;

(3) 否则,本文方法无法对地址表达式进行比较.

在归纳地址表达式比较的谓词操作时,本文假定 *flag* 标记条件的布尔值,True 表示条件为真,False 表示条件为假,True or False 表示无法判断,见表 3.对于其他关系符号的地址表达式比较,可以定义类似形式的谓词操作.

Table 3 Predicate operation

表 3 谓词操作

条件	谓词操作
$p+iexp_1>=q+iexp_2$	$\frac{B^l[\![r_1]\!]=B^l[\![r_2]\!]\wedge O^l[\![r_1]\!]+iexp_1\geqslant O^l[\![r_2]\!]+iexp_2}{flag\mapsto \text{True}}(\exists r_1\in P^l[\![p]\!],\exists r_2\in P^l[\![q]\!])$ $\frac{B^l[\![r_1]\!]=B^l[\![r_2]\!]\wedge O^l[\![r_1]\!]+iexp_1< O^l[\![r_2]\!]+iexp_2}{flag\mapsto \text{False}}(\forall r_1\in P^l[\![p]\!],\forall r_2\in P^l[\![q]\!])$ $\frac{B^l[\![r_1]\!]\neq B^l[\![r_2]\!]}{flag\mapsto \text{True or False}}(\forall r_1\in P^l[\![p]\!],\forall r_2\in P^l[\![q]\!])$
$p+iexp_1>=A+iexp_2$	$\frac{B^l[\![r]\!]=R^l[\![A]\!]\wedge O^l[\![r]\!]+iexp_1\geqslant iexp_2}{flag\mapsto \text{True}}(\exists r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]=R^l[\![A]\!]\wedge O^l[\![r]\!]+iexp_1< iexp_2}{flag\mapsto \text{False}}(\forall r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]\neq R^l[\![A]\!]}{flag\mapsto \text{True or False}}(\forall r\in P^l[\![p]\!])$
$p+iexp_1>=\&A[j]+iexp_2$	$\frac{B^l[\![r]\!]=B^l[\![R^l[\![A[j]]\!]]\wedge O^l[\![r]\!]+iexp_1\geqslant[j,j]+iexp_2}{flag\mapsto \text{True}}(\exists r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]=B^l[\![R^l[\![A[j]]\!]]\wedge O^l[\![r]\!]+iexp_1<[j,j]+iexp_2}{flag\mapsto \text{False}}(\forall r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]\neq B^l[\![R^l[\![A[j]]\!]]}{flag\mapsto \text{True or False}}(\forall r\in P^l[\![p]\!])$
$p>=\&a$	$\frac{B^l[\![r]\!]=B^l[\![R^l[\![a]\!]]\wedge O^l[\![r]\!]=[0,0]}{flag\mapsto \text{True}}(\exists r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]=B^l[\![R^l[\![a]\!]]\wedge O^l[\![r]\!]\neq[0,0]}{flag\mapsto \text{False}}(\forall r\in P^l[\![p]\!])$ $\frac{B^l[\![r]\!]\neq B^l[\![R^l[\![a]\!]]}{flag\mapsto \text{True or False}}(\forall r\in P^l[\![p]\!])$

以 $p+iexp_1>=q+iexp_2$ 为例.

(1) 若存在 $r_1\in P^l[\![p]\!],r_2\in P^l[\![q]\!]$,其基址区域相同,且偏移区间满足 $O^l[\![r_1]\!]+iexp_1\geqslant O^l[\![r_2]\!]+iexp_2$,则本文方法判定谓词为 True;

(2) 若对于任意的 $r_1 \in P^l[\![p]\!], r_2 \in P^l[\![q]\!]$,其基址区域相同,且偏移区间无法满足 $O^l[\![r_1]\!]+iexp_1 \geqslant O^l[\![r_2]\!]+iexp_2$,则本文方法判定谓词为 False;

(3) 若对于任意的 $r_1 \in P^l[\![p]\!], r_2 \in P^l[\![q]\!]$,其基址区域不相同,则本文方法无法进行判定.

由上文对图 2 的 L14 行代码分析可得,指针 *line* 的模型表示为⟨*line*,*pm*_9,*line*_2425,{⟨*bm*_16,[6,6]⟩}⟩,其中,"*bm*_16"为内存对象 *buf*[6]对应区域的区域编号,右端指针 *buf* 经过运算,其指向内存对象 *buf*[6],两者存在共同的基址;同时,左端偏移区间[6,6]与右端偏移区间[6,6]满足小于等于关系,故该条件为真,一定会执行 if 语句块内代码.

此外,在分支语句执行完成后,本文分析方法会对各分支数据流事实进行合并操作,以 L19 行分支语句块为例,其控制流图如图 5 所示.在该控制流图中,只给出了与 if 语句块相关的变量存储状态.在执行 if 语句前,*x*,*i*,*ptr* 的模型表示在上述工作中已确定.在 L20 行,*i* 的取值范围变为[1,*inf*],并为新出现的 *x*.*result*[0]分配抽象区域,生成新的符号表达式,其取值为[−*inf*,*inf*].因此,*x* 的模型表示更新为⟨*x*.*usm*_0,{*data*:3.0,*result*:{⟨0,*ubm*_19⟩}}⟩,其中,"*ubm*_19"为 *x*.*result*[0]对应的区域编号.L20 执行完成后,*ptr* 指向 *x*.*result*[0],其模型表示为⟨*ptr*,*pm*_7,*ptr*_1011,⟨{⟨*ubm*_19,[0,0]⟩}⟩⟩.同样地,L22 行语句也执行类似分析.在 if 语句块执行结束后,需对各汇入结束节点的边上的数据流事实进行合并.因此,*x* 的模型表示更新为⟨*x*.*usm*_0,{*data*:3.0,*result*:{⟨0,*ubm*_19⟩,⟨5,*ubm*_20⟩}}⟩,*ptr* 模型表示更新为⟨*ptr*,*pm*_7,*ptr*_4445,⟨{⟨*ubm*_19,[0,0]⟩,⟨*ubm*_20,[5,5]⟩}⟩⟩,其中,*Domain* 已经发生变化,需要生成新的符号表达式与之对应,即"*ptr*_4445".

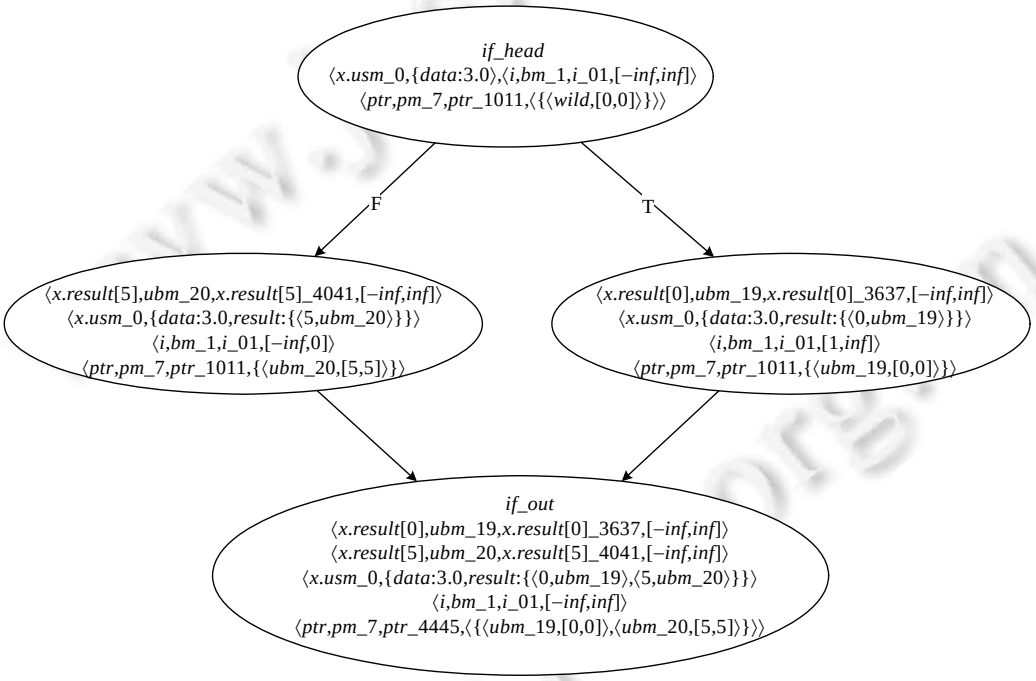


Fig.5 Control flow graph of if statement at L19

图 5 L19 行 if 语句块的控制流图

3.3 循环语句分析

C 程序中循环语句一般为 3 类:for 语句、while 语句、do-wihile 语句.循环语句分析复杂且 do-wihile 语句在实际工程中出现次数较前两者少,因此本文将只对 for 语句与 while 语句中对顺序存储结构遍历的操作进行细致讨论,以 while 语句为例,for 语句分析与 while 语句等价.

本文利用加宽/收窄算子[23]对循环语句的抽象存储迭代求精,直至达到循环内外的抽象存储的不动点.通过

加宽算子使程序不动点语义的计算过程收敛或加速使其收敛,收窄算子对程序不动点语义计算结果进行精化.在对某个顺序存储结构进行遍历时,指针对应偏移区间可唯一确定顺序存储结构的各个元素,因此,本文在对循环语句分析时,只关注当前指针取值 *Domain* 中的偏移区间,其指向区域固定为 *NotNull*.表 4 为本文偏移区间的加宽/收窄算子运算规则,其中,“–1”表示指针指向顺序存储结构 0 号单元的前一个单元.

Table 4 Rules for widening/narrowing operators

表 4 加宽/收窄算子运算规则

加宽算子	收窄算子
$[a_1,b_1]\nabla[a_2,b_2]=[L_1,L_2]$	$[a_1,b_1]\Delta[a_2,b_2]=[L_1,L_2]$
if ($a_2<a_1$)	if ($a_1=-1$)
then $L_1=-1$;	then $L_1=a_2$;
else	Else
$L_1=a_1$;	$L_1=\min(a_1,a_2)$;
if ($b_2>b_1$)	if ($b_1=A.length$)
then $L_2=A.length$;	then $L_2=b_2$;
else	Else
$L_2=b_1$;	$L_2=\min(b_1,b_2)$;

遍历顺序存储结构的循环语句的控制流图结构如图 6 所示.循环语句块的直接前驱节点为 n_1,包含循环条件的循环开始节点为 n_2,循环语句块内循环变量变化的语句对应节点为 n_3,循环结束节点为 n_4.

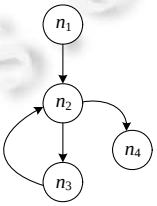


Fig.6 Control flow graph of loop statement traversing sequential storage structure

图 6 遍历顺序存储结构的循环语句的控制流图

本文利用加宽/收窄算子对循环语句进行分析的算法如下所示.

算法 1. 遍历顺序存储结构的循环语句分析算法.

Begin

1: **let** $O^{\bullet n_2}=O^{n_1\bullet}$, *var* as cyclic variable

2: $O^{n_1\bullet}_{temp}=O^{n_1\bullet}-O^{n_1\bullet}[\![R^{n_1\bullet}[\![var]\!]]\!]$;

3: $O^{\bullet n_3}=O^{n_2\bullet\mapsto T}$;

4: $O^{\bullet n_2}_{temp}=O^{n_3\bullet}\cup O^{n_1\bullet}[\![R^{n_1\bullet}[\![var]\!]]\!]$;

5: $O'=O^{\bullet n_2}_{temp}$;

6: **while** true

7: $O^{\bullet n_3}=O^{n_2\bullet\mapsto T}$;

8: $O^{\bullet n_2}_{temp}=O^{n_3\bullet}\cup O^{n_1\bullet}[\![R^{n_1\bullet}[\![var]\!]]\!]$;

9: $O^{\bullet n_2}_{temp}=O'\nabla O^{\bullet n_2}_{temp}$;

10: **if** $O'\neq O^{\bullet n_2}_{temp}$ $O'=O^{\bullet n_2}_{temp}$;

11: **else** break;

12: **let** $O'=O^{\bullet n_2}_{temp}$;

13: **while** true

14: $O^{\bullet n_3}=O^{n_2\bullet\mapsto T}$;

15: $O^{\bullet^{n_2}}_{temp} = O^{n_3\bullet} \cup O^{n_1\bullet}[\![R^{n_1\bullet}[\![var]\!]]\!]$;

16: $O^{\bullet^{n_2}}_{temp} = O' \Delta O^{\bullet^{n_2}}_{temp}$;

17: **if** $O' \neq O^{\bullet^{n_2}}_{temp}$ $O' = O^{\bullet^{n_2}}_{temp}$;

18: **else** break;

19: $O^{n_4\bullet} = O^{\bullet^{n_2}}_{temp} \cup O^{n_1\bullet}_{temp}$

图 7 中,*p* 在 L3 行指向数组 *A*,其指向基址等于 *A* 的基址;同时,偏移区间与[50,50]满足小于关系,故可执行循环体.在分析循环时,本文方法只对偏移区间进行加宽/收窄,最终 *p* 的偏移区间为[50,50].

```
L1  void f(·){
L2      int A[50];
L3      int *p=A;
L4      while (p<A+50)
L5      {
L6          p++;
L7      }
L8  }
```

Fig.7 A sample program that traverses a sequential storage structure

图 7 遍历顺序存储结构的程序示例

3.4 过程间分析

过程间分析是静态分析的重要组成部分.同一函数在不同的调用点被调用,因为上下文环境的不同可能造成实参与全局变量存储状态的不同.近些年,研究者们已对函数调用中指针别名的指向映射进行了大量工作,但对被调函数内形参指针引用顺序存储结构元素与实参的映射研究相对较少.本节针对这一问题开展研究,提出了一种映射推导规则.

本文采用符号化函数摘要[1]的思想,解决偏移相关下实参与形参指针引用时的映射问题.假定 *acPara*, *foPara* 分别表示指向顺序存储结构的实参指针与形参指针,*foPara*[*iexp*]表示通过指针 *foPara* 引用顺序存储结构元素. $P^l_o[\![p]\!] = \{\langle PtRegion_1, offset_1\rangle, \langle PtRegion_2, offset_2\rangle, ..., \langle PtRegion_n, offset_n\rangle\}$ 表示指针 *p* 的指向区域与对应偏移的集合,$V^l[\![r]\!]$表示获得区域 *r* 对应的内存对象 *var*, nR^r_{iexp} 表示区域 *r* 偏移 *iexp* 之后的新区域.本文提出的推导规则如下所示.

$$\frac{foPara: P^l_o[\![foPara]\!] = P^l_o[\![acPara]\!]}{foPara[iexp]: V^l[\![nR^r_{iexp}]\!] \wedge B^l[\![nR^r_{iexp}]\!] = B^l[\![r]\!]} (\forall r \in P^l[\![acPara]\!]).$$

对于形参 *foPara*,其指向区域、偏移及基址区域与实参 *acPara* 相同;对于 *foPara*[*iexp*],获得 *foPara* 的指向区域 $P^l[\![foPara]\!]$,对于每一个区域 *r*,其偏移为 $O^l[\![r]\!]$,基地址区域为 $B^l[\![r]\!]$,*foPara*[*iexp*]的基地址区域与 *foPara* 完全相同,其指向区域在区域 *r* 基础上向左或向右偏移 *iexp* 个单位,对应内存对象为 $V^l[\![nR^r_{iexp}]\!]$.

对于图 2 中的代码,*ean_make* 函数在 L25 行被调用,传递指针 *ptr*.*ptr* 指向区域与对应偏移的集合 $P^l_o[\![ptr]\!] = \{\langle ubm\_19,[0,0]\rangle, \langle ubm\_20,[5,5]\rangle\}$,则形参指针 *spc* 的指向区域与对应偏移集合 $P^l_o[\![spc]\!] = P^l_o[\![ptr]\!]$.对于 *ean_make* 函数内的 *spc*[2],首先获得 *spc* 的指向区域 $P^l[\![spc]\!]$:对于 *spc* 的指向区域{"*ubm*_19"},其偏移为[0,0],则 *spc*[2]指向区域的偏移为[2,2],指向区域为{"*ubm*_23"},指向内存对象为 *x*.*result*[2];对于 *spc* 的指向区域{"*ubm*_20"},其偏移为[5,5],则 *spc*[2]指向区域的偏移为[7,7],指向区域为{"*ubm*_24"},指向内存对象为 *x*.*result*[7].同时,*x*.*result*[2], *x*.*result*[7]的值修改为 31,*x*.*result* 的模型表示更新为

$\langle x.result, uam\_18, \{\langle 0, ubm\_19\rangle, \langle 2, ubm\_23\rangle, \langle 5, ubm\_20\rangle, \langle 7, ubm\_24\rangle\}\rangle$.

4 内存泄漏检测算法

内存泄漏是程序动态内存分配及释放方面常见的软件漏洞.通常情况下,内存泄漏不会产生可直接观察的

错误状态,而是逐渐积累,造成系统内存的浪费,严重情况下可能造成软件崩溃.

近年来,研究人员已从多个方面开展对内存泄漏的研究工作.Šor 在他的博士论文[24]中提出基于 GenCount 概念的增长分析方法来检测 Java 内存泄漏,该方法可识别重复创建新对象的代码位置,但不处理先前创建的对象.在作者另外两篇文献[25,26]中,该方法得到进一步增强.同时,Andrzejak 等人[27]将文献[24,25]的方法转移到 C/C++中,通过对 SPEC CPU2006 中 14 个程序的评估,展示了他们的可行性和准确性.文献[28]提出了一种 C 语言内存泄漏的智能化检测算法,通过使用机器学习算法学习程序与内存泄漏之间的相关性.文献[29]提出了一种基于混合执行测试的静态内存泄漏警报的自动化确认方法,并对内存泄漏警报进行分类,降低人工确认的工作量.

静态分析中,数据流分析是检测多种缺陷的必要工作.本节利用第 2 节提出的 SeqMM 模型,对动态分配内存进行描述,同时利用第 3 节对访问顺序存储结构迁移操作、谓词操作的数据流分析对各程序点处指向动态内存的指针进行刻画,从而实现内存泄漏缺陷检测算法的设计.

本文根据常见的内存泄漏缺陷,将 C 程序中可能发生的内存泄漏缺陷总结为如下两类.

(1) 已成功动态分配的内存未释放.主要是指在程序中对分配的内存没有对应的释放操作或未在全部可能执行路径下释放内存导致无法正确释放内存.

(2) 分配与释放不匹配.主要是指释放的内存空间与分配的内存空间大小不符,如图 2 示例代码 1,在 L16 行执行对指针 *buf* 的赋值操作后,*buf* 指向连续内存的 7 号单元,而在执行 L18 行 *free*(*buf*)操作后,造成内存泄漏,未将内存 0 单元~6 单元成功释放.

针对上述情况,本文以函数为单位,提出一种内存泄漏检测算法,用于有效检测 C 代码中出现的上述内存泄漏缺陷.本文假定在程序点 l 处,$MePtr^l=\{\langle Memory_1,Pointer_1\rangle,\langle Memory_2,Pointer_2\rangle,\ldots,\langle Memory_n,Pointer_n\rangle\}$,其中,$Memory_i$ 为动态分配后连续内存的对应区域或连续内存的某个单元对应区域,$Pointer_i=\{p_1,p_2,\ldots,p_n\}$ 为指向 $Memory_i$ 的指针变量集合.在进行内存泄漏缺陷检测时,首先对控制流图节点上与动态分配内存相关的指针进行识别,并根据归纳的迁移操作计算其存储状态信息,同时更新 $MePtr^l$.

对于一个控制流图 G,假定程序动态分配连续内存对应区域为 r,由指针 p 指向,则在不同的控制流图节点,对 $MePtr^l$ 执行不同的更新操作.

(1) 在内存分配节点,将分配的区域 r 与指针 p 添加至 $MePtr^l$.

(2) 在执行指针相关迁移操作的控制流图节点上,p 可能指向其他区域或有新指针 q 指向区域 r 或 r 的某个单元 $r[i]$,此时对 $MePtr^l$ 执行相应的更新操作.若更新之后,$MePtr^l$ 中某区域 r 对应的指针变量集合 *Pointer* 为空并且 $MePtr^l$ 中没有其余与区域 r 基址相同的区域(或存在基址相同的区域 r_1,但 r_1 对应指针变量集合均为空),则发生内存泄漏,并将区域 r 及其指针变量集合从 $MePtr^l$ 删除.

(3) 在释放内存节点,$MePtr^l$ 对释放的区域 r,$r[i]$及对应的指针集合执行删除操作.对于 p 指向的每一个区域 r 不为 *wild*,若 $O^l[\![r]\!]$不为[0,0],则释放内存大小与分配大小不一致;否则,将区域 r 及其对应的指针变量集合从 $MePtr^l$ 中删除.

(4) 在语句块结束节点或程序退出节点,$MePtr^l$ 将根据指针变量的作用域进行更新,将该语句块包含的指针变量在 $MePtr^l$ 中删除.若删除后 $MePtr^l$ 中某区域 r 对应的指针变量集合 *Pointer* 为空并且 $MePtr^l$ 中没有其余与区域 r 基址相同的区域(或存在基址相同的区域 r_1,但 r_1 对应指针变量集合均为空),则发生内存泄漏,并将区域 r 及其指针变量集合从 $MePtr^l$ 中删除.

下面是本文提出的内存泄漏缺陷检测算法.

算法 2. 内存泄漏缺陷检测算法.

输入:控制流图 G.

输出:缺陷检测表 $mdb=\langle line,variable,type\rangle$,包括发生内存泄漏缺陷的语句行数、指针变量及内存泄漏类型,*type*=0 表示未释放内存,*type*=1 表示释放内存大小与分配大小不符.

说明:

- *getPointer*(*r*)为获得区域 *r* 在 *MePtr* 中对应的指针变量集合;
- *getRegion*(·)为获得 *MePtr* 中所有的区域;
- *delete*(*r*)为从 *MePtr* 中删除区域 *r* 及其对应的指针变量集合;
- *delete*(*r*,*p*)为从 *MePtr* 中区域 *r* 对应的指针变量集合中删除变量 *p*;
- *node.getTreeNode*(·)为获得控制流图节点对应的抽象语法树节点 *ASTtreenode*;
- *ASTtreenode.getScope*(·)为获得抽象语法树节点上对应作用域中的变量集合.

Begin

1: **let** *MePtr*=∅
2: **for** each node in *G*
3: **if** node is *mallocNode*
4: *add Region* and *PointerSet* to *MePtr*;
5: **if** node is *migrationNode*
6: **for** each pointer *p* in node
7: compute data flow value of *p* by migration operation rules;
8: *update Region* and *PointerSet* in *MePtr*;
9: **for** each r_1 in *MePtr.getRegion*(·)
10: **if** *MePtr.getPointer*(r_1)==∅
11: **let** r_2 as other *Region* in *MePtr.getRegion*(·);
12: **if** $B^l[\![r_2]\!]\neq B^l[\![r_1]\!]$ or $B^l[\![r_2]\!]==B^l[\![r_1]\!]$ and *MePtr.getPointer*(r_2)==∅
13: *type*=0; *add line*, *MePtr.getPointer*(r_1) and *type* to *mdb*;
14: *MePtr.delete*(r_1);
15: **if** node is *freeNode*
16: **if** $P^l[\![p]\!]$!=*wild*
17: **for** *r* in $P^l[\![p]\!]$
18: **if** $O^l[\![r]\!]$!=[0,0] *type*=1; *add line*, *p and type* to *mdb*;
19: **else** *MePtr.delete*(*r*);
20: **if** node is *exitNode*
21: **let** *ASTtreenode*=*node.getTreeNode*(·);
22: **let** *scope*=*ASTtreenode.getScope*(·);
23: **for** *r* in *MePtr.getRegion*(·)
24: **for** *p* in *MePtr.getPointer*(*r*)
25: **if** *p* in *scope MePtr.delete*(*r*,*p*);
26: **for** *r*1 in *MePtr.getRegion*(·)
27: **if** *MePtr.getPointer*(r_1)==∅
28: **let** r_2 as other *Region* in *MePtr.getRegion*(·)
29: **if** $B^l[\![r_2]\!]\neq B^l[\![r_1]\!]$ or $B^l[\![r_2]\!]==B^l[\![r_1]\!]$ and *MePtr.getPointer*(r_2)==∅
30: *type*=0; *add line*, *MePtr.getPointer*(r_1) and *type* to *mdb*;
31: *MePtr.delete*(r_1);

End

对于图 2 的代码片段,$MePtr^l$ 的变化过程如图 8 所示.

$$MePtr=\varnothing \Rightarrow MePtr^{L11}=\{\langle mbm\_12\_8,\{buf\}\rangle\}$$
$$\Rightarrow MePtr^{L12}=\{\langle mbm\_12\_8,\{buf\}\rangle,\langle bm\_15,\{line\}\rangle\}$$
$$\Rightarrow MePtr^{L13}=\{\langle mbm\_12\_8,\{buf\}\rangle,\langle bm\_15,\{\cdot\}\rangle,\langle bm\_16,\{line,more\}\rangle\}$$
$$\Rightarrow MePtr^{L15}=\{\langle mbm\_12\_8,\{buf\}\rangle,\langle bm\_15,\{\cdot\}\rangle,\langle bm\_16,\{more\}\rangle\}$$
$$\Rightarrow MePtr^{L16}=\{\langle mbm\_12\_8,\{\ \}\rangle,\langle bm\_15,\{\cdot\}\rangle,\langle bm\_16,\{more\}\rangle,\langle bm\_18,\{buf\}\rangle\}$$

Fig.8 Part of the change in $MePtr^l$

图 8 $MePtr^l$ 的部分变化过程

5 实 验

本文的实验是在DTSC[30]缺陷检测系统基础上做进一步的研究.DTSC是一款静态缺陷检测工具,它通过分析程序源代码识别并得出代码中可能存在的缺陷,其缺陷检测流程如图 9 所示.在 DTSC 检测缺陷过程中,源程序依次转换为抽象语法树、符号表、控制流图等结构,同时利用函数调用关系、定义-使用链及数据流分析实现缺陷检测.为了验证本文所提方法的有效性,在 DTSC 系统中实现了本文提出的 SeqMM 模型及分析方法,通过对 5 个开源工程进行迁移操作、谓词操作的统计验证本文研究的必要性,并对本文提出的内存泄漏缺陷检测算法的精度与效率进行对比分析.

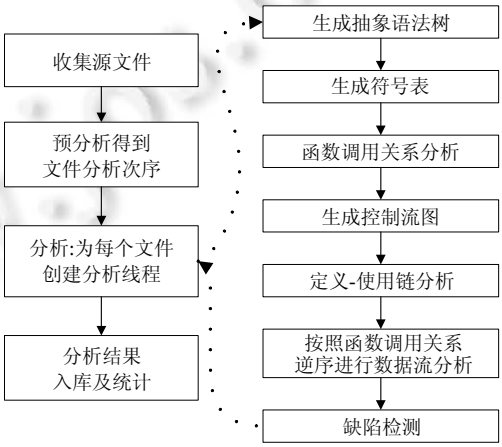


Fig.9 Analysis stages of DTSC

图 9 DTSC 分析流程

5.1 访问顺序存储结构迁移、谓词操作统计

本文对 5 个开源工程中上述指针访问顺序存储结构的迁移操作、谓词操作统计结果如表 5 所示.本文按照操作的出现顺序依次对每个操作进行编号,o_1 代表 $p=q+expr$,o_{11} 代表 $p>=\&a$.

由表 5 可以看出,迁移操作在 C 程序中出现频率较高,地址访问迁移操作大约为 8 个/KLOC,值访问迁移操作大约为 11 个/KLOC.在地址访问迁移操作中,超过 50%为指针的自加操作,超过 41%为指针变量与整型表达式组成的地址表达式之间的赋值操作,最少的是指针的对数组元素取地址与整型表达式组成的地址表达式的赋值.在值访问操作中,通过对指针变量直接取值或利用指针变量引用顺序存储结构均在 C 程序中被大量使用,其中通过指针引用顺序存储结构元素约占 53.8%.

此外,由表 5 可以看出,指针相关谓词操作在 C 程序中较少出现,其中,使用最高为指针与整型表达式组成的地址表达式的比较;同时,指针与取地址表达式的比较在 5 个开源工程中未出现.

Table 5 Operation statistics

表 5 操作统计

工程	代码行	地址访问迁移操作					值访问迁移操作		谓词操作			
		o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_{10}	o_{11}
antiword-0.37	24 315	128	8	1	43	2	141	371	1	0	0	0
readline-6.1	21 460	47	0	2	72	6	147	157	5	0	0	0
make3.81	23 284	193	3	0	355	54	174	161	9	3	0	0
uucp-1.07	52 595	272	16	1	338	6	289	401	8	0	0	0
openssl-0.9.8	226 722	593	34	23	689	97	1 023	977	31	0	0	0
Total	348 376	1 233	61	27	1 497	165	1 774	2 067	54	3	0	0

综上所述,顺序存储结构及其上的操作在C程序中被广泛使用.本文的工作聚焦这一方面,具有针对性,能够有效提高数据流分析的准确性,为检测 C 程序中顺序存储结构相关的缺陷奠定基础.

5.2 内存泄漏缺陷检测对比实验

本文提出的内存泄漏缺陷检测算法已在 DTSC 中实现.本文在上述开源工程中手工植入第 4 节提出的两类内存泄漏缺陷,并使用 DTSC_SeqMM,Klocwork12,DTSC_RSTVL 对工程分别进行内存泄漏的缺陷检测,以此验证 DTSC_SeqMM 及提出的内存泄漏检测算法的有效性,实验结果见表 6.表 6 中列出了被测项目的分析时间、不同工具测试输出的检查点(IP)、缺陷数(RD)及相对漏报(RFN).相对漏报数指的是不同工具间的相对于缺陷总数的漏报数量,缺陷总数(FB)为 3 个工具测出的缺陷总计.

衡量检测精度的误报率(FPR)与相对漏报率(FNR)分别为:

$$FPR = \frac{IP - RD}{IP},$$

$$FNR = \frac{FB - RD}{FB}.$$

经过人工确认并对检测结果进行对比分析可得,DTSC_SeqMM 检测出的缺陷包含 Klocwork12 与 DTSC_RSTVL 检测出的所有缺陷,对于所有实验用例,DTSC_SeqMM,Klocwork12,DTSC_RSTVL 的误报率分别为 27.9%,14.4%,35.8%,相对漏报率分别为 0,10.7%,7.5%.说明本文实现的 DTSC_SeqMM 与内存泄漏缺陷检测算法能够有效分析 C 程序,并降低漏报.

Table 6 Memory leak detection results

表 6 内存泄漏检测结果

工程	DTSC_SeqMM		Klocwork12		DTSC_RSTVL	
	IP/RD/RFN	Time (s)	IP/RD/RFN	Time (s)	IP/RD/RFN	Time (s)
antiword-0.37	12/11/0	1327	9/8/3	235	13/11/0	1210
readline-6.1	13/11/0	801	10/9/2	327	11/10/1	693
make3.81	17/13/0	953	13/12/1	389	17/12/1	897
uucp-1.07	24/19/0	4 781	21/17/2	2 193	26/17/2	3 662
openssl-0.9.8	63/39/0	11 343	44/37/2	6 779	67/36/3	1 0987
Total	129/93/0	19 205	97/83/10	9 923	134/86/7	17 449

图 10 为本文手工植入的释放空间与分配空间不符的内存泄漏缺陷.

```
File: readline-6.1/examples/rlfe/rlfe.c
L387: static char empty_string[1]="·";
…
L650: char*test2_implanted=empty_string;
L651: test2_implanted=malloc(buf_count+1);
L652: test2_implanted[buf_count]='\0';
L653: test2_implanted++;
L654: free(test2_implanted);
```

Fig.10 Implanted example of a memory leak defect

图 10 植入内存泄漏缺陷示例

Klockwork12 与 DTSC_SeqMM 均能检测出因 654 行 *test2_implanted* 的释放而造成的内存泄漏,而 DTSC_RSTVL 却漏报了该缺陷.在 653 行指针 *test2_implanted* 执行的自加操作导致其偏移一个单元,因此在 654 行对其释放,则使得在 651 行分配成功的内存未完全得到释放,发生内存泄漏.因为 DTSC_RSTVL 缺少对指针偏移的刻画,故无法对指针访问顺序存储结构的迁移操作与谓词操作进行识别,导致了此内存泄漏缺陷的漏报.

同时,从表 6 中也可以清晰地看出,DTSC_SeqMM 较其余两种检测工具时间开销增加明显.因为 DTSC_SeqMM 比 DTSC_RSTVL 能够描述更多的变量与操作信息,同时在检测内存泄露时,需要对 $MePtr^{l}$ 频繁地进行查询与修改操作,故在检测过程中效率有所降低.

综上所述,本文的实验结果表明,本文提出的数据流分析方法能够对带顺序存储结构的 C 程序进行有效的描述.同时,本文的内存泄漏缺陷算法对本文提出的两种类型缺陷具有准确的检测效果.

6 结束语

本文通过对顺序存储结构的刻画与描述进行研究,提出了用于顺序存储结构数据流分析的抽象内存模型 SeqMM,总结指针相关的迁移与谓词操作,并对遍历顺序存储结构的循环操作、函数调用时形参指针引用顺序存储结构元素进行分析.该模型将指针的指向关系与数值性质进行有效结合,对访问顺序存储结构的操作进行分析,解决了现有数据流方法未能对顺序存储结构进行精确分析等问题.同时,将上述分析结果应用于内存泄漏的缺陷检测.实验结果表明,本文提出的内存泄漏缺陷检测算法在效率降低的可接受范围内,实现对工程的有效检测.在下一步的工作中,我们将着重提高设计模型的分析能力,降低数据流分析结果的误报率,在本文工作的基础上进行完善提高.

References:

[1] 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 with English abstract). http://www.jos.org.cn/1000-9825/4532.htm [doi: 10.13328/j.cnki.jos.004532]

[2] James CK. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394.

[3] Xu ZX, Kremenek T, Zhang J. A memory model for static analysis of C programs. In: Proc. of the Int'l Conf. on Leveraging Applications of Formal Methods. 2010. 535−548.

[4] Zhang J. Symbolic execution of program paths involving pointer structure variables. In: Proc. of the Int'l Conf. on Quality Software. 2004. 87−92.

[5] Hackett B, Rugina R. Region-based shape analysis with tracked locations. ACM SIGPLAN Notices, 2005,40(1):310−323. [doi: 10.1145/1040305.1040331]

[6] Dong LM, Wang J, Chen LQ, Liu JC. Field-sensitive memory model for memory safety of heap-manipulating programs. Computer Science, 2012,39(9):109−114 (in Chinese with English abstract). [doi: CNKI:SUN:JSJA.0.2012-09-028]

[7] Zhao YS, Wang YW, Gong YZ, Chen HH, Xiao Q, Yang ZH. STVL: Improve the precision of static defect detection with symbolic three-values logic. In: Proc. of the 18th Asia Pacific Software Engineering Conf. 2011. 179−186. [doi: 10.1109/APSEC.2011.23]

[8] Yin BH, Chen LQ, Wang J. Analysis of program with pointer arithmetic by combining points to and numer. Computer Science, 2015,42(7):32−37 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2015.7.008]

[9] Steensgaard B. Points-to analysis in almost linear time. In: Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 1996. 32−41. [doi: 10.1145/237721.237727]

[10] Andersen LO. Program analysis and specialization for the C programming language. University of Cophenhagen, 1994.

[11] Hackett B, Aiken A. How is aliasing used in systems software? In: Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2006. 69−80. [doi: 10.1145/1181775.1181785]

[12] Kang HG, Han T. A bottom-up pointer analysis using the update history. Information and Software Technology, 2009,51(4):691−707. [doi: 10.1016/j.infsof.2008.11.003]

[13] Yu HT, Xue JL, Huo W, Feng XB, Zhang ZQ. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proc. of the 8th Int'l Symp. on Code Generation and Optimization. 2010. 218−229. [doi: 10.1145/1772954.1772985]

[14] De A, D'Souza D. Scalable flow-sensitive pointer analysis for java with strong updates. In: Proc. of the 26th European Conf. on Object-oriented Programming. 2012. 665−687. [doi: 10.1007/978-3-642-31057-7-29]

[15] Feng Y, Wang XY, Dillig I, Dillig T. Bottom-up context-sensitive pointer analysis for Java. In: Proc. of the 13th Asian Symp.on Programming Languages and Systems. 2015. 465−484. [doi: 10.1007/978-3-319-26529-2_25]

[16] Sui YL, Xue JL. On-demand strong update analysis via value-flow refinement. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2016. 460−473. [doi: 10.1145/2950290.2950296]

[17] Bush WR, Pincus JD, Sielaff DJ. A static analyzer for finding dynamic programming errors. Software Practice and Experience, 2001,30(7):775−802. [doi: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO]

[18] Dillig I, Dillig T, Aiken A. Sound, complete and scalable path-sensitive analysis. ACM SIGPLAN Notices, 2008,43(6):270−280. [doi: 10.1145/1379022.1375615]

[19] Gutzmann T, Lundberg J, Lowe W. Towards path-sensitive points-to analysis. In: Proc. of the 7th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. 2007. 59−68. [doi: 10.1109/SCAM.2007.26]

[20] Sui YL, Ye S, Xue JL, Yew PC. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In: Proc. of the 9th Asian Symp. on Programming Languages and Systems. 2011. 155−171. [doi: 10.1007/978-3-642-25318-8_14]

[21] Pathade K, Khedker UP. Computing partially path-sensitive MFP solution in data flow analysis. In: Proc. of the 27th Int'l Conf. on Compiler Construction. 2018. 37−47. [doi: 10.1145/3178372.3179497]

[22] Litvak S, Dor N, Bodik R, Rinetzky N, Sagiv M. Field-sensitive program dependence analysis. In: Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2010. 287−296. [doi: 10.1145/1882291.1882334]

[23] Cousot P, Cousot R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Proc. of the 4th Int'l Symp. on Programming Language Implementation and Logic Programming. 1992. 269−296. [doi: 10.1007/3-540-55844-6_142]

[24] Šor V. Statistical approach for memory leak dectection in Java applications [Ph.D. Thesis]. Tartu: University of Tartu, 2015.

[25] Šor V, Oü P, Treier T, Srirama SN. Improving statistical approach for memory leak detection using machine learning. In: Proc. of the 29th IEEE Int'l Conf. on Software Maintenance. 2013. 544−547. [doi: 10.1109/ICSM.2013.92]

[26] Šor V, Srirama SN, Salnikov-Tarnovski N. Memory leak detection in Plumbr. Software: Practice and Experience, 2015,45(10): 1307−1330. [doi: 10.1002/spe.2275]

[27] Andrzejak A, Eichler F, Ghanavati M. Defect of memory leaks in C/C++ code via machine learning. In: Proc. of the 28th IEEE Int'l Symp. on Software Reliability Engineering Workshops. 2017. 252−258. [doi: 10.1109/ISSREW.2017.72]

[28] Zhu YW, Zuo ZQ, Wang LZ, Li XD. Memory leak intelligent detection method for C programs. Ruan Jian Xue Bao/Journal of Software, 2019,30(5):1330−1341 (in Chainexe with English abstract). http://www.jos.org.cn/1000-9825/5715.htm [doi: 10.13328/j.cnki.jos.005715]

[29] Li X, Zhou Y, Li MC, Chen YJ, Xu GQ, Wang LZ, Li XD. Automatically validating static memory leak warnings for C/C++ programs. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):827−844 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5189htm [doi: 10.13328/j.cnki.jos.005189]

[30] Yang ZH, Gong YZ, Xiao Q, Wang YW. A defect model based testing system. Journal of Beijing University of Posts and Telecommunications, 2008,31(5):1−4 (in Chinese with English abstract).

附中文参考文献:

[1] 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 [doi: 10.13328/j.cnki.jos.004532]

[6] 董龙明,王戟,陈立前,刘江潮.一种面向堆操作程序内存安全性的域敏感内存模型.计算机科学,2012,39(9):109−114. [doi: CNKI:SUN:JSJA.0.2012-09-028]

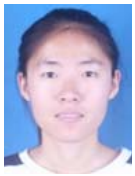
[8] 尹帮虎,陈立前,王戟.基于指向与数值抽象的带指针算术程序的分析方法.计算机科学,2015,42(7):32−37. [doi: 10.11896/j.issn.1002-137X.2015.7.008]

[28] 朱亚伟,左志强,王林章,李宣东.C 程序内存泄漏智能化检测方法.软件学报,2019,30(5):1330−1341. http://www.jos.org.cn/1000-9825/5715.htm [doi: 10.13328/j.cnki.jos.005715]

[29] 李筱,周严,李孟宸,陈园军,Xu GQ,王林章,李宣东.C/C++程序静态内存泄漏警报自动确认方法.软件学报,2017,28(4): 827−844. http://www.jos.org.cn/1000-9825/5189.htm [doi: 10.13328/j.cnki.jos.005189]

[30] 杨朝红,宫云战,肖庆,王雅文.基于软件缺陷模型的测试系统.北京邮电大学学报,2008,31(5):1−4.

王淑栋(1973－),女,山东青岛人,博士,教授,博士生导师,主要研究领域为生物计算,软件工程.



尹文静(1995－),女,硕士生,CCF 学生会员,主要研究领域为数据流分析,缺陷检测.

董玉坤(1981－),男,博士,讲师,CCF 专业会员,主要研究领域为缺陷检测,程序自动修复.

张莉(1994－),女,硕士生,CCF 学生会员,主要研究领域为程序自动修复,软件工程.

刘浩(1992－),男,硕士生,主要研究领域为缺陷警报关联,数据流分析.