基于值依赖分析的空指针解引用检测

马 森[1,2],赵 文[2,3],习翔宇[2,3],王栋伟[2,3]
(1.北京大学信息科学技术学院,北京 100871;2.北京大学软件工程国家工程研究中心,北京 100871;
3.北京大学信息科学技术学院软件研究所高可信软件技术教育部重点实验室,北京 100871)

摘 要: 本文提出了一种基于程序值依赖分析的、路径敏感的空指针解引用检测方法.该方法通过结合数据流分析中的到达定值分析、区间分析及指向分析创建了值依赖分析图,该图刻画了可能产生空指针语句到其解引用语句的值依赖关系.该图中的边采用守卫标注,即描述了相邻点之间的到达条件.为了降低误报率,本文同时提出了一种需求驱动的必然别名算法.由本文所述方法实现的工具展示了良好的实验效果,在10个SPEC2000项目中发现了70余个空指针解引用缺陷,误报率仅为6%左右.

关键词: 程序分析;静态缺陷检测;空指针解引用检测;需求驱动别名分析
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2015)04-0647-05
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2015.04.004

Null Pointer Dereference Detection Based on Value Dependences Analysis

MA Sen[1,2], ZHAO Wen[2,3], XI Xiang-yu[2,3], WANG Dong-wei[2,3]
(1. *School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*;
2. *National Engineering Research Center for Software Engineering, Peking University, Beijing* 100871, *China*;
3. *Key Laboratory of High Confidence Software Technologies (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*)

Abstract: This paper presents a context-sensitive and path-sensitive algorithm for detecting null pointer dereferences (NPD). Our algorithm tracks the flow of values from the points where a null pointer might be produced to dereference points via value dependence graph that captures def-use relations and combines interval analysis results. Edges in the graph are annotated with guards that describe branch conditions in the program. In the meantime, for reducing the false warnings we propose an innovative demand-driven must-alias algorithm using this graph. Our implemented tool detects more than 70 points which might produce null pointer dereferences in ten SPEC 2000 benchmarks while keeping the false positive rate around 6%, which is excellent experimental results.

Key words: program analysis; static error detection; null pointer dereference detection; demand-driven alias analysis

1 引言

空指针解引用已经成为程序最危险的缺陷之一[1].空指针解引用缺陷并不容易检测,原因有三:(1)尽管在大多数情况下,对于解引用会有非空保护存在,但保护可能与解引用的指针在不同函数中.分析器将会判断从指针赋值点到解引用点的所有可能路径中是否都包含保护.(2) 空指针保护中的指针变量与解引用的指针并不是同一指针.那么保护能否生效就取决于这两个指针是否为必然别名关系.但目前必然别名算法复杂度较高.(3)空指针解引用的前提条件是在可能为空指针与其解引用在同一定义使用链中.C语言中有以下两种情况可能会产生空指针定义点:(a) 动态内存分配,如malloc,calloc函数等.如果系统没有足够的堆内存将返回空指针.(b) 对指针变量赋空.为了确定上述节点与解引用在执行时是否可达,通用的方法采用定义使用链,但标准数据流分析并不是跨函数的且不是路径敏感的,这会导致相当数量的误报.

为了解决上述问题,更有效地检测空指针解引用缺陷,本文进行了如下研究:

(1)提出了一种新的跨函数分析算法,该算法通过数据流分析、指向分析以及区间分析辨别出变量的从

收稿日期:2014-03-03;修回日期:2014-09-16;责任编辑:李勇锋

“可能空指针定义产生点”到该指针的解引用点的值依赖关系,并创建了值依赖图(Value Dependence Graph, VDG)以表达整个程序,解决了第一及第三个问题.

(2)以值依赖分析为基础,提出了一种需求驱动的、域敏感的的必然别名分析算法,解决了上文中第二个问题.

(3)通过实验证明本文提出的方法能够进行大规模的程序检测并且有着较低的误报及漏报率.通过对10个 SPEC2000[2]的项目进行空指针解引用检测分析,共报出 79 个缺陷,其中 74 个为真实缺陷,误报率为 6.3%.

2 实例

本节结合 SPEC2000 工程中的代码片段,阐述空指针解引用分析的研究动机.

2.1 实例 1 必然别名关系

图 1 中的代码是一个出现空指针解引用缺陷的简化片段,取自于 SPEC2000 中的 parser 工程.

```
1. int *   func(int a,int b){                    //n0
2.   int *p, *q;
3.   if(a>0)                                     //n2
4.     p = malloc(sizeof(int));                  //n3
5.   else
6.     p = &b;                                   //n4
7.   …/*使用指针 p*/
8.   return p;                                   //n7
9.  }
10. void caller(int x){                          //n8
11.   int h = 1,                                 //n9
12.   int u = 3;                                 //n10
13.   int *s = func(h,u);                        //n11
14.   int *t;
15.   if(x>0)                                    //n12
16.     t = s;                                   //n13
17.   if(t! =null)                               //n14
18.     int c = *s;                              //n15
19.}
```

图 1 实例 1 在第 4 行内存分配失败的情况下,第 18 行构成空指针解引用.左侧注释是行数,右侧对应值依赖图节点标号

2.1.1 基于数据流分析的空指针解引用检测

采用任何一种在文献[3,4]提到的数据流分析方法.从第 4 行动态堆内存分配开始,前向遍历所有可能的路径,并跟踪记录指针 p 的状态,直到 p 在第 8 行将值返回给调用点,p 结束其生命周期.数据流分析将会记录指针 p 在第 4 行可能产生空指针,并在摘要中记录 p 指针返回的信息.空指针解引用在第 18 行出现,因为在第 17 行的空指针保护 t 与指针 s 并不是必然别名关系[5,6],不为别名的条件为 $x\leqslant 0$.总体来说,此方法通过遍历控制流图记录状态,跨函数采用摘要的方式,以及采用传统的必然别名的分析方式,效率较低,对于较大程序容易发生状态爆炸.

2.1.2 基于值依赖图的空指针解引用检测

我们的目标是仅通过值依赖关系推理检测出空指针解引用缺陷.图 1 程序对应的值依赖图如图 2 所示.实线表达了定义使用及在赋值情况下的值流关系.粗虚线表示函数调用.守卫标注到点之间连线上以表达到达条件.本例对应的值依赖图,存在一个从动态内存分配点 n_3 到其解引用点 n_{15} 切片,切片中存在两个守卫,$x>0$ 和 t ! = null,后者为空指针保护守卫,那么分析器将会判断该保护是否生效,即判断 t 与第 18 行解引用 s 是否为必然别名.该算法同样是在值依赖图上进行,通过计算得到当 $x\leqslant 0$ 时,s 和 p 不是别名关系.所以在该切片中存在空指针解引用.

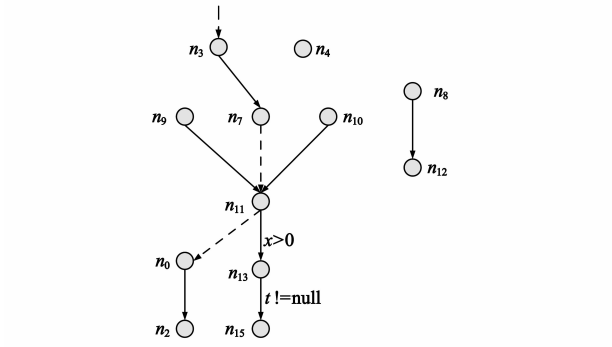


图2 实例1程序对应的初步值依赖图

2.2 实例 2 为空指针定义节点的确定

空指针解引用前提是解引用的对应指针是否可能被定义为空指针,实际中对于某个解引用的指针定义在不同的条件下可能存在多个,所以需要确认这些可能性的存在,代码举例如图 3 所示:

```
1.  void func() {
2.    int * hdSav = 0;
3.    callee(hdSav);
4.  }
5.  void callee(int * hdStore){
6.    int p = 0;
7.    hdTmp = 1;
8.    if(p == 0)
9.      hdStore = &hdTmp;
10.   int c = * hdStore;
11. }
```

图 3 实例 2 在第 10 行解引用的指针存在两个可能的定义点

图 3 代码片段中第 10 行的 * hdStore 可能定义点存在两个分别是第 9 行的 hdStore = &hdTmp;及第 2 行的 int * hdSav = 0.该实例对应的值依赖图如图 4 所示.

通过对该图进行检测发现 int * hdSav = 0 所在路径

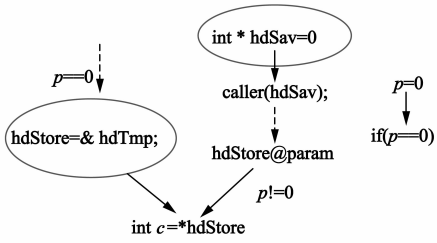


图4　实例2对应值依赖图

会出现空指针解引用. 但实际执行中 $p!=0$ 是永假式,int * hdSav = 0 所在分支是不可能到达其解引用点 int c = * hdStore,该分支实际并不存在. 因此,分支条件布尔值的判断,对于减少误报是很关键的.

3　系统概述

图 5 给出了分析系统的基本流程. 阴影部分是本文主要工作. 分析步骤如下所示:

(1)初步 VDG 构建:根据定值使用关系以及控制流图,构造初级值依赖图,边表达变量值的依赖关系,变量的依赖关系构成跨函数的连通图.

(2)值依赖图的守卫构建:通过控制图选择语句分析对守卫计算,并标注在 VDG 相应的边上.

(3)值依赖图的精化构建:根据区间分析对变量范围的保守计算移除真值为 false 的边.

(4)非守卫空指针解引用可达分析:找到可能发生空指针解引用的切片.

(5)守卫空指针解引用可达分析:判断保护指针是否有效.

(6)需求驱动的必然别名分析:对保护指针以及解引用指针进行必然别名分析.

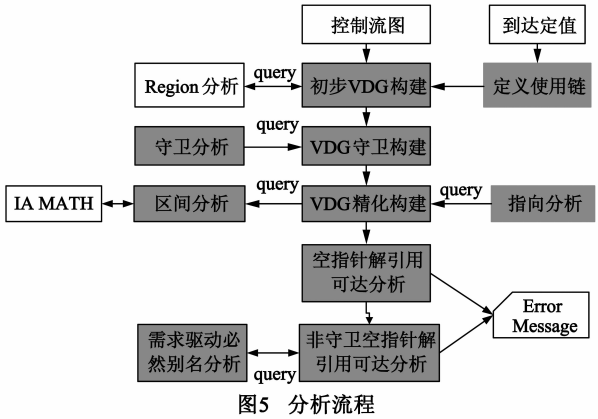


图5　分析流程

4　分析算法

本节将对图 5 中主要的阴影块的进行讨论,并给出基本思想以及相应算法.

本文提出的空指针解引用算法是建立在值依赖图基础上的,其采用数值的维度抽象程序,根据变量所代表的数值依赖关系表达程序语句之间的关系,针对空指针解引用检测中值的依赖关系主要包括由定值使用关系分析得出的流依赖关系、解引用依赖关系以及地址依赖关系. 通过这些关系抽象程序并以此进行分析,依赖的条件通过守卫关系表达. 本文的值依赖图与文献[7]的值依赖图的依赖的构建方式以及图中点与边含义均不同.

4.1　值依赖图的守卫构建

守卫,即构建值依赖图点之间的到达的条件. 图 6 给出通过遍历 CFG 计算守卫的算法:

```
/* x is the left value of n, n and m is start node and end node for guard computation, E is the edge set in the process of computation */
GUARDCOMP(LeftValue x, Node n, Node m, Edgeset E)
1   Let G be the cguard(n→m)
2     if( n and m are the same node)
3       then let G = true;
4     else
5       Let N be the successor nodes of n
6     for (each successor n1∈N)
7     if (the formula ¬DEF(n1, x) ∧ <n, n1> ∉ E is satisfiable)
8       then compute guard
9   C1 = COND(n, n1) ∧ GUARDCOMP(x, n1, m, E ∪ {<n, n1>}))
10      else ci = false;
11      Let G be the disjunction of Ci, G = ∪ni∈N Ci;
12      return G;

/* ni is the one of succeed nodes for n in VDG */
COND(Node n, Node ni)
1   if(n is switch of the form e(…, ci:ni, …))
2   then Let C be the condition (e = ci)n, under C and n reaches ni;
3     return C;
4   else
5     return true;
```

图 6　守卫计算方法,*define*(*n*, *x*)表达了节点 *n* 中 *x* 是作为左值出现,pdom 是后支配缩写

4.2　精化值依赖图的构建

我们采用标准的区间分析[8]对变量的范围进行计算,记录在变量的所在的节点的定义点中. 然后计算守卫的真值,对于永假式去除掉相应的边,永真式修改相应的条件,并重新计算边对应的守卫条件.

4.3　非守卫可达性检测

本阶段需要找到可能为空指针到其解引用的切片. 确定可能为空的指针能否到达解引用节点. 基本思想为:

1. 找到一组可能造成空指针的语句作为始点,称之为 NPD_{src},有以下两种情况:(1)动态分配内存语句;

(2)指针赋空的语句.

2.在某个 NPD_{src}下前向遍历值依赖图,找到一组解引用节点.该算法使用工作列表(Work List)对值依赖图进行遍历,对调用及返回进行相应的匹配处理.

3.最后进行非守卫可达性检测,有以下两种情况:

(1)如果没有找到解引用节点 NPD_{src},则不会发生空指针解引用,排除该 NPD_{src}.

(2)如果从 NPD_{src}能够到达的一组解引用,则采用后向遍历的方式,将这组解应用和 NPD_{src}的切片从值依赖图中截出并将其放入切片数组中,作为下个阶段守卫检测的输入.

4.4 守卫可达性检测

将上阶段得到的切片作为输入,分析切片中是否存在有效的指针"不为空"保护,这就需要进行必然别名关系判定以确定是否所有情况下该保护都能生效.空指针解应用分析如下述公式所示:

$$C_{\mathrm{NPD}} = \wedge_{k\in K} \vee_{i\in S} g_{ei} \tag{1}$$

g_{ei}即相邻节点间的守卫,对于每个从解引用点 N_i 到可能为空的指针初始点作为一条路径,对该路径中所有的 CGuard 进行析取得到该路径的 VDG-Guard,然后再将不同的路径的 VDG-Guard 进行合取运算来计算是否发生空指针解引用缺陷.如果结果为 true 那么就不会发生空指针解引用.CGuard 的构造在创建值依赖图中已经给出,但其布尔表达式指针的计算,取决于不同的缺陷类型.对于 g_{ei}来说,如具备如下形式 if(p! =0) 或者 if(p)都有可能进行了不为空保护,情况有如下几种:

(1)如果解引用的指针和保护指针,具有同样的定义节点集,那么为同一指针,g_{ei}设为 true,否则将该边的真值修改为 false.

(2)如果解引用的指针和保护节点的指针,不是同一指针,那么需要计算解引用指针是否为保护节点指针的必然别名,如果是那么 g_{ei}就设为 true.

(3)如果在切片中不含有保护指针,那么该切片存在发生空指针解引用缺陷.

4.5 需求驱动的必然别名的计算

在空指针解引用缺陷检测问题中,需要确定保护语句的指针 s 是否和解引用指针 p 为必然别名.该分析被转换为布尔可满足性问题,计算工作由 SAT-Solver[9] 完成.如果计算的结果为 true,则两个指针为必然别名,反之则非必然别名.由于值依赖图表达了由赋值语句产生的数值传播的关系,对于赋值关系导致的别名指针在同一连通图中,所以分析需要考虑指针之间守卫,基本思想如下:

1.首先计算 s 和 q,定义节点集 S 和 Q.

2.对于 s 的每个定义点 s_i 计算其到 q 的所有定义点集合 Q 的守卫,即对 s_i 到 q_j 的每条路径的守卫进行合取得到 $\vee_{j\in Q} VGuard(s_i, q_j)$.对于所有 s_i 的 $VGuard$ 结果进行析取,即为 $\wedge_{i\in S} \vee_{j\in Q} VGuard(s_i, q_j)$,结果如为 true 则 s 为 q 的必然别名,否则不是.

5 实验及结果分析

以本文上述所提及的方法实现的工具 COBOT 以 Crystal[10] 为基本分析框架实现了 C 语言程序的词法、语法分析、控制流分析、Region 分析以及到达定值分析等.而对于在区间分析中的区间计算,采用 IA_MATH 开源框架[11].此外,COBOT 使用开源布尔可满足性包 SAT4J[9],进行布尔约束公式计算.实验执行在 Linux 下,CPU 为 Intel i5,频率为 2.5GHz,内存为 4GB.

5.1 本文提出的分析方法

本文提到的采用区间分析精化以及需求驱动的必然别名分析算法能够显著提高分析精度,如表 1 所示.我们选择 10 个开源项目,前 5 个来自 SPEC 2000 后 5 个来自于 GNU."Before MA"列检测器在发现 NPD 切片后,如果找到了指针保护就认为不会发生 NPD,不去做必然别名判断."AFTER MA BEFORE IA"列检测器应用了必然别名算法但 VDG 没有采用区间分析方法重构.使用了本文提出的必然别名算法误报率从 40.1% 下降到 13.9%,这也是本文最大的贡献.而本文的区间分析精化模型的算法增强了 VDG 的路径敏感程度从而进一步降低了误报率.

表 1 本文算法对比实验结果

Programs			Before MA	After MA Before IA	After Both
Name	Size	Files	FP/Total	FP/Total	FP/Total
parser	22.3	96	8/34	0/21	0/21
mcf	6.5	43	17/31	1/15	1/15
vpr	6.0	17	6/12	2/8	0/6
gzip	9.0	34	3/6	2/5	1/4
gap	66	62	3/15	1/13	1/13
make	35.9	59	3/6	2/5	1/4
memfile	10.9	27	5/9	1/5	1/5
combine	23.4	74	3/11	1/9	0/8
ed	3.2	10	3/6	1/4	0/3
spell	0.7	1	2/2	1/1	0/0
Average	–	–	40.1%	13.9%	6.3%

5.2 缺陷报告结果

对该 10 个工程进行检测,COBOT 共检测得到 79 个 NPD 缺陷,其中 74 个为正确结果,工具误报率为 6.3%.经分析产生误报主要原因如下:(1)区间分析仅表达单变量的范围,不能表达变量之间的关系可能会导致守卫真值预估错误.(2)区间分析不能准确预估循环次数也可能会导致不能正确识别一些不能执行的路径.

5.3 与其他分析工具的对比

与 COBOT 对比的是两个经典的静态分析工具:商业工具 Klocwork insight 9 和开源静态分析工具 Saturn.从表 2 的结果来看,COBOT 相对于其他两种工具,表现出了更高的检测效率及更低的误报率.

表 2 COBOT、Klocwork 与 Saturn 检测结果对比

工程		COBOT		Klocwork		Saturn	
名称	大小	时间	误报	时间	误报	时间	误报
parser	66.0	7.2	0/21	40.1	0/13	80.3	1/21
mcf	22.3	13.5	1/15	34.4	1/10	52.4	2/14
vpr	35.9	12.6	0/6	38.2	0/3	62.4	0/5
gzip	9.0	1.2	1/4	8.1	1/6	20.4	1/4
gap	23.4	10.4	1/13	32.2	2/10	38.8	1/12
make	6.5	2.1	1/4	12.3	1/5	18.7	0/4
memfile	5.9	0.8	1/5	10.3	0/3	36.5	1/5
combine	3.2	0.7	0/8	8.4	0/7	22.1	1/7
ed	0.7	0.14	0/3	8.2	0/3	12.2	0/4
spell	10.9	4.7	0/0	15.6	0/0	34.5	0/0
平均		53.3	6.3%	207.8	8.3%	368.3	9.2%

COBOT 相对于其他两种工具具有更低的漏报率,如表 3 所示.对于 Klocwork 所采用的技术,我们不得而知,但从检测效果看,Klocwork 对于跨函数的必然别名是无法精确地计算,导致一些漏报.对于 Saturn 而言,其造成漏报稍高于 COBOT 的主要原因是缺少跨函数的路径敏感的计算.

表 3 漏报率近似预估及比较

Tools	FP/Reports	Estimated FN
Klocwork[12]	5 /60	55/80(31.2%)
Saturn[13]	7/76	69/80 (10%)
COBOT_NPD	5/79	74/80(7.5%)

6 结论

本文提出了一种新的空指针解引用分析方法.综合使用多种分析技术通过构造值依赖图来表达程序.在值依赖图的基础上,提出一种必然别名算法,以有效检测空指针解引用缺陷.相比其他分析工具,该方法适用于较大规模的程序并且效率更高,并且具有更低漏报及误报率.

参考文献

[1] The MITRE Corporation. Common Weakness Enumeration [OL]. http://cwe.mitre.org/,2014 - 07 - 31.

[2] J Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems[R]. Technical Report 1, SPEC Newsletter 1,1989.

[3] Das M,S Lerner,M Seigle. ESP: Path-sensitive program verification in polynomial time[J]. ACM Sigplan Notices,2002,37(5):57 - 68.

[4] Engler D,Chelf B,Chou A,et al. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions [M]. Berkeley: Usenix Assoc,2000. 1 - 16.

[5] Jagannathan,Suresh,et al. Single and loving it: must-alias analysis for higher-order languages[A]. 25th ACM Sigplan-Sigact Symposium on Principles of Programming Languages[C]. USA: ACM,1998. 329 - 341.

[6] Altucher, Rita Z, William Landi. An extended form of must alias analysis for dynamic allocation[A]. 22nd ACM Sigplan-Sigact Symposium Principles of Programming Languages[C]. USA: ACM,1995. 74 - 84.

[7] Jeannet B, Mine A. Apron: a library of numerical abstract domains for static analysis[A]. Lecture Notes in Computer Science[C]. Berlin: Springer-Verlag,2009. 661 - 667.

[8] Weise,Daniel,et al. Value dependence graphs: representation without taxation[A]. 21st ACM Sigplan-Sigact Symposium on Principles of Programming Languages[C]. USA: ACM,1994. 297 - 310.

[9] Daniel Le Berre, Anne Parrain. SAT4J: a satisfiability library for java[OL]. http://www.sat4j.org/,2011 - 01 - 01.

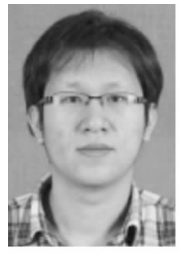
[10] Radu Rugina, Maksim Orlovich, Xin Zheng. Crystal: a program analysis system for C[OL]. http://www.cs.cornell.edu/projects/crystal,2006 - 11 - 01.

[11] Moore R E, Kearfott R B, Cloud M J. Introduction to Interval Analysis[M]. Siam,2009.

[12] Pär Emanuelsson, Ulf Nilsson. A comparative study of industrial static analysis tools[A]. Proceedings of the 3rd International Workshop on Systems Software Verification[C]. Australia,2008. 5 - 21.

[13] Xie Y C, Aiken A. Scalable error detection using boolean satisfiability[J]. ACM Sigplan Notices,2005,40(1):351 - 363.

作者简介



马　森　男,1980 年 10 月出生于天津,博士,主要研究领域为静态程序分析、软件工程.
E-mail: masen@pku.edu.cn

赵　文　男,1967 年 11 月出生于辽宁省大连市,博士,北京大学软件工程国家工程研究中心副研究员,主要研究领为软件工程、形式化分析技术.
E-mail: zhaowen@pku.edu.cn