基于区域内存模型的空指针引用缺陷检测

董玉坤[1,2],宫云战[1],金大海[1]
(1.北京邮电大学网络与交换技术国家重点实验室,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)

摘 要: 为了实现对C程序中空指针引用的充分检测,本文提出了一种基于区域内存模型的空指针引用缺陷检测方法.首先,提出了基于区域的符号化三值逻辑(Region-based Symbolic Three-Valued Logic,RSTVL),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;其次,给出了基于抽象语法树与函数摘要识别被引用指针方法;最后,结合基于RSTVL的数据流分析结果,将对被引用指针的检测转换为对相应区域的检测,给出了空指针引用缺陷检测的方法,通过函数摘要实现过程间的空指针引用缺陷检测.对比实验结果表明,本文方法在保证一定检测准确率的前提下,能够极大的减少空指针引用缺陷的漏报.

关键词: 空指针引用;内存模型;静态分析;函数摘要;缺陷检测
中图分类号: TP311.5 **文献标识码:** A **文章编号:** 0372-2112(2014)09-1744-09
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2014.09.013

Null Pointer Dereference Defect Detected Based on Region-Based Memory Model

DONG Yu-kun[1,2],GONG Yun-zhan[1],JIN Da-hai[1]
(1. *State Key Lab of Networking and Switching Tech, Beijing University of Posts and Telecommunications, Beijing* 100876, *China*;
2. *College of Computer and Communication Engineering, China University of Petroleum, Qingdao, Shandong* 266580, *China*)

Abstract: In order to fully detect null pointer dereference for C procedures, this paper introduces a method based on region-based memory model. Firstly, region-based symbolic three-valued logic (RSTVL) is proposed, which can describe shape of data structures, all kinds of memory states and relations of addressable expressions. Then, an approach to fully recognizing pointer dereferences based on abstract syntax tree and procedure summary is introduced. Furthermore, this paper introduces a null pointer dereference detection method, which translates pointer dereference detection into region detection applying the result of data flow analysis based on RSTVL, and detects interprocedural null pointer dereference based on procedure summary. Experiment results show that compared with DTSC_STVL and Klocwork9, the proposed method could dramatically reduce null pointer dereference false negative on the precondition of guarantee the detection precision.

Key words: null pointer dereference; memory model; static analysis; function summary; defect detection

1 引言

空指针引用是软件中最常见且难以完全消除的一类缺陷,当前,对空指针引用测试的方法可分为动态方法[1,2]和静态方法[3~7].动态的空指针引用测试方法能够准确的定位出空指针引用错误,但因为受限于测试所用的输入用例,不能检查所有的空指针引用缺陷.静态方法在不运行软件前提下检查是否存在空指针,目前已有许多静态的空指针引用测试方法,这些方法可分为两大类,第一类可称之为空指针引用检测[3~5],第二类可称之为指针引用验证[6,7].空指针引用检测的方法大多是首先对程序进行数据流分析或单纯的指针分析,基于分析结果检测程序中被引用的指针是否是空指针.指针引用验证的方法采取需求驱动的思想,首先识别出被引用的指针变量,从指针引用所在的程序点逆向的沿着程序的控制流,分析是否有满足该指针为空指针的可能.

空指针引用检测的方法具有效率高,能够与其它类型缺陷一并检测的优点,通常基于数据流迭代算法分析出控制流图上每个程序点的程序状态,但因为静态分析技术本身的局限,数据流分析结果不可能既是完备的又

收稿日期:2014-01-23;修回日期:2014-04-24;责任编辑:孙瑶
基金项目:国家自然科学基金(No.91318301,No.61202080);国家863高技术研究发展计划(No.2012AA011201)

是可靠的,这将导致缺陷检测存在漏报或误报.如果分析所得的程序状态未包含实际运行的可能,将会导致漏报;如果未能识别出所有被引用的指针,也将会导致漏报.对某些安全攸关的软件,在误报率能够接受的前提下,低漏报甚至零漏报尤甚重要.

目前的大部分静态分析方法及工具难以实现空指针引用缺陷检测的零漏报,对于 C 语言开发的软件尤为突出.主要是因为 C 程序包含结构体、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;并且程序中可能出现复杂的指针引用,特别是指针被广泛作为参数使用,尤其是指向复合类型变量的参数,这都导致对程序难以进行可靠的分析,特别是对指针分析,并加剧了空指针引用检测的困难.

例如对于图 1 程序片断(*a*),第 7 行语句中的指针引用表达式 * pst[*i*] -> *m*,隐含着三个被引用的指针表达式:pst,pst[*i*],pst[*i*] -> *m*,这三个指针均需要被识别出并确定安全才能确保 * pst[*i*] -> *m* 是安全指针引用.

对于图 1 程序片断(*b*),当第 4 行语句中指针 *p* 被引用,而 *p* 与 ps -> *a* 具有别名关系,因为 ps 是形参,在方法 *f*2 内是无法确定 ps -> *a* 的具体的指向信息的,需要根据具体的调用点上下文信息才能确定;因此,在第 9 行调用 *f*2 时,需要保证传递的实参能够保证被调用函数 *f*2 的 ps -> *a* 不能为空,否则会产生空指针引用.

```
typedef struct{
    int * m;
}st;
void f1 (st * * pst){
    int i = 0;
    for(;i<9;i++){
        int j = * pst[i] -> m;
    }
}
```
(*a*)

```
typedef struct{int * a;}st;
int f2(st * ps){
    int * p = ps -> a;
    * p = 2;
}
void f3(){
    st s;
    s.a = NULL;
    f2(&s);
}
```
(*b*)

图 1　程序示例

由图 1 的两个程序片断可以看出,因为 C 程序中具有复杂的表达式,导致了对指针表达式分析以及被引用指针变量识别的复杂性.指针变量作为参数进行传递,需要分析出被调用的形参对应的调用点上的实参,以实现过程间的空指针引用检测.

本文将主要对可寻址表达式间的各种关联的描述,被引用指针的充分识别,以及各种类型的指针参数被引用的检测等情况进行研究.首先,为了在数据流分析阶段全面考虑变量间的各种关联以保证分析的可靠性,本文提出了基于区域的符号化三值逻辑(Region-based Symbolic Three-Valued Logic,RSTVL),RSTVL 应用区域描述每个内存对象的存储,并能够描述可寻址表达式[8]间的各种关联.区域是对物理的内存区域的抽象描述,通过静态分析可获得每个程序点上指针所指向的区域,进而将对是否为指针引用的判定问题转化为指针所指向的区域是否是安全区域的判定问题.

本文基于可寻址表达式与抽象语法树节点的映射关系,从抽象语法树上识别出所有被引用的指针,并基于数据流分析的结果得到被引用指针的指向属性.如果指向属性是未知的,表明该指针指向了外部变量的未知区域,则将这些未知区域对应的内存对象的父变量添加到所在函数的空指针引用缺陷模式前置约束(NPDPreSummary)中;在函数调用点,获得 NPDPreSummary 中的指针在调用点上对应的指针变量,判定所对应的调用点上的指针变量所指向区域是否安全,以判定是否会产生空指针引用.

本文实现了低漏报的空指针引用缺陷检测工具 DTSC_RSTVL,对 8 个实际工程进行了实验,结果表明 DTSC_RSTVL 的误报率平均为 49%,无相对漏报.

2　DTSC 静态缺陷检测框架

DTSC 是一个面向具体缺陷模式的静态缺陷检测工具,通过静态地分析程序源代码能够测试 100 多种缺陷,整体分析流程如图 2 所示.

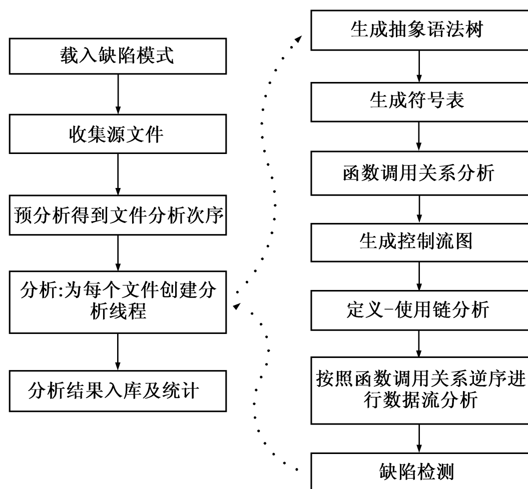

图2　DTSC整体分析流程

首先,载入需要检测的缺陷模式;其次是收集需要检测的文件及相关的文件,生成源文件的中间文件;再分析文件间的依赖关系得到文件的分析次序;然后为每个中间文件创建分析线程进行正式的分析与缺陷检测;最后是检测结果保存及统计.

为每个文件的分析与缺陷检测可分为七个步骤:

(1)为中间文件生成抽象语法树;

(2)遍历抽象语法树识别出符号并生成符号表;

(3)分析函数的调用关系以生成函数调用关系图；

(4)为每一个函数生成控制流图；

(5)生成变量的定义-使用链；

(6)按照函数的调用关系逆序的流敏感、域敏感及上下文敏感的数据流分析，分析得到每个程序点上的抽象存储状态；

(7)基于数据流分析的结果进行缺陷检测.

3 基于区域的符号化三值逻辑

3.1 可寻址表达式

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 标准所支持 C 程序的内存对象 var 的语法可归纳为：

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

其中 v 为顶级变量，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.C99 所支持的 C 程序的可寻址表达式 e 的语法可归纳为：

$$e::=v\,|\,e.f\,|\,e\text{->}f\,|\,e[\text{exp}]\,|\,(e)\,|\,*e\,|\,m(\text{exp})$$

其中 m 为函数指针.

$$*e::=*e'\,|\,*(++e')\,|\,*(--e')\,|\,*(e'++)\,|\,*(e'--)\,|\,*(e'\ \text{op}\ \text{exp}')$$

其中 e' 为指针，op = + | -，exp′为整型表达式.

图 1 程序片断(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.

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.指针变量为对其引用的可寻址表达式的父变量，e 为 * e 的父变量.

对可寻址表达式进行操作将影响内存单元的存储状态，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.因为内存对象的内存单元间的左值或右值间的关联，导致可寻址表达式间存在三种关联：

(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；

(2)左值间的关联，称为层次关系，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，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.

对于指针类型的形参和全局指针表达式，引入扩充变量来抽象地表示其指向信息及成员.扩充变量的引入规则为：

(1)对于 n 级指针 p，扩充出 * p、* * p、* * * p、…等共 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，…，f_n 共 n 个成员，则扩充出 $s.f_1$、$s.f_2$、…、$s.f_n$ 共 n 个变量.

例如，对图 1 中程序片断(b)函数 $f2$ 的形参 ps，ps 为一级指针，引入扩充变量 * ps；* ps 为结构体类型的可寻址表达式，而且该结构体只有一个成员 a，因此再引入扩充变量(* ps).a.

定义 4(外部变量) 对于一个函数 p，p 的参数与可用的全局变量及它们的扩充变量为 p 的外部变量.

3.2 基于区域的符号化三值逻辑

为全面描述可寻址表达式间的各种关联.本文结合基于区域的抽象内存模型[9]与 STVL[10]的优点，提出了 RSTVL.

定义 5 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[11]，Domain 表示取值区间.

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，x〉表示.x 的含义由 Var 的类型决定，如果 Var 是数组类型，x 是{〈i，Region〉}，$i\in\mathbf{N}$ 是数组 Var 的下标；如果 Var 是结构体，x 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，其中空指针的区域编号为“null”，野指针的区域编号为“wild”.

定义 6 将对 v、var.f、var[n]等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，将为参数或全局变量分配的区域称为未知区域，对这三种区域统称为可操作区域.将“null”与“wild”标识的区域称为不可操作区域.动态区域与未知区域经过非空判断后变为安全区域，动态区域与未知区域经过是空判断后变为不可操作区域.

对于取值区间，采用区间抽象域[12]的方法，每个 Symbol 的取值用区间表示.本文的区间采用区间集抽象，分为数值型区间与指针型区间两大类，其中指针型区间 PointerDomain 的元素为区域的编号，标识所指向的区域.RSTVL 的区间及其上的操作构成完备格〈L，$\leqslant$，$\sqcup$，$\sqcap$，$\perp$，$\top$〉，其中$\perp$为空集，数值型区间的$\top$为[$-\infty$，$+\infty$]，指针型区间的$\top$为“null”、“wild”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.基于 RSTVL 的数据流分析[12]可将数据流值映射

为在格上的操作.

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,可满足流敏感、域敏感的数据流分析.在每个程序点 l,R^l 表示在 l 处能够被访问的区域集合,$S^l=\langle s,\mathrm{domain}\rangle$表示在 l 处使用的符号及其区间集合.每个程序点 l 都有一个抽象存储 $\rho^l=(\rho_v^l,\rho_r^l,\rho_f^l)$.其中:

- ρ_v^l: $\mathrm{Var}\rightarrow R^l$,映射一个内存对象到一个区域;
- ρ_r^l: $R^l\rightarrow R^l$,表示区域间的指向关系;
- ρ_f^l: $(R^l\times F)\rightarrow R^l$,映射一个复合类型内存对象的成员到一个区域.

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集合的策略:

- $R^l[\![\mathrm{var}]\!]=\rho_v^l(\mathrm{var})$
- $R^l[\![e.f]\!]=\bigcup_{r\in R^l[e]}\rho_f^l(r,f)$
- $R^l[\![e[i]]\!]=\bigcup_{r\in R^l[e]}\rho_f^l(r,i)$
- $R^l[\![*e]\!]=\bigcup_{r\in R^l[e]}\rho_r^l(r)$
- $R^l[\![(e)]\!]=R^l[\![e]\!]$
- $R^l[\![e->f]\!]=\bigcup_{r\in R^l[e]}\{\bigcup_{r'\in\rho_r^l(r)}\rho_f^l(r',f)\}$

4　空指针引用检测

空指针引用作为一类典型的程序源代码级缺陷,可被总结为一种缺陷模式,其缺陷特征表现为与一个指针变量相关,该指针变量是或可能是空指针.

4.1　指针指向属性

对于每一个被引用的指针,可根据其指向属性进行是否是空指针引用的判定.其中指向属性被描述为一个格:$AL_{PTR}=(V_{PTR},F_{join},F_{meet})$,其哈斯图如图 3 所示.

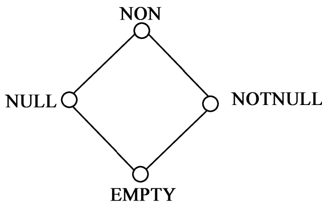

图3　AL_{PTR}的哈斯图

V_{PTR}刻画指针指向属性值的集合,可以有效的表示指针被引用时的安全性,方便的应用于空指针引用的检测.EMPTY 表示初始属性格值,NULL 表示指针指向空地址,NOTNULL 表示指针指向安全的内存区域,NON (NULL_OR_NOTNULL)表示指针可能指向空地址.当一个指针被引用时,指向属性为 NULL 则肯定发生空指针引用,指向属性为 NON 则可能发生空指针引用.

$F_{join}: V_{PTR}\times V_{PTR}\rightarrow V_{PTR}$是 AL_{PTR}的最大下界函数.

$F_{meet}: V_{PTR}\times V_{PTR}\rightarrow V_{PTR}$是 AL_{PTR}的最小上界函数.

为了更好的表达程序语义,为 V_{PTR}引入 UNKNOWN 表示指针的指向不确定,用于初始化外部变量的指向属性,UNKNOWN 与其他指向属性值 X 在格上的操作为:

$F_{join}(X,\mathrm{UNKNOWN})=X$

$F_{meet}(\mathrm{NOTNULL},\mathrm{UNKNOWN})=\mathrm{UNKNOWN}$

$F_{meet}(\mathrm{NULL},\mathrm{UNKNOWN})=\mathrm{NON}$

$F_{meet}(\mathrm{NON},\mathrm{UNKNOWN})=\mathrm{NON}$

$F_{meet}(\mathrm{EMPTY},\mathrm{UNKNOWN})=\mathrm{UNKNOWN}$

$F_{meet}(\mathrm{UNKNOWN},\mathrm{UNKNOWN})=\mathrm{UNKNOWN}$

用 $T^l[\![r_n]\!]$表示在程序点 l 上编号为 r_n的区域的区域类型,区域类型共有:安全、动态、未知、不可操作;令 pd 表示指针表达式 e_p在程序点 l 的指针型区间,指向属性的抽象化函数 α_ρ^l 定义为:

$$\alpha_\rho^l(\mathrm{pd})=\begin{cases}\mathrm{EMPTY}, & \mathrm{pd}==\varnothing\\ \mathrm{NULL}, & \mathrm{pd}==\{\text{“null”}\}\\ \mathrm{NOTNULL}, & \forall r_n\in\mathrm{pd}, T^l[\![r_n]\!]\ \text{is 安全}\\ \mathrm{UNKNOWN}, & (\forall r_n\in\mathrm{pd}, T^l[\![r_n]\!]\ \text{is 安全}\\ & \text{or 未知})\ \text{and}\ (\exists r_n\in\mathrm{pd},\\ & T^l[\![r_n]\!]\ \text{is 未知})\\ \mathrm{NON}, & \text{others}\end{cases}\tag{1}$$

4.2　指针引用判定规则

用 $D^l[\![e]\!]$表示在程序点 l 上可寻址表达式 e 的取值区间,ψ 表示空指针引用缺陷检测点的上下文环境,NPD 表示肯定空指针引用,DPD 表示可能空指针引用,SPD 表示安全指针引用.指针引用判定规则如下:

$$\frac{\alpha_\rho(D^l[\![e_p]\!])==\mathrm{NULL}}{\psi\vdash\mathrm{NPD}}\tag{2}$$

$$\frac{\alpha_\rho(D^l[\![e_p]\!])==\mathrm{NON}}{\psi\vdash\mathrm{DPD}}\tag{3}$$

$$\frac{\alpha_\rho(D^l[\![e_p]\!])==\mathrm{NOTNULL}}{\psi\vdash\mathrm{SPD}}\tag{4}$$

如果指针 e_p 在程序点 l 上的指向属性为 EMPTY,说明程序点 l 出现了矛盾,为不可达程序点,e_p 不可能被实际引用,不需要进行判定.如果指针 e_p 在程序点 l 上的指向属性为 UNKNOWN,表明在当前的函数内不能确定是否为空指针,需要将与其具有别名关系且指向属性也是 UNKNOWN 的外部指针变量添加到当前函数的 NPDPreSummary 中.

4.3　空指针引用缺陷模式的前置约束

在静态缺陷检测阶段,分析某个函数时往往无法确定其上层调用函数的上下文信息,需要将是否存在缺陷的判定权沿着调用关系向上传递,本文将此类信息抽象为前置约束信息保存在函数摘要[12,13]中,其中

函数摘要被存放于一个全局环境中. 函数摘要的前置约束相当于将缺陷判定权转移给被测函数的调用者,在遇到包含前置约束信息的明确的上下文时进行判定;如果上下文依然不确定,则继续向上提交该前置约束信息,直到遇到足够明确的上下文信息.

对于空指针引用缺陷的检测,如果指针 e_p 在程序点 l 上被引用且其指向属性为 UNKNOWN,说明指向了某些未知区域,则将这些未知区域所关联内存对象的父变量添加到所在函数的 NPDPreSummary 中,算法 2 的第 5 – 9 行描述了相关处理过程. 例如,图 1 中程序片段(b)中函数 $f2$ 的在第 3 行对参数 ps 进行了引用,而 ps 的指向属性为 UNKNOWN,在函数 $f2$ 内无法判定 ps 是否是空指针,需要将该缺陷的判定权提交给 $f2$ 的调用者 $f3$.

4.4 指针引用检测对象识别

定义 7(指针引用检测对象) 具有明确指向属性的指针,且其所指向的地址被访问,通过检测该指针变量是否为空指针可以判定是否产生空指针引用.

指针引用检测对象的识别需要首先是识别出被引用指针,本文基于抽象语法树识别被引用指针. DTSC 基于 C 的文法生成被测 C 文件的抽象语法树,可寻址表达式与抽象语法树上的节点具有映射关联,在生成符号表阶段,DTSC 将识别出的可寻址表达式添加在抽象语法树的相应节点上[8]. 其中跟可寻址表达式密切相关的三类节点是 UnaryExpression, PostfixExpression 与 PrimaryExpression,三类节点的 BNF 描述如下:

UnaryExpression: : = PostfixExpression |“ + + ” UnaryExpression |“ – – ”UnaryExpression | ⟨SIZEOF⟩ (UnaryExpression |“(“TypeName”)”) | UnaryOperator CastExpression,其中,UnaryOperator: : = “&” |“ * ” |“ + ” | “ – ” |“ ~ ” |“!”);

PostfixExpression: : = PrimaryExpression(“.” ⟨IDENTIFIER⟩ |“[“Expression”]” | “(“(ArgumentExpressionList)?”)” | “ –> ”⟨IDENTIFIER⟩ |“ + + ” |“ – – ”) * ;

PrimaryExpression: : = ⟨IDENTIFIER⟩ | Constant | “(“Expression”)”.

对于指针表达式 e_p,在语法层面 e_p 被引用的方式有三种情况: * e_p | e_p –> f | e_p[exp],即指针引用表达式也是可寻址表达式,因此可通过搜索抽象语法树获得每一个指针引用,进而获得每一个被引用的指针. 本文采用 XPath 从抽象语法树上进行搜索,其中对于指针引用 * e_p 引用的指针 e_p,搜索其对应的抽象语法树节点的查询语句为:

.//AssignmentExpression//UnaryExpression [/UnaryOperator[@Operators = ' * ']]/UnaryExpression.

其中对于指针引用 e_p –> f 与 e_p[exp]引用的指针 e_p,搜索其对应的抽象语法树节点的查询语句为:

.//AssignmentExpression//UnaryExpression/PostfixExpression[./PrimaryExpression][contains(@Operators,'[') orcontains(@Operators,' –> ')].

例如,对于图 1 程序片断(a)中第 7 行的可寻址表达式 * pst[i] –> m,是 * e_p 类型的指针引用,则可以在相应的 UnaryExpression 节点上识别出被引用的指针 pst[i] –> m;而 pst[i] –> m 是 e_p –> f 类型的指针引用,则可以在相应的 PostfixExpression 节点上识别出 pst[i]; pst[i]是类型的指针 e_p[exp]引用,则可以在相应的 PostfixExpression 节点上识别出 pst. 即对于 * pst[i] –> m,共识别出了三个被引用指针: pst[i] –> m、pst[i]、pst.

如果指针 e_p 在程序点 l 上被引用且其指向属性为 UNKNOWN,说明指向了某些未知区域,这些未知区域为外部变量的区域,则这些未知区域所关联内存对象的父变量添加到所在函数的 NPDPreSummary 中,具体处理如算法 1 的第 5 – 9 行所示.

算法 1 空指针引用检测算法

输入:函数 f

输出:函数 f 的 NPDPreSummary f_{nps}

声明:

getPointers(f):应用 XPath 从抽象语法树上搜索函数 f 内被引用的指针与指针引用语句对应的控制流图节点

getMethods(f):应用 XPath 从抽象语法树上搜索函数 f 内调用的函数

getDereferences(m):基于算法 1 获得被调函数 m 的 NPDPreSummary 中的指针所对应的调用点上的指针集合与函数调用语句对应的控制流图节点

judgeDereference(v_{ptr}):基于指针指向属性 v_{ptr} 判定是什么类型的指针引用

```
1  let pointers⟨Variable, VexNode⟩ = getPointers(f);
2  for each m ∈ getMethods(f);
3    add getDereferences(m) to pointers;
4  for each⟨e_p, n⟩ ∈ pointers
5    if(α_ρ(D^{·n}⟦e_p⟧) = = UNKNOWN)
6      for each r_n ∈ D^{·n}⟦e_p⟧ && T^{·n}⟦r_n⟧ is 未知
7          let var = E_r⟦R_n^{·n}⟦r_n⟧⟧;
8          let fvar = getParent(var);
9          add fvar to f_nps;
10   else if(α_ρ(D^{·n}⟦e_p⟧)!  = EMPTY)
11       judgeDereference(α_ρ(D^{·n}⟦e_p⟧));
```

函数调用的语法可表示为 f(exp),基于该语法形式可从抽象语法树上搜索其对应的抽象语法树节点的查询语句为:

".//PrimaryExpression[@Method = 'true']"

当一函数被调用时,都首先需要获得该被调用函数的函数摘要,并根据调用点的上下文环境对其

NPDPreSummary 进行实例化,获得调用点上对应的指针.对于 NPDPreSummary 的实例化,关键是对其中每个被约束的指针 e_p 获得其对应的调用点上的可寻址表达式集合 e_pList,并根据调用点上由 RSFVL 描述的抽象存储,获得 e_pList 中的每个指针的指向属性,并根据指针引用判定规则进行判定是否产生了空指针引用;如果指针的指向属性为 UNKNOWN,则应用算法 1 中第 5-9 行处理将对应的外部指针添加到主调函数的 NPDPreSummary 中.这其中的关键是获得函数摘要中被约束指针对应的调用点上的可寻址表达式集合,属于形参到实参集合的映射问题,具体处理如算法 2 所示.对控制流图上的每个节点 n,定义两个程序点,$\cdot n$ 表示 n 前的程序点,$n\cdot$ 表示 n 后的程序点.用 $R_n^l[\![r_n]\!]$ 表示在程序点 l 上编号为 r_n 的区域,用 $E_r[\![r]\!]$ 表示区域 r 对应的内存对象,用 $R_f^l[\![r, m]\!]$ 表示在程序点 l 上获得 StructRegion 类型区域 r 的成员 m 的区域.

算法 2　映射形参到实参集合

输入:形参 para,调用语句的控制流图节点 n

输出:形参 para 对应的调用点上的实参集合 args⟨Variable⟩

声明:

getParents(var):获得包括变量 var 在内的按照父子关系排序的 var 的父变量集合

getArgument(var, n):获得顶级形参 var 在调用点 n 上对应的实参集合

getParent(var):获得变量 var 的父变量

getType(aexp):获得可寻址表达式 aexp 的类型,其中返回的整数值表示的类型分别为 0: v、1: aexp. f、2: aexp -> f、3: aexp[exp]、4: (aexp)、5: * aexp、6: ap(exp)

getMemName(s, var):基于结构体变量 s,获得其子变量 var 对应的成员名

```
1  let args⟨Variable⟩ = ∅;
2  let parents⟨Variable⟩ = getParents(para);
3  get firstVariable v0 in parents;
4  args = {getArgument(v0, n)};
5  for each p ∈ parents && p ! = v0
6     let vp = getParent(p);
7     let vars⟨Variable⟩ = ∅;
8     for each v ∈ args
9        for each r ∈ R^{·n}⟦v⟧
10          if getType(p) = = 1 then
11             let m = getMemName(vp, p);
12             vars ∪ = {R_f^l⟦r, m⟧};
13          else if getType(p) = = 5 then
14             vars ∪ = {V_r^{·n}⟦r⟧};
15    args = vars;
```

4.5　空指针引用检测算法

DTSC 在以每个函数为单位进行空指针引用缺陷检测,首先识别出被引用的指针,然后根据基于被引用指针的指向属性根据指针引用判定规则对其进行判定,具体处理流程如算法 1 所示.

例如,对于图 1 中程序片断(b)的函数 $f2$,在第 3 行参数 ps 被直接进行了引用,第 4 行的被引用指针 p 被分析为是对(* ps). a 所指向的区域的访问,而 ps 与(* ps). a 的指向属性均为 UNKNOWN,因此 $f2$ 的 NPDPreSummary 为:{ps,(* ps). a}.

函数 $f3$ 在第 9 行调用 $f2$,ps 作为顶级参数而且被约束为不能是空指针,根据参数序号可知 ps 对应着实参 &s,为安全指针引用.被约束为不能为空的指针(* ps). a 的父变量集合为{ps, * ps,(* ps). a}.根据在调用点由 RSTVL 描述的抽象存储可得到:ps 对应着{&s}, * ps 对应着{s},(* ps). a 对应着{s. a}.因此,需要判定调用函数 $f2$ 前的程序点上 s. a 的指向属性.因为第 8 行语句对 s. a 赋值为 NULL,其指向属性为 NULL,因此在第 9 行将报告对 s. a 进行了空指针引用.

5　实验结果与分析

基于 RSTVL 的空指针引用缺陷检测方法已经 DTSC 中实现,并开发了一个新的版本 DTSC_RSTVL.为验证本文方法的效果,本文选取了 8 个 C 开源工程进行了相关实验,首先是识别指针引用与空指针引用检测对象的实验;并与文献[10]中的 STVL 方法开发的 DTSC_STVL,以及同类商业测试工具 Klocwork9 的检测结果进行了对比.实验结果表明,本文方法在效率略有降低且误报率可以接受的前提下,能够极大的降低空指针引用的漏报.

5.1　指针引用对象识别

通过本文方法识别出 8 个 C 工程的被引用的指针信息如表 1 所示.其中被引用的指针按照其指向属性与作用域分为:指向属性确定的局部指针(Local Known Pointer, LKP),指向属性不确定的局部指针(Local Unknown Pointer, LUP),指向属性确定的外部指针(External Known Pointer, EKP),指向属性不确定的外部指针(External Unknown Pointer, EUP),函数指针(Function Pointer, FP).

表 1　被引用指针统计

Benchmark	Files	Lines	Referenced pointers					
			LKP	LUP	EKP	EUP	FP	Total
antiword-0.37	67	24215	770	142	50	1716	5	2683
uucp-1.07	251	52595	1849	1112	401	2397	22	5781
sphinxbase-0.3	62	22517	691	203	602	2011	12	3519
optipng-0.6	68	27075	415	239	178	1258	13	2103
barcode-0.98	14	3409	176	52	249	378	8	863
make-3.81	39	24062	1542	574	154	524	25	2819
openssl-0.9.8	960	226722	10067	914	3427	8683	39	23130
openssh-4.2	164	53252	2156	175	370	3257	17	5975
Total	1625	433747	17666	3411	5431	20224	141	46873

由表 1 可知,C 程序中指针引用出现频率较高,大概 108 个/KLOC,而且 50%左右的被引用指针为 LUP 型或 EUP 型指针,在所在函数内是不能确定是否指向为空的指针,再考虑函数指针,超过一半的指针引用需要进行过程间分析,通过识别出具有明确指向属性的空指针引用缺陷检测对象才能判定.

对于表 1 中的 LUP 型与 EUP 型被引用指针,按照算法 1 中第 5－9 行的处理将对应的外部指针添加到所在函数的 NPDPreSummary 中.对于库函数,采用人工的方式创建了相应的 NPDPreSummary.在函数调用点上,根据被调用函数的 NPDPreSummary 识别出空指针引用检测对象.

共需要进行是否为空判定的指针检测对象如表 2 所示.其中根据函数摘要创建的检测对象包括:基于自定义函数的 NPDPreSummary 创建的检测对象(External Precondition Pointer, EPP),基于库函数的人工摘要的 NPDPreSummary 创建的检测对象(Lib Function Precondition Pointer, LFPP).

表 2 检测对象识别统计

Benchmark	Detection objects					
	LKP	EKP	FP	EPP	LFPP	Total
antiword-0.37	770	50	5	372	105	1370
uucp-1.07	1849	401	22	1382	1314	5325
sphinxbase-0.3	691	602	12	170	222	1767
optipng-0.6	415	178	13	579	358	1583
barcode-0.98	176	249	8	56	259	758
make-3.81	1542	154	25	358	740	2819
openssl-1.19	10076	3472	39	1501	2242	17330
openssh-4.3	2156	370	17	785	925	4308
Total	17675	5476	741	5203	6165	35260

5.2 空指针引用检测结果分析

表 3 给出了 DTSC_RSTVL 与 DTSC_STVL、Klocwork9 对 8 个工程的空指针引用检测结果,统计数据包括错误数(IP)、实错数(Bug)、相对漏报(Relative False Negative, RFN)和测试耗时.其中错误数是检测出来的结果,实错数与相对漏报数通过人工分析获得,相对漏报数指的是不同工具间的相对于实错总数的漏报数量.实错总数(Fatural Bug, FB)为三个工具测出的实错总计.

衡量检测精度的误报率(False Positive Rates, FPR)与相对漏报率(False Negative Rates, FNR)分别为:

$$FPR = \frac{IP - Bug}{IP} \tag{5}$$

$$FNR = \frac{FB - Bug}{FB} \tag{6}$$

DTSC_STVL、Klocwork9、DTSC_RSTVL 的误报率分别为:59%、32%、49%,相对漏报率分别为:21%、53%、0%,测试速度分别为:51LOC/s、131LOC/s、45LOC/s.检测结果表明 DTS_RSTVL 在测试速度略有下降、误报率可接受的前提下,极大的降低了漏报率.

通过对检测结果分析,发现 DTSC_STVL 对 LUP 型与 EUP 型空指针引用缺陷漏报较多.主要原因一是 DTSC_STVL 对指针别名处理的不够精确,未能将全部与 LUP 型指针具有别名的且指向属性为 UNKNOWN 的外部指针添加到 NPDPreSummary 中;二是对于 NPDPreSummary 中的指针,在函数调用点上对于一些扩充变量的指针,不能分析出对应的调用点上的指针,导致遗漏识别一些 EPP 型检测对象.Klocwork9 对各种类型空指针引用缺陷均有漏报,而且 EPP 型与 LFPP 型检测对象漏报偏多.

DTSC_RSTVL 能够降低漏报主要是因为采用了 RSTVL 描述变量的存储,能够全面表示可寻址表达式间的关联,而且全面的考虑了各种指针引用的情况,对于未知情况采取了保守的策略,保证了对空指针引用检测的充分性.特别是 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 DTSC_RSTVL 方法较 DTSC_STVL 能够检测出更多的对复合类型变量的成员引用的缺陷.DTSC_RSTVL 的误报率较 Klocwork 偏高,主要是对 EPP 型检测对象误报较多,误报的原因主要是 NPDPreSummary 中指针是在某种条件下才可能是空指针,而 DTSC_RSTVL 未考虑路径条件.

表 3 空指针引用缺陷检测结果统计

Benchmark	DTSC_STVL				Klocwork9				DTSC_RSTVL			
	IP	Bug	RFN	Time(s)	IP	Bug	RFN	Time(s)	IP	Bug	RFN	Time(s)
antiword-0.37	42	17	7	204	9	4	20	153	37	24	0	301
uucp-1.07	101	58	10	1924	33	29	39	560	104	68	0	2183
sphinxbase-0.3	38	19	6	290	13	9	16	104	51	25	0	358
optipng-0.6	37	23	4	443	14	11	16	172	38	27	0	610
barcode-0.98	11	4	0	85	6	4	0	28	10	4	0	93
make-3.81	24	9	4	317	19	11	2	127	29	13	0	374
openssl-1.19	170	48	11	4429	58	35	24	1859	134	57	0	4827
openssh-4.3	51	16	10	851	18	12	14	314	76	26	0	928
Total	474	194	52	8543	170	115	131	3317	479	244	0	9674

图 4 是 DTSC_RSTV 检测出的一个空指针引用缺陷,而 DTSC_STVL 与 Klocwork9 均漏报了该缺陷.函数 Barcode_128_make_array 在 321 行调用了库函数 strlen, strlen 的 NPDPreSummary 要求其参数不能为空指针,而 bc -> ascii 的指向属性为 UNKNOWN,因此将(* bc).ascii(等同于 bc -> ascii)添加到所在函数 Barcode_128_make_array 的 NPDPreSummary 中.函数 Barcode_128_encode 在 439 行调用了函数 Barcode_128_make_array,因为与 bc -> ascii 具有别名的 text 在 434 行进行了非空判断,导致在 434 行 bc -> ascii 的指向属性为 NON,即可能造成调用函数 Barcode_128_make_array 时可能发生空指针引用.

```
文件:barcode \ code128.c
314 行的被调函数:
static int * Barcode_128_make_array(struct Barcode_Item * bc, * )

321:len = 2 * strlen(bc -> ascii) + 5;

414 行的主调函数:
int Barcode_128_encode(struct Barcode_Item * bc)
433:text = bc -> ascii;
434:if(! text){
        ……;
    }
439:codes = Barcode_128_make_array(bc,&len);
```

图 4　本文方法检测出的空指针引用缺陷

从三个工具的测试耗时可知,Klocwork 测试效率较高,DTSC_STVL 与 DTSC_RSTVL 测试效率略低.DTSC 主要是数据流分析阶段耗时较多,但对多重循环因为多次迭代会造成时间指数级增加,导致分析时间剧增.例如 uucp-1.07 工程中有几个函数中具有多级的嵌套循环,导致对 uucp-1.07 的检测效率较低.

通过测试对比结果可知,在缺陷检测领域,效率与精度、误报率与漏报率是相互制约的因素,相对于 DTSC_STVL,DTSC_RSTVL 以牺牲一定的效率而提高了分析的精度;相对于 Klocwork9,DTSC_RSTVL 以牺牲一定的效率及误报率的提升而极大的降低了漏报率.

6　结束语

本文提出了一种基于区域内存模型进行空指针引用缺陷检测的方法.RSTVL 能够描述每个程序点上的抽象存储以及可寻址表达式间的关联关系,用区域编号集合表示指针的指向,进而将对指针引用是否安全的判定转化为了是否指向了安全区域的判定.根据可寻址表达式与抽象语法树上节点的对应关系,能够识别出所有被引用的指针型可寻址表达式,根据该指针在其引用所在程序点的指向属性进行引用判定.对于指向属性不确定的指针,将与该指针具有别名的外部变量添加到所在函数的 NPDPreSummary 中,在函数调用点,基于 RSFVL 描述的抽象存储获得具体被引用的指针,进行是否为空的判定,以实现过程间空指针引用的检测.实验结果表明本文方法在保证一定检测精度与效率的前提下,能够极大的降低空指针引用缺陷的漏报.

参考文献

[1] Michael D, Graham Z, Samuel Z. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses[A]. Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation[C]. New York: ACM, 2010.13 – 24.

[2] Javelund K, Rosu G. Monitoring Java Programs with Java PathExplore[EB/OL]. http://ti. arc. nasa. gov/m/pub – archive/264h/0264%20(Havelund).pdf, 2001 – 06 – 15.

[3] Manevich R, Sridharan M, Adams S. PSE: Explainint program failure via psotmortem static analysis[A]. Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering[C]. New York: ACM, 2004.63 – 72.

[4] Ma XD, Wang J, Dong W. Computing must and may alias to detect null pointer dereference[A]. Leveraging Applications of Formal Methods, Verification and Validation[C]. Berlin Heidelberg: Springer, 2008, Vol_17.252 – 261.

[5] M Buss. Summary-Based Pointer Analysis Framework for Modular Bug Finding[D]. Columbia: Columbia University, 2008.

[6] Y Xie, A Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability[J]. ACM Transactions on Programming Languages and Systems, 2007, 29(3): 1 – 43.

[7] Madhavan Ravichandhran, Komondoor Raghavan. Null dereference verification via over-approximated weakest pre-conditions analysis[A]. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications[C]. New York: ACM, 2011.1033 – 1052.

[8] Dong Yukun, Xing Ying, Jin Dahai, Gong Yunzhan. An approach to fully recognizing addressable expression[A]. The 13th International Conference on Quality Software[C]. Piscataway, NJ: IEEE, 2013.149 – 152.

[9] Hackett B, Rugina R. Region-based shape analysis with tracked locations[A]. Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages[C]. USA: ACM, 2005.310 – 323.

[10] Zhao Yunshan, Wang Yawen, Gong Yunzhan, et al. STVL: Improve the precision of static defect detection with symbolic three-valued logic[A]. Proceedings of the 8th Asia-Pacific Software Engineering Conference[C]. NJ: IEEE, 2011.179 –

186.

[11] 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[J].电子学报,2011,39(2):296-303.
Wang Yawen,Gong Yunzhan,Xiao Qing,Yang Zhaohong.A method of variable range analysis based on abstract interpretation and its applications[J].Acta Electronica Sinica,2011,39(2):296-303.(in Chinese)

[12] 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[J].软件学报,2014,25(2):357-372.
Dong Yukun,Jin Dahai,Gong Yunzhan,Xing Ying.Static analysis of C programs via region-based memory model[J].Journal of Software,2014,25(2):357-372.(in Chinese)

[13] Zhao Yunshan,Gong Yunzhan,et al.Context-sensitive interprocedural defect detection based on a unified symbolic procedure summary model[A].Proceedings of the 11th International Conference on Quality Software[C].NJ:IEEE,2011.51-56.

作者简介

董玉坤 男,1981年出生于山东省梁山县.中国石油大学(华东)计算机与通信工程学院讲师,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 男,1962年生于山东省乳山市,北京邮电大学网络与交换技术国家重点实验室教授,博士生导师.研究方向为软件测试、可信计算等.

金大海 男,1974年出生于辽宁省沈阳市,北京邮电大学网络与交换技术国家重点实验室副教授.研究方向为软件测试、软件安全等.