软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(10):2983−3003 [doi: 10.13328/j.cnki.jos.006063]

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

基于污点分析的数组越界缺陷的静态检测方法*

高凤娟, 王 豫, 陈天骄, 司徒凌云, 王林章, 李宣东

(南京大学 计算机科学与技术系,江苏 南京 210023)
(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)
通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

摘 要: 随着移动计算、物联网、云计算、人工智能等领域的飞速发展,也涌现出了很多新的编程语言和编译器,但是C/C++语言依旧是最受欢迎的编程语言之一,而数组是C语言最重要的数据结构之一.当在程序中通过数组下标访问数组元素时,必须确保该下标在该数组的边界之内,否则就会导致数组越界.程序中的数组越界缺陷会使得程序在运行时导致系统崩溃,甚至使攻击者可以截取控制流以执行任意恶意代码.当前针对数组越界的静态检查方法无法达到高精度的分析,尤其是无法处理复杂约束和表达式,过多的误报额外增加了开发者的负担.因此,提出了一种基于污点分析的数组越界的静态检测方法.首先,提出流敏感、上下文敏感的按需指针分析方法,实现数组长度区间分析.然后,提出按需污点分析方法,实现数组下标和数组长度污染情况的计算.最后,定义数组越界缺陷判定规则,提出使用后向数据流分析方法,检测数组下标是否越界.在进行数组越界检测的过程中,为了处理程序中的复杂约束和表达式,在分析过程中将调用约束求解器来判断约束的可满足性.如果没有发现相应的语句,则报告数组越界缺陷警报.同时,实现了自动静态分析工具Carraybound,并通过实验展示了方法的有效性.

关键词: 数组越界;静态分析;缓冲区溢出;约束求解
中图法分类号: TP311

中文引用格式: 高凤娟,王豫,陈天骄,司徒凌云,王林章,李宣东.基于污点分析的数组越界缺陷的静态检测方法.软件学报, 2020,31(10):2983−3003. http://www.jos.org.cn/1000-9825/6063.htm

英文引用格式: Gao FJ, Wang Y, Chen TJ, Situ LY, Wang LZ, Li XD. Static checking of array index out of bounds defects in C programs based on taint analysis. Ruan Jian Xue Bao/Journal of Software, 2020,31(10):2983−3003 (in Chinese). http://www.jos.org.cn/1000-9825/6063.htm

Static Checking of Array Index out of Bounds Defects in C Programs Based on Taint Analysis

GAO Feng-Juan, WANG Yu, CHEN Tian-Jiao, SITU Ling-Yun, WANG Lin-Zhang, LI Xuan-Dong

(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)
(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: During the rapid development of mobile computing, IoT, cloud computing, artificial intelligence, etc, many new programming languages and compilers are emerging. Even so, C/C++ language is still one of the most popular languages. And array is one of the most important data structures of C language. It is necessary to check whether the index is within the boundary of the array when using it to access the element of an array in a program. Otherwise, array index out-of-bounds will happen unexpectedly. When there are array index out-of-bounds defects existing in programs, some serious errors may occur during execution, such as system crash. It is even worse that array index out-of-bounds defects open the doors for attackers to take control of the server and execute arbitrary malicious code by carefully constructing input and intercepting the control flow of the programs. Existing static methods for array boundary checking cannot

* 基金项目: 国家重点研发计划(2017YFA0700604); 南京大学优秀博士研究生创新能力提升计划B; 江苏省研究生科研与实践创新计划

Foundation item: National Key Research and Development Program of China (2017YFA0700604); Program B for Outstanding PhD Candidate of Nanjing University; Postgraduate Research & Practice Innovation Program of Jiangsu Province of China

本文由“系统软件前沿进展”专题特约编辑武延军研究员、陈海波教授、包云岗研究员、李玲研究员推荐.

收稿时间: 2020-01-02; 修改时间: 2020-04-04; 采用时间: 2020-05-09; jos 在线出版时间: 2020-06-10

achieve high accuracy and deal with complex constraints and expressions, which lead to too many false positives. And it will increase the burden of developers. In this study, a static checking method is proposed based on taint analysis. First, a flow-sensitive, context-sensitive, and on-demand pointer analysis is proposed to analyze the range of array length. Then, an on-demand taint analysis is performed for all array indices and array length expressions. Finally, the rules are defined for checking array index out of bounds defects and the checking is realized based on backward data flow analysis. During the analysis, in order to deal with complex constraints and expressions, it is proposed to check the satisfiability of the conditions by invoking the constraint solver. If none statement for avoiding array index out-of-bound is found in the program, an array index out-of-bound warning will be reported. An automatic static analysis tool, Carray bound have been implemented, and the experimental results show that Carraybound can work effectively and efficiently.

Key words: array index out-of-bounds; static analysis; buffer overflow; constraint solving

随着移动计算、物联网、云计算、人工智能、开源软件、RISC-V 开源指令集等领域的飞速发展,相关的软/硬件都迎来了新的发展机遇和挑战.为了适应软/硬件的发展,当前也涌现出了很多新的编程语言和编译器.但是,C 语言作为高效率、面向过程、抽象化的通用程序设计语言,仍然广泛应用于系统软件的开发.系统软件中如果存在漏洞,就可能会被恶意利用,将会严重影响人们的生产生活,甚至威胁到生命财产安全.软件安全已成为软件企业不能回避的挑战.

C 语言被广泛应用于底层软件生态系统的开发,这是因为 C 语言程序具有更高的运行效率.其中,数组是 C 语言最重要的数据结构之一.当一个数组在程序中被使用时,访问该数组的下标索引必须在一定范围之内,即不小于 0 并且小于数组的大小,否则会造成数组下标越界.数组越界的缺陷在很多编译后的系统中都存在,而且通过实验,我们发现,像 gcc、clang 这样主流的 C 语言编译器,在编译过程中不会对数组下标取值范围的合法性加以严格检查.数组越界有两种模式,读越界和写越界.读越界会导致读取到随机的值,继而使用该值会导致未定义行为.相比之下,写越界会造成更加严重的后果,不仅会造成未定义行为,甚至会导致控制流截取,使得攻击者可执行任意恶意代码[1,2].如图 1 所示,根据 CVE 历史统计可以发现,占据前三的漏洞类型为拒绝服务、代码执行和溢出[3].而数组越界缺陷常常会导致拒绝服务、代码执行和溢出.比如 Adobe 阅读器 2017 之前版本中存在的远程代码执行和拒绝服务漏洞就是由于外部输入作为数组下标从而可能导致写越界引起的(CVE-2017-16391、CVE-2017-16410).同时,研究表明[4−6],31%的缓冲区溢出是由数组越界造成的.因此,可见数组越界缺陷仍然是严重威胁系统软件安全的重要缺陷类型.本文主要针对在给定 C 程序源码时,检测数组下标越界和由循环导致的数组下标越界问题.为了提高系统软件的安全性,程序必须对由外部输入控制的数组下标进行边界检查.但是,开发者可能会遗漏边界检查或者没有进行正确的边界检查,使得程序处于可被攻击利用的不利状态.

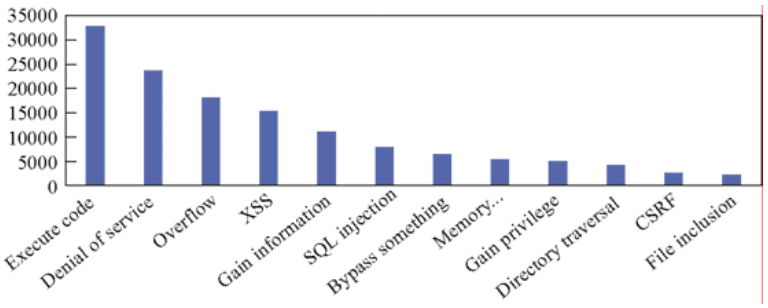


Fig.1 Statistics of the number of common types of CVE vulnerabilities

图 1 CVE 常见类型漏洞的数目统计

目前已有一些研究工作提出使用静态分析方法和动态测试方法来进行数组越界检查.由于动态方法总是依赖于测试用例的完整性,从而导致这些方法无法达到足够高的程序覆盖率.静态分析方法通过扫描分析程序源代码以检查程序中的缺陷.当前的静态分析方法出于效率的考虑,未对程序进行高精度的分析,未处理循环导致的数组越界,并且多是基于规则的匹配方法,无法处理复杂约束、复杂表达式等情况,导致一些已有的数组越界静态检测方法从而存在大量的误报和漏报现象.

因此,为了进行高精度的、高效的静态分析来检测数组越界缺陷,一方面,我们提出使用按需污点分析计算数组下标和数组长度的污点值,当数组长度污染时,即使在数组下标非污染的情况下仍然可能导致数组越界(比如图 2 第 11 行所示的数组访问语句),而在数组下标污染时,则需要进行高精度的数据流分析以检测是否会导致数组越界;另一方面,我们提出在静态分析的过程中使用优化后的约束求解器来处理程序中与数组访问相关的复杂约束和复杂表达式,从而有效地提高检测方法的准确性.对于数组下标越界问题,我们关注从程序入口到数组访问语句之间,数组下标越界条件的可满足性,通过将约束求解引入到数据流分析过程中,能够更加精确地检查数组越界问题.

在本文中,我们提出了一个静态分析框架 Carraybound,该框架基于静态的污点分析、数据流分析以及约束求解检查程序中是否存在潜在的数组越界缺陷.除此之外,Carraybound 还提供待增加的数组边界检查条件,可以帮助程序员更加方便、快速地定位、确认和修复我们报告的数组越界警报.实验结果表明 Carraybound 可以快速、有效地报告出程序中的数组越界缺陷,并通过与已有静态分析工具 Cppcheck、Checkmarx 和 HP Fortify 进行对比,展示了 Carraybound 在发现数组越界缺陷上的优越性.

本文的主要贡献包括:

- 提出流敏感、上下文敏感的按需指针分析方法,实现数组长度区间分析;提出按需污点分析方法,实现数组下标和数组长度污染情况的计算.
- 提出基于污点分析的数组越界缺陷的检测方法,定义了数组越界缺陷判定规则,然后依据判定规则,使用后向数据流分析检测数组越界,并将约束求解引入到数据流分析过程中以检查数组越界缺陷,同时通过优化尽量减少对约束求解器的调用,以提高分析效率.
- 实现了一个静态分析工具 Carraybound,检测 C 程序中的数组越界缺陷(包括由循环导致的数组越界缺陷),并通过实验展示了工具的有效性.

本文第 1 节介绍我们工作的背景知识.第 2 节介绍所提出的基于污点分析的数组越界的静态检测方法.第 3 节介绍实现工具 Carraybound,并通过实验展示该工具的有效性和效率,同时讨论该工具的不足之处.第 4 节介绍相关工作.第 5 节进行总结,并提出未来工作展望.

1 背景知识

1.1 数组越界

数组是在内存中连续存储的具有相同类型的一组数据的集合.C 语言中数组分为静态数组和动态数组.静态数组在内存中位于栈区,其长度为常量,是定义时就在栈上分配了固定大小,运行时,这个大小不能改变,例如 *char a*[7].对静态数组的写越界访问将会导致栈上的缓冲区溢出.动态数组在内存中位于堆区,其长度可以是变量,亦即可以在程序运行时在堆上动态分配大小,例如 *int *a=(int*)malloc(sizeof (int)*10)*.对动态数组的写越界访问将会导致堆上的缓冲区溢出.

当一个数组在程序中被使用时,访问该数组的下标索引必须在一定范围之内,即不小于 0 并且小于数组的大小,否则会造成数组下标越界[8].如图 2 中第 11 行所示的数组,由于数组 *arr* 的长度 *m* 来自于外部输入,因此常量数组下标也可能会导致数组越界访问.

如图 3 所示,我们将 C 语言程序中的越界访问问题归结为以下两类.

① 数组下标越界:包括读越界和写越界.例如 *char c=a*[5]是读越界而 *a*[5]=0 属于数组的写越界.其中,数组下标访问写越界会导致缓冲区溢出,即图 3 所示的交集部分.数组下标越界还包括循环导致的数组下标越界,比如 *char a*[5];for (*int i*=0;*i*<6;*i*++) {*a*[*i*]=0;}.

② 缓冲区溢出:包括 API 调用和数组下标访问写越界导致的缓冲区溢出.比如 *strcpy*(*dest*,*src*)和 *a*[5]=0.

本文方法目前侧重于数组下标越界(包含循环导致的数组下标越界)问题.对于由 API 调用导致的缓冲区溢出问题,我们的另一项缓冲区溢出静态警报确认的工作重点关注了该类问题[5].

通常情况下,程序员可以通过特定的方式限制数组下标的范围,以避免数组越界的发生.常见的 3 种方式如

下所示.

① idx=idx % size;

② if (idx>=size||idx<0)…

③ assert (idx>=0 && idx<size);

程序中也可能存在一些复杂的约束和表达式实现了对数组下标的范围限制,比如按位操作、包含多个运算符的线性运算,甚至是非线性约束.这些情况会加大分析难度,导致传统方法无法精准地检查出程序中的数组越界缺陷.

```
1.  typedef unsigned int UINT32;
2.  typedef struct {
3.      UINT32 noisy[12];
4.      UINT32 arr[15];
5.  }myStruct;
6.  myStruct s;
7.
8.  void f(UINT32 m,char* arr) {/*main->f: m<12*/
9.      UINT32 tmp[3]={1,2,3};
10.     UINT32 n=m;
11.     s.noisy[n]=0;
12.     arr[2]=0;
13.     for(UINT32 i=0;i<n;i++) {
14.         if(i>=15)
15.             break;
16.         s.arr[i]=tmp[i];
17.     }
18. }
19. int main(int argc, char** argv) {
20.     UINT32 j, k;
21.     scanf("%d %d",&j,&k);
22.     char* p = malloc(j);
23.     char* q = malloc(k);
24.     char** t;
25.     if (j>k)
26.         t=&p;
27.     else
28.         t=&q;
29.     if(argc+2<15){
30.         f(argc-1,*t); /*argc-1<12*/
31.     }
32.     return 0;
33. }
```

Fig.2 Example code of test.c

图 2 test.c 代码示例

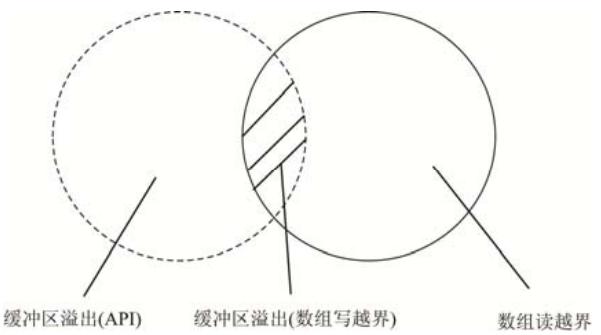


Fig.3 Buffer overflow vs. array index out of bound

图 3 缓冲区溢出和数组下标越界关系图

1.2 污点分析

污点分析是检测程序漏洞的常用技术[7,9,10].如果攻击者向程序输入一些恶意数据,而程序没有进行恰当的防护,则可能会导致系统处于不安全的状态.这些受外部输入影响的数据在污染分析中标记为污染.外部输入包括用户或文件的输入、主函数的参数.污点分析尝试识别程序中那些会被用户输入污染的变量,并最终追溯到可能会导致程序缺陷的语句.如果在该语句之前,未经检查就直接使用被污染的数据,则视为一个程序缺陷.污点分析分为静态污点分析和动态污点分析.其中,动态污点分析需要执行程序,无法保证对源码的覆盖率.静态污点分析主要依赖于程序抽象语法树和控制流图进行数据流分析,不需要实际执行程序.因此静态污点分析可以实现比动态污点分析更高的源代码覆盖率.但静态污点分析可能因缺乏运行时信息而产生误报和漏报.

1.3 数据流分析

数据流分析通常用于静态代码分析,是一种基于控制流图来收集数据流信息的技术.对程序进行数据流分析的一种简单方法是为控制流图的每个节点建立数据流方程,并通过在每个节点上重复计算输入的输出来反复迭代,直到整个系统到达不动点[11,12].前向数据流分析和后向数据流分析是数据流分析的两种不同方法.前向

数据流分析沿着正常的执行路径,从入口节点开始并在目标节点处结束.一个基本块的出口状态是该基本块中的语句对该基本块的入口状态作用的结果,一个基本块的入口状态则是其所有前驱基本块的出口状态的组合.与之相对地,后向数据流分析与控制流图中的有向边方向相反,从目标节点开始并在入口节点处结束.一个基本块的入口状态是该基本块中的语句对该基本块的出口状态作用的结果,一个基本块的出口状态则是其所有后继基本块的入口状态的组合.

1.4 指针分析

数组越界缺陷的根源实际上是由于指针对内存的越界访问引起的.数组名代表了一个指向数组首元素的指针,通过数组下标访问数组元素(即 $p[i]$),实际上与指针从数组首元素移动到特定元素进行访问(即 $*(p+i)$)是等价的.为了计算数组名实际指向的内存区域,就需要进行指针分析.

指针分析是一类特殊的数据流问题,是指通过程序分析方法计算指向相同内存区域的指针表达式的集合.指针分析有几个重要的精度衡量属性,如流敏感性、上下文敏感性等.流敏感的指针分析会区分指针变量在不同控制流位置的指向信息.上下文敏感性用来反映过程间分析在分析某个过程时,是否会区分不同调用点的上下文对过程输入带来的不同,从而影响过程的输出.

1.5 SMT可满足性问题

可满足性模理论(satisfiability modulo theories,简称 SMT)求解器是用来判定一阶逻辑公式可满足性的程序,是许多形式化方法的验证引擎.SMT 求解技术在有界模型检验、基于符号执行的程序分析、线性规划和调度、测试用例生成以及电路设计和验证等领域有着非常广泛的应用[13].

Z3[14,15]是一个由微软研究院开发的高性能 SMT 求解器,是目前为止综合求解能力最强的 SMT 求解器.因此,本文中引入 Z3 约束求解器以帮助 Carraybound 能够更精确地检测数组越界缺陷.

2 Carraybound:基于污点分析的数组越界缺陷的静态检测方法

2.1 方法框架

本文的数组越界缺陷的检查方法主要基于静态污点分析技术和数据流分析技术,这些方法主要基于程序的抽象语法树、函数调用图和控制流程图进行分析.本文方法的框架如图 4 所示,首先根据程序的源码生成抽象语法树(AST),再根据 AST 构建函数调用图(call graph)和控制流程图(CFG).然后,基于 CFG、AST 和函数调用图,执行污点分析以确定可能被污染的数组下标.然后,定位所有包含这些被污染的数组下标的数组表达式的语句,将数组下标符号化,通过边界分析得到每个数组下标的边界信息.接下来,执行后向数据流分析,并提供简单匹配和约束求解两种方法,检测程序中是否存在相应的表达式保证了数组下标的边界条件.如果不存在这些表达式,则报告数组越界缺陷警报.

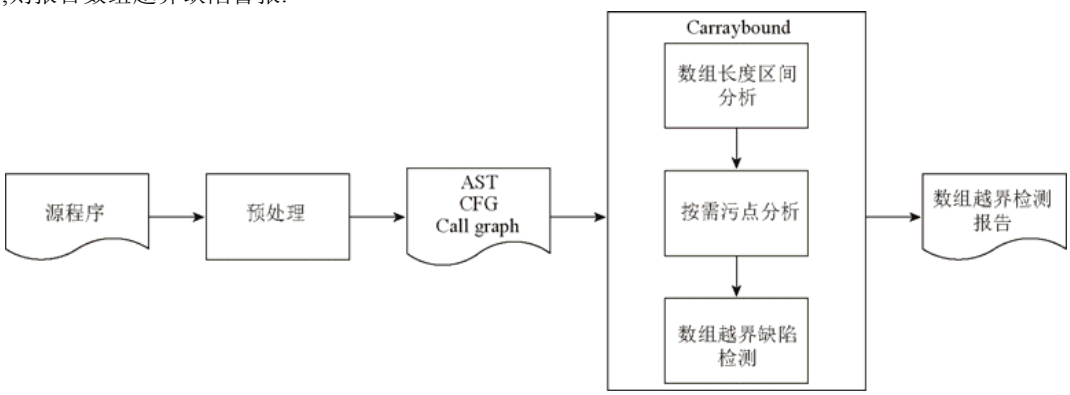


Fig.4 The overall framework of our method

图 4 方法框架

2.2 数组长度区间分析

首先,定位数组下标访问语句,然后,通过后向数据流分析得到所关注数组名的别名,进而定位到数组声明语句,每一个数组声明语句对应一个相应的数组长度.由于一个数组可能对应多个数组声明语句,因此,这一过程将分析得到数组的长度区间,具体步骤如下.

数组长度区间分析.对于数组访问语句 *arr*[*idx*],在 AST 上判断是否存有数组声明大小,如果无法直接获取数组声明大小,则进行指针分析,得到 *arr* 实际是某个或者某几个静态数组或动态数组的别名.

本文设计了一种流敏感、上下文敏感的按需指针分析.按需是指本文只对所关注的数组名(即通过被污染数组下标访问数组元素的语句中的数组名)进行指针分析,目的是计算出数组名实际对应的静态数组或动态数组表达式.对于每个函数 *f*,首先,统计每个基本块 *block* 中包含被污染的数组下标的数组表达式,然后从最底层包含关注数组表达式的位置开始,向上进行一个后向数据流分析,后向数据流分析中,使用 *OutState* 表示一个基本块在该基本块出口时的状态,是其所有后继基本块的 *InState* 的并集,如公式(1)所示,此处代表在该基本块每个所关注的数组名对应的待分析的指针集合;*InState* 表示一个基本块在该基本块入口时的状态,如公式(2)所示,此处是在其 *OutState* 的基础上,该基本块的赋值语句根据下文的指针处理规则对每个所关注的数组名对应的待分析的指针集合进行杀死(kill)和生成(gen)的结果.

$$OutState(block)=\bigcup_{s\in succ(block)} InState(s) \tag{1}$$

$$InState(block)=Gen(block)\cup(OutState(block)-Kill(block)) \tag{2}$$

指针分析时,*InState* 和 *OutState* 维护的是在某一个基本块中每个所关注的数组名 *arr* 对应的待分析的指针集合 $AliasSet=\{p_1,p_2,\ldots,p_n\}$.初始时,$AliasSet=\{arr\}$.对于 *AliasSet* 中每一个指针 *p*,首先查询 AST 是否可以直接获取静态数组声明的大小,如果可以,则将 *AliasSet* 中该元素 *p* 标记为**终止点**,停止对 *p* 进行指针分析;否则,继续对 *p* 进行指针分析,针对以下赋值语句,后向数组流分析的具体指针处理规则为:

- *p*=*malloc*(*size*)将 *AliasSet* 中的 *p* 替换为 *malloc*(*size*),并将 *AliasSet* 中该元素 *malloc*(*size*)标记为终止点,停止对 *malloc*(*size*)进行指针分析.
- *p*=*&a* 将 *AliasSet* 中的 *p* 替换为*&a*.
- *p*=*q* 将 *AliasSet* 中的 *p* 替换为 *q*,后续数据流分析继续分析 *q* 的别名.
- *p*=**q* 将 *AliasSet* 中的 *p* 替换为**q*.
- **p*=*q* 将 *AliasSet* 中的 *p* 替换为*&q*.
- *p*=*g*(…)将进入函数 *g* 中,从函数返回语句开始进行指针分析.

如果 *AliasSet* 中的元素包含*&*符号,形如*&p*,则在后续数据流分析中分析对 *p* 的赋值表达式,如果替换的新指针为 *q*,则 *AliasSet* 中替换*&p* 为*&q*,如果新指针为**q*,则 *AliasSet* 中*&p* 替换为*&*(**q*),即替换为 *q*,以此类推.规则中将 *AliasSet* 中的 *p* 替换为 *q*,对应着对 *AliasSet* 中杀死(*Kill*)*p* 和生成(*Gen*)*q*.

如果在函数 *f* 中没有定位到数组名对应的数组声明语句,则用相同方法分析该函数的所有调用者.以此类推,直到定位到数组声明语句.由于不同的上下文、不同的分支将导致一个数组名可能存在多个数组声明语句作为别名,其中,每一个数组声明语句将对应一个数组长度,因此,对于一个数组名 *arr*,通过指针分析将得到它的一组数组长度 $\{s_0,s_1,\ldots,s_n\}$.为了支持流敏感和上下文敏感,将在数据流分析过程中额外记录:(1) 每个基本块 *block* 的后继节点集合(计算方法参照公式(1)),记为 *succs*(*block*);(2) 在分析函数 *f* 时,过程间后向数据流分析的函数调用链,记为 *succs*(*f*).对于数组名 *arr*,当在指针分析时,在函数 *f* 的基本块 *bb* 中定位到一个 *arr* 的数组声明语句作为别名时(即 *AliasSet* 中的终止点之一记为 p_0),设 p_0 对应的数组长度为 s_0,则将记录 s_0 对应的有效函数和基本块信息,有效函数信息为过程间后向数据流分析的函数调用链,即 $ValidFuncs(s_0)=succs(f)$,有效基本块信息为所在基本块的后继节点集合,即 $ValidBBs(s_0)=succs(bb)$.最终,每个数组名 *arr*,记录一组数组长度 $\{s_0,s_1,\ldots,s_n\}$,且其中每一个数组长度 s_i 记录该值的作用域,即有效函数 $ValidFuncs(s_0)=succs(f)$和有效基本块 $ValidBBs(s_0)=succs(bb)$,并由此推导出每个函数 *f* 的每个基本块 *bb* 中数组名 *arr* 对应的数组长度的集合 $size(arr,f,bb)=\{s_i,s_j,\ldots,s_k\}$.

基于每个函数 f 的基本块 bb 中数组名 arr 的数组长度的集合 $size(arr,f,bb)=\{s_i,s_j,\ldots,s_k\}$,取其中的最大值记为数组长度的上界 up,取其中的最小值记为数组长度的下界 low,则数组长度的区间为$[low,up]$,如果数组长度为外部输入的变量,无法确定上下界,则保留长度集合.

示例解析.如图 2 代码示例,遍历可知,在函数 f 中,共有 4 个数组表达式的使用位置,即 11 行的 $s.noisy[n]$、12 行的 $arr[2]$和 16 行的 $s.arr[i]$以及 $tmp[i].s.noisy[n]$和 $s.arr[i]$均为结构体成员数组,可以直接定位其数组声明位置为第 3 行、第 4 行,继而可知数组长度分别为 12 和 15.$tmp[i]$的数组声明位置为第 9 行,数组长度为 3.$arr[2]$来自于函数参数,通过指针分析可知,动态数组 p 和 q 为 arr 的别名,因此 arr 数组长度集合为$\{j,k\}$.

2.3 按需污点分析

按需污点分析是指本文中的污点分析只针对程序中的数组下标和数组长度进行静态污点分析,包括过程内和过程间污点分析.本文将外部输入(包括用户或文件的输入以及主函数的参数)作为污点源.通过污点传播,可以得到程序中每个关注变量 v 的污点值 $T(v)$,可以是被污染(tainted)或者无污染(untainted),即

$$T(v)\in\{tainted,untainted\}.$$

污点值的 $tainted$ 可以对应布尔值 1,$untainted$ 可以对应布尔值 0,因此可以使用逻辑运算符“或”(记为$\vee$)来计算污点值的和,即只要有一个子表达式的污点值为 $tainted$,则整个表达式的污点值为 $tainted$.

2.3.1 污点传播规则

对于污点分析中遇到的每条语句,将按照如下污点传播规则计算该语句的污点值.

常量.每个常量 c 是非污染的.比如字符串常量、整型常量和浮点型常量等.

$$T(c)=untainted.$$

类型转换.类型转换后的表达式 $CastExpr(e)$的污点值与原来类型的表达式 e 的污点值一致.

$$T(CastExpr(e))=T(e).$$

数组下标表达式. 将被当作一个整体,数组的某一个元素被污染,则整个数组为污染的.结构体同理.

$$T(arr[i])=T(arr),T(expr.elem)=T(expr).$$

一元运算表达式.$op\ expr$ 的污点值等于其中表达式 $expr$ 的污点值.

$$T(op\ expr)=T(expr).$$

二元运算表达式.$expr1\ op\ expr2$ 的污点值等于子表达式 $expr1$ 和 $expr2$ 的污点值之和.

$$T(expr1\ op\ expr2)=T(expr1)\vee T(expr2).$$

三元运算表达式. $expr1?expr2:expr3$ 的污点值等于子表达式 $expr2$ 和 $expr3$ 的污点值之和.

赋值表达式.赋值语句 $expr1=expr2$ 将把右侧表达式的污点值传递给左侧变量.

$$T(expr1)=T(expr2).$$

条件语句.if c then $expr1$ else $expr2$ 将把条件语句中条件表达式 c 的污点值传递给基本块中的赋值语句中的左值.循环语句同理,同时,循环变量的污点值等于循环上界的污点值.

函数调用语句.设函数 f 有 n 个参数,对 f 的调用语句将把第 i 个实参 p_i 的污点值传递给第 i 个形参 a_i.

$$\forall i\in[0,n),T(a_i)=T(p_i).$$

同时,函数调用语句将把被调用函数返回值的污点状态传递给调用者被赋值的变量.

函数返回语句.如果返回的是变量,则函数返回值的污点值等于该变量的污点值;如果返回的是常量(包括返回值为空),则函数返回值的污点值为 $untainted$.

2.3.2 按需过程内污点分析

污点分析前,首先统计程序中所有与数组下标和数组长度相关的函数,以及直到入口函数的所有调用者函数,构成数组相关函数集合 FS.过程内的污点分析是对 FS 中的每个函数进行前向数据流分析.对于函数中的每个基本块,使用 $InState$ 表示一个基本块在该基本块入口时的污点状态,是其所有前驱基本块的 $OutState$ 的并集,代表在该基本块执行前所有表达式的污点状态;$OutState$ 表示一个基本块在该基本块出口时的污点状态,是在该基本块的 $InState$ 的基础上,由该基本块中的语句按照上一节中的污点传播规则更新表达式的污点状态的结

果,其中,*Kill* 将消除表达式的污点状态,*Gen* 将生成表达式的污点状态:

$$InState(block)=\bigcup_{p\in pred(block)}OutState(p) \tag{3}$$

$$OutState(block)=Gen(block)\cup(InState(block)-Kill(block)) \tag{4}$$

对于包含循环的函数,将迭代计算每个基本块的 *InState* 和 *OutState*,直到该基本块的 *InState* 和 *OutState* 的状态不再改变.函数的污点状态与函数出口基本块的 *OutState* 相同.这样,就可以得到函数内所有表达式与相应函数形参之间的污点关系,即为该函数 *f* 的污点摘要 *TS*(*f*).

对于每个函数 *f*,其形参列表为 $A=\{a_1,a_2,\ldots\}$,函数中每个变量 *v* 的污点状态记为 *T*(*v*),其值可能是被污染、无污染或依赖于函数的形参,即

$$T(v)=\begin{cases} tainted \\ untainted \\ \bigcup_{a\in A'}T(a), A'=\{a_i \mid RelyOn(a_i,v), a_i\in A\} \end{cases} \tag{5}$$

2.3.3 按需过程间污点分析

首先将入口函数的参数标记为污染的.然后从入口函数开始,对 *FS* 的所有函数依据调用图上的拓扑顺序分析,通过函数调用语句,在调用点将实参的污点值传递给函数形参,计算得到 *FS* 中每个函数形参的污点值.如果有多个函数调用同一函数,则被调用函数的参数污点值是其所有调用者实参的污点值之和.

对于函数 *f* 的第 *i* 个形参 a_i^f,其污点值是 *f* 所有调用者 $Caller_1,\ldots,Caller_j$ 相应实参 p_i 的污点值之和,即

$$T\left(a_i^f\right)=\bigcup_{k=1}^{j}T\left(p_i^{caller_k}\right).$$

当需要查询程序中表达式的污点状态时,如果发现可以直接得到污点值,则直接返回(*tainted*/*untainted*);否则,表示该表达式的污点状态依赖于函数的形参,此时只需要将所依赖的函数形参的污点值 *T*(*a*)代入到公式(5)第 3 个赋值中即可得到原表达式的污点值.

2.3.4 示例解析

如图 2 所示的代码片段,其中,函数 *f* 的控制流程图如图 5 所示,图 5 中冒号后的数字是入口语句的行号.分析可知,4 个数组表达式的数组下标为 *n*、2 和 *i*.通过对函数 *f* 进行污点分析,可以知道 *f* 中的变量 *n* 和 *i* 的污点值与 *f* 的参数 *m* 一致.然后对 *main* 函数进行污点分析,*argc* 和 *argv* 由外部输入,因此是污染的.由于 *main* 函数通过参数 *argc*−1 调用函数 *f*,导致函数 *f* 的形参 *m* 为污染的.进而,*f* 中的变量 *n* 和 *i* 也是污染的.

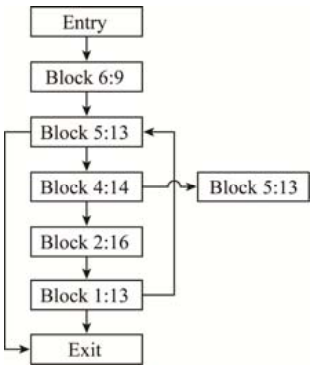


Fig.5 CFG of function *f* in test.c

图 5 示例 test.c 函数 *f* 的控制流程图

2.4 数组越界缺陷检测

对于每一条数组访问语句 *arr*[*idx*],数组 *arr* 对应的数组长度列表为 $\{len_0,len_1,\ldots,len_n\}$,其数组越界缺陷的检测结果记为 $W(arr[idx])$,$W(arr[idx])\equiv T$ 表示该数组访问语句会导致数组越界,$W(arr[idx])\equiv F$ 表示该数组访问语

句不会导致数组越界.

判定规则 1.对于数组访问语句 *arr*[*idx*],如果数组下标 *idx* 为非污染的,数组长度列表中任意一个长度为污染的,则该数组访问语句会导致数组越界.

$$\begin{cases} T(idx)\equiv untainted \\ \exists i\in[0,n], T(s_i)\equiv tainted \end{cases} \Rightarrow W\left(arr[idx]\right)\equiv T \tag{6}$$

判定规则 2.对于数组访问语句 *arr*[*idx*],如果数组下标 *idx* 为非污染的,数组长度列表中每一个长度都为非污染的,则如果数组下标小于数组长度列表中的每一个长度,则该数组访问语句不会导致数组越界;否则,该数组访问语句会导致数组越界.

$$\begin{cases} T(idx)\equiv untainted \\ \forall i\in[0,n], T(s_i)\equiv untainted \end{cases} \Rightarrow W\left(arr[idx]\right)\equiv \begin{cases} T, \exists i\in[0,n], idx\geqslant s_i \\ F, \forall i\in[0,n], idx< s_i \end{cases} \tag{7}$$

判定规则 3.对于数组访问语句 *arr*[*idx*],如果数组下标 *idx* 为污染的,假设程序中有 *n* 条与数组下标 *idx* 相关的语句,构成语句序列为 $S_1,S_2,\ldots,S_n$,将每一条语句转换为约束表达式,则约束表达式序列为 $c_1,c_2,\ldots,c_n$,设第 *i* 条语句 S_i 在函数 *f* 的基本块 *bb* 中,此时数组 *arr* 对应的数组下标为 idx_i,数组长度列表为 $LenSet_i=\{len_1^i,len_2^i,\ldots,len_m^i\}$.如果存在一条语句 S_i,可以推导出 *idx* 大于等于 $LenSet_i$ 中任一长度(公式(8)中的条件 *A*);或者如果存在一条语句 S_i,可以推导出 *idx* 小于 0(公式(8)中的条件 *B*);或者对所有语句都不能推导出 *idx* 小于 $LenSet_i$ 中所有长度(公式(8)中的条件 *C*)或者大于等于 0(公式(8)中的条件 *D*),则该数组访问语句会导致数组越界.如果存在一条语句 S_i,可以推导出 *idx* 小于 $LenSet_i$ 中所有长度(公式(8)中的条件 *E*),并且存在一条语句 S_j,可以推导出 *idx* 大于 0(公式(8)中的条件 *F*),则该数组访问语句不会导致数组越界.

$$\left.\begin{aligned} &A:\left(\exists i,\exists k,!\left(c_i\rightarrow\left(idx\geqslant len_k^i\right)\right)\equiv UNSAT\right) \\ &B:\left(\exists i,!\left(c_i\rightarrow\left(idx<0\right)\right)\equiv UNSAT\right) \\ &C:\left(\forall i,\forall k,!\left(c_i\rightarrow\left(idx<len_k^i\right)\right)\equiv SAT\right) \\ &D:\left(\forall i,!\left(c_i\rightarrow\left(idx\geqslant 0\right)\right)\equiv SAT\right) \\ &E:\left(\exists i,\forall k,!\left(c_i\rightarrow\left(idx<len_k^i\right)\right)\equiv UNSAT\right) \\ &F:\left(\exists j,!\left(c_j\rightarrow\left(idx\geqslant 0\right)\right)\equiv UNSAT\right) \\ &T\left(idx\right)\equiv tainted\Rightarrow W\left(arr[idx]\right)=\begin{cases} T, A\vee B\vee C\vee D \\ F, E\wedge F \end{cases} \end{aligned}\right\} \tag{8}$$

依据以上 3 个判定规则,本文将参照算法 1 对程序进行数组越界检测.对于每一条数组访问语句,首先查询数组下标的污点值.如果数组下标为非污染的,则根据判定规则 1 和规则 2 检测该数组访问语句是否会导致数组越界.如果数组下标为污染的,则通过高精度的后向数据流分析,检测该数组下标是否可能导致数组越界.后向数据流分析时,使用 *OutState* 表示一个基本块在该基本块出口时的状态,是其所有后继基本块的 *InState* 的并集,如公式(4)所示(见第 3.2 节),此处代表在该基本块待检查是否会导致数组越界的数组访问语句信息的集合;*InState* 表示一个基本块在该基本块入口时的状态,如公式(5)所示(见第 3.2 节),此处是在其 *OutState* 的基础上,该基本块的语句根据判定规则 3 和表 1 对待检查是否会导致数组越界的数组访问语句信息的集合进行杀死和生成的结果.过程内后向数据流分析从数组访问语句所在的基本块开始,向上遍历函数中的每个基本块,到函数的入口点结束.当分析完一个基本块 *bb* 后向上分析其前驱基本块 *pred* 时,将同时根据 *bb* 位于 *pred* 的哪个分支,以决定在使用分支语句的条件时是否需要取反.如果在到达入口点时待检查语句集合为空,则将终止后向数据流分析,否则将继续进行过程间的后向数据流分析.首先,需要根据程序的函数调用图获取函数 *f* 的所有父函数.对于每个父函数,遍历其 CFG 并找到调用数组函数 *f* 的调用语句.对于每个数组边界条件,如果其下标索引与函数 *f* 的形参之一相同,则将获取相应的实参来构造新的数组边界条件,见算法 *interABChecker* 第 7 行.然后在父

函数中继续进行过程内的后向数据流分析.过程间后向数据流分析过程将一直执行,直到待检查语句集合为空或达到配置的检查深度.

Table 1 Types of statements related to array out-of-bound checking

表 1 数组越界检测相关的语句类型

语句类型	语句模式	简单匹配法	约束求解法
声明语句	Type idx=expr	idx=const	!(idx==expr→idx<len) !(idx==expr→idx>=len)
		idx=expr%const	!(idx==expr→idx<0) !(idx==expr→idx>=0)
赋值语句	idx=expr	ditto	ditto
复合赋值语句	idx op=expr	idx%=const	!(idx_1==idx_0 op expr→idx_1<len)
条件语句	if(expr)	idx<const idx<=const	!(expr→idx<len)
		idx>const idx>=const	!(expr→idx>0)
for 循环语句	for(…;expr;…)	ditto	ditto
while 循环语句	while(expr)	ditto	ditto

算法 1. 数组越界检测算法.

函数:**ABChecker**(**CallGraph**,**CFG**,**Depth**).

输入:CallGraph,CFG,Depth;

输出:Warnings.

```
1.for each f in CallGraph
/*in backwards topological order*/
2.   bbSet=∅
3.    for each BB in CFG off
/*in backwards topological order */
4.      for each ArrStmt in BB
5.         if T(ArrStmt.idx)==untainted
6.            if T(ArrStmt.LenSet)==tainted
7.               Warnings.add(ArrStmt)
8.            else
9.               for each Len in ArrStmt.LenSet
10.                  if ArrStmt.idx>=Len
11.                     Warnings.add(ArrStmt)
12.                     break
13.         else
14.            OutState[BB].add(ArrStmt)
15.      if OutState[BB]≠∅
16.         bbSet.add(all predecessors of BB)
17.   ABCSet=intraABChecker(bbSet,OutState)
18.   if ABCSet≠∅
19.      result=interABChecker(ABCSet,f,Depth−1)
20. Warnings.add(all ArrStmt in result)
```

函数:**intraABChecker**(**bbSet**,**OutState**).

```
1.while bbSet≠∅
2.   BB=bbSet.pop()
```

```
3.  for each succ of BB
4.      OutState[BB]+=InState[succ]
5.  InState[BB]=∅
6.  for each stmt in BB
7.      checkStmt(stmt, &OutState[BB])
8.  if OutState[BB]≠InState[BB]
9.      InState[BB]=OutState[BB]
10.      bbSet.add(all predecessors of BB)
11.  ABCSet=OutState[BB]
12.return ABCSet
```

函数:**checkStmt**(**stmt**,**OutState**[**BB**]).

```
1.for each ABC in OutState[BB]
2.  if imply(stmt, ArrStmt.idx,less,0)
3.      Warnings.add(ABC)
4.      OutState[BB].remove(ABC,low)
5.  else if imply(stmt,ArrStmt.idx,notless,0)
6.      OutState[BB].remove(ABC,low)
7.  for each len in ArrStmt.LenSet
8.      if imply(stmt,ArrStmt.idx,less,len)
9.          cnt++
10.      else if imply(stmt,ArrStmt.idx,notless,len)
11.          Warnings.add(ABC)
12.          OutState[BB].remove(ABC,up)
13.          break
14.  if cnt==ArrStmt.LenSet.size()
15.      OutState[BB].remove(ABC,up)
```

函数:**interABChecker**(**ABCSet**,**f**,**Depth**).

```
1.if Depth<=0 or ABCSet==∅
2.  return ABCSet
3.result=∅
4.for each caller off
5.  bbSet=∅
6.  bbSet.add(caller.callsite.BB)
7.  OutState[callerBB]=update(ABCSet)
8.  set=intraABChecker(bbSet,OutState)
9.  set2=interABChecker(set,caller,Depth−1)
10.  result+=set2
11.return result
```

如算法 1 中函数 *checkStmt* 所示,在数据流分析过程中,对于遇到的每一条语句 *stmt*,将根据表 1 和判定规则 3 进行处理.如果分析到一条语句可使公式(8)中的条件 A 或条件 E 满足,则后续数据流分析中不再关注 A、C、E;如果分析到一条语句可使公式(8)中的条件 B 或条件 F 满足,则在后向数据流分析过程中不再关注 B、D、F.在后向数据流分析过程中,主要关注与数组下标相关的语句,如表 1 的前两列所示,主要包括包含该数组下标的

条件语句(包括循环条件)和对数组下标的赋值表达式(包括声明赋值表达式和复合赋值表达式).其中,声明赋值表达式中 *idx* 声明的数据类型如果是无符号类型,则公式(8)中的条件 *F* 成立,同时,条件 *B* 和条件 *D* 不成立.本文提供了两种判断方法来检查公式(8)中条件 *A*~*F* 是否满足,即简单匹配方法和约束求解方法,将条件 *A*~*F* 统一用 $c_i \rightarrow (idx\ op\ expr)$来表示,以下描述中均先假设数组访问语句所在的基本块条件语句.

简单匹配处理方法.主要处理包含目标数组下标和常量的语句,即条件格式为$(idx\ op\ const_1) \rightarrow (idx\ op\ const_2)$,且当其中的两个操作符 *op* 一致时,通过比较两个常量 $const_1$ 和 $const_2$ 来判断条件的可满足性.针对不同类型的语句,如表 1 的第 3 列所示,具体处理规则如下.

- 赋值语句.只能处理语句为 *idx=const* 和 *idx=expr%const* 的情形,其中,若 *idx=const* 中 *const* 大于 0,则公式(8)中条件 *F* 成立,同时条件 *B* 和条件 *D* 不成立;同时,如果公式(8)中条件 *E* 中的数组长度 *len* 也为常量,且若语句中的常量 *const* 小于或等于所有数组长度 *len*,则条件 *E* 成立,同时条件 *A* 和条件 *C* 不成立;如果语句中的常量 *const* 大于任一常量数组长度 *len*,则条件 *A* 成立,报告数组越界.当语句为 *idx=expr%const* 时,则只在公式(8)中条件 *E* 中的数组长度 *len* 同时也为常量时,若常量 *const* 小于或等于所有数组长度 *len*,则条件 *E* 成立,同时条件 *A* 和条件 *C* 不成立.
- 复合赋值语句.只能处理语句为 *idx%=const* 的情形,判定方法与 *idx=expr%const* 相同.
- 条件语句.只能处理语句条件为 *idx<const*、*idx<=const*、*idx>const* 和 *idx>=const* 的情形.当条件为 *idx<const* 时,如果公式(8)中条件 *E* 中的数组长度 *len* 同时也为常量,且如果语句中的常量 *const* 小于或等于所有数组长度 *len*,则条件 *E* 成立,同时条件 *A* 和条件 *C* 不成立.当条件为 *idx<=const* 时,判定 *const* 是否小于数组长度 *len*.当条件为 *idx>const* 时,判定 *const* 是否大于–1;当条件为 *idx>=const* 时,判定 *const* 是否大于 0.

约束求解处理方法.直接将条件 $c_i \rightarrow (idx\ op\ expr)$作为约束,并将约束取反(即$!(cond \rightarrow idx<size)$交给约束求解器,通过约束求解来判断条件的可满足性.如果约束求解的结果为 UNSAT(不可满足),则表明原来的约束 $c_i \rightarrow (idx\ op\ expr)$恒为真,也就是说当前的语句 S_i 隐含了 *idx op expr*;如果约束求解的结果为 SAT(可满足),则表明原来的约束 $c_i \rightarrow (idx\ op\ expr)$不可满足.针对不同类型的语句,具体处理规则如下.

- 赋值语句.如表 1 第 4 列所示,将赋值语句 *idx=expr* 和待检查的数组边界检查条件 *idx*⟨*len*/*idx*⟩=*len*/*idx*<0/*idx*>=0,构成$!(idx==expr \rightarrow idx<len)$等约束交给约束求解器进行求解.
- 复合赋值语句.如表 1 第 4 列所示,将复合赋值语句 *idx op=expr* 和待检查的数组边界检查条件 *idx<len*,构成$!(idx_1==idx_0\ op\ expr \rightarrow idx_1<len)$约束交给约束求解器进行求解.
- 条件语句.当遇到“if”语句、“for”语句或“while”语句时,将抽取出语句中的条件 *expr* 和待检查的数组边界检查条件 *idx<len*,构成约束$!(expr \rightarrow idx<len)$交给约束求解器进行求解.

循环越界检测.如果在“for”或“while”条件中找不到对相应数组边界的检查,那么我们将检查数组下标是否是循环变量.如果“for”或“while”条件与模式 *idx<var* 相匹配,则在后续数据流分析过程中,公式(8)中的条件将用 *var* 替换 *idx* 以进行更新.也就是说将数组越界问题转换为循环越界问题继续检查.

当分析结束时,将报告数组越界警报.数组越界检查报告主要包括关于每个数组下标索引的以下信息:文件、行号、函数及其所在的数组表达式,以及待添加的边界检查条件.这些信息比较详细,可以帮助程序员更加方便、快速地定位和确认工具报告的数组越界警报,也可以作为修复推荐建议提供给程序员作为参考.

示例解析. 如图 2 所示代码,对于数组访问语句 *arr*[2],数组下标 2 为非污染,数组长度{*j*,*k*}为污染的,根据判定规则 1,确定为数组越界缺陷.对于数组访问语句 *s.noisy*[*n*]、*s.arr*[*i*]和 *tmp*[*i*],其数组下标 *n* 和 *i* 为污染的,将通过后向数据流分析并根据判定规则 3 检测是否会导致数组越界.由于结构体中的数组 *noisy* 和 *arr* 以及数组 *tmp* 是无符号类型,因此数组下标一定不小于 0,只需要检测是否越出数组上界.如图 5 所示的 CFG,我们从包含数组表达式的最底层基本块 *Block*2 开始执行数组越界检查,也就是从源码中的第 16 行代码开始进行分析.首先,将遍历基本块 *Block*2,未发现对数组下标边界的检查.然后继续向上分析,得到 *Block*2 的前驱块 *Block*4.接下来,我们得到 *Block*4 的后继(*Block*2 和 *Block*3),从而得到 *Block*4 中待检查的数组信息,即 *OutState* 为 16 行的 *s.arr*[*i*]中 *i*<15、*tmp*[*i*]中 *i*<3.由于 *Block*2 在 *Block*4 的 false 分支上,所以 if 条件为$!(i>=15)$.由此可以推出 *i*<15,因此,

找到了 16 行的 *s.arr*[*i*]应满足 *i*<15 的边界检查,第 16 行的 *s.arr*[*i*]将从待检查数组集合中移除.也就是说,*Block*4 的 *InState* 是 16 行的 *tmp*[*i*]中的,*i*<3.然后继续向上分析 *Block*5,在 *Block*5 中遇到 for 语句时,16 行的 *tmp*[*i*]中的 *i* 恰好为循环变量,转换为循环越界问题处理,即在循环上方检测循环上界 *n* 是否会超出数组 *tmp* 的长度,即检测 *n*<4.*Block*6 中的待检测数组信息 *OutState* 为 16 行的 *tmp*[*i*]中的 *n*<4,11 行的 *s.noisy*[*n*]中的 *n*<12.当遇到 *Block*6 中的赋值语句时,待检测数组信息将更新为 16 行的 *tmp*[*i*]中 *m*<4,11 行的 *s.noisy*[*n*]中的 *m*<12.因此,当遇到函数 *f* 的入口时,待检测数组信息不为空.如果配置的检测深度为 1,那么将报告数组越界警告.否则,将执行过程间的后向数据流分析.

在函数 *main* 中,根据实参更新待检测数组信息为 16 行的 *tmp*[*i*]中的 *argc*<5,11 行的 *s.noisy*[*n*]中的 *argc*<13.if 语句的条件为 *argc*+2<15,使用简单匹配方法,则 11 行的 *p.noisy*[*n*]的数组边界检查条件 *n*<12 可以被满足,因此,简单匹配方法中的警报"*test.c*,*line* 11,*p.noisy*[*n*],*n*<12"为误报.但是 12 行的循环上界 *n* 应小于 11 才可以保证 15 行的 *arr*[*i*]中的 *i*<10,因此,简单匹配方法和约束求解方法中的警报"*test.c*,*line* 12,*arr*[*i*],*n*<11"为真警报.

因此,当使用简单匹配方法进行判断时,无法匹配可以处理的模式,将会报告:

```
test.c,line 11,s.noisy[n],n<12;
test.c,line 16,tmp[i],i<3;
```

当使用约束求解方法进行判断时,可得!((*argc*+2<15)→(*argc*<13))≡*UNSAT*,!((*argc*+2<15)→(*argc*<5))≡*SAT*,因此将会报告:

```
test.c,line 16,tmp[i],i<3;
```

3 实现和实验评估

本文扩展了我们的前期工作[16,17],实现了一个面向数组越界缺陷检测的全自动跨平台的静态分析工具 Carraybound,优化了按需污点分析,并增加了按需指针分析,从而以此来分析数组长度的区间,引入了定理证明器 Z3[15],在数组越界检测过程中用以解决约束求解问题,工具的架构如图 6 所示.该工具可以在 Linux 和 Windows 系统上运行,底层依赖于 Clang 3.6 和约束求解器 Z3,由数组长度区间分析、按需污点分析和数组越界缺陷检测共 3 个模块组成.提供了可配置的功能实现用户按需调节检测精度,用户可以配置函数层数来控制执行过程间数据流分析的深度,并通过内存优化、求解优化等措施提升了工具的效率.

Fig.6 Architecture of Carraybound

图 6 Carraybound 工具架构

3.1 实现优化

内存优化.大规模程序总是包含大量 AST 文件.如果一次性读入所有 AST 文件,包含 AST 文件的所有内容,将会消耗大量内存.这将会严重制约 Carraybound 对大规模程序的支持.比如,PHP-5.6.16 包含 25 万行源代码和 211 个 AST 文件,当我们尝试一次性读入所有 AST 文件时,在 2GB 内存的机器上将无法运行.为了支持在有限

的内存资源下扫描 10 万行甚至 100 万行代码的程序,我们在 Carraybound 中实现了内存优化策略.内存优化策略的关键思想是使用一个 AST 队列仅在内存保留最新使用的 AST 文件,例如只保留 200 个 AST 文件.AST 文件越少,内存消耗就越少.并且,AST 队列的最大容量可以由用户配置.用户可以根据需求和计算机容量配置 AST 队列的最大容量.在分析 AST 的内容时,Carraybound 将首先检查相应的 AST 是否在内存中.如果 AST 在内存中,Carraybound 会将 AST 移动到队列的末尾.如果 AST 不在内存中,Carraybound 将从 AST 文件读入 AST 的内容.当 AST 队列达到其最大容量时,Carraybound 将删除最先读入的 AST.值得注意的是,如果用户设置了一个较小的最大 AST 数,Carraybound 将会更加频繁地读取 AST 文件.因此,如果有足够的内存,用户应选择更大的最大 AST 数,以减少频繁的读操作并提高 Carraybound 的效率.

求解优化.约束求解是非常耗时的,尤其是频繁地调用约束求解器将严重增加分析时间,制约工具的可扩展性.而在我们的方法中,由于需要计算不动点,会增加对相同约束求解的需求.因此,我们针对 Carraybound 分析方法的特征,在使用时进行了特殊优化.

- **结果缓存**.为了减少对约束求解器的调用,将保存函数内语句是否隐含数组边界检查的结果列表,首先查询这个列表,若未查询到结果,再调用约束求解器,这样可以大大减少对约束求解器的调用次数.
- **快速求解**.由于在通过数据流分析判断所遇到的表达式是否隐含数组边界检查时,总是将!(*cond*→*idx*<*size*)这样的约束条件交给约束求解器进行求解,并且这个约束条件满足 *cond* 中必须包含 *idx* 这个对象才能使得约束条件 UNSAT.因此,为了辅助约束求解器更快求解,我们通过映射表的比对提前完成约束的快速求解,过滤不满足 *cond* 中必须包含 *idx* 这个对象的约束条件,可以有效地减少对约束求解器的调用次数.
- **时间限制**.程序中可能存在一些按位操作及一些其他操作语句,对应到 Z3 约束求解时可能比较耗时.因此,我们为 Z3 提供了 *timeout* 的配置项.

3.2 实验评估

我们对 Carraybound 的实验评估主要试图回答以下几个问题.

Q1:Carraybound 的有效性?

Q2:Carraybound 的效率?

Q3:Carraybound 与已有方法/工具的比较?

Q4:Carraybound 发现真实缺陷(CVE)的能力如何?

实验对象和工具.为了评估 Carraybound 的有效性,我们选用了几个常用的开源项目作为实验对象,见表 2.由于在相关工作的文献中未发现可用的工具,因此我们与如下几个知名的可以检查数组越界缺陷的静态分析工具进行了比较,包含开源静态分析工具 Cppcheck[18]、商业静态分析工具 Checkmarx[19]和 HP Fortify[20].Cppcheck 是针对 C/C++语言的开源静态分析工具,该工具主要检查未定义行为相关的程序缺陷,包括除零错、整型溢出和越界访问等安全问题[18].Checkmarx 是一个基于源码的静态分析工具,对于被测程序,该工具会首先根据代码的元素和流程信息构造逻辑图,然后通过在该图上进行查询以发现程序中可疑的安全漏洞和业务逻辑问题.HP Fortify 是一个基于规则的源码级静态分析工具,支持对 25 种编程语言的漏洞分析.对这些工具的警报数目进行了统计,结果见表 2,其中,CAB-Simple 为赋值语句简单匹配处理方法,CAB-Z3 为约束求解处理方法.程序的规模最大可以达到 200+万行.我们通过人工审查程序源码,对表 2 中 Carraybound 报告的数组越界检查警报进行了人工确认,确认过程默认为两层函数调用.由于其他几个静态分析工具内存消耗不易统计,我们仅统计了其时间开销,并在表 2 的基础上增加两个大规模程序,见表 3.由于我们不知道表 2 中的程序包含多少个真正的数组越界缺陷,通过人工确认的方式也无法完全确认所有警报,因此,为了解答问题 4,我们在 CVE 中阅读了与缓冲区溢出漏洞相关的报告,找到其中几个由于数组越界缺陷导致的缓冲区溢出的程序以及其修复后的版本,见表 4.

Table 2 Warnings of Carraybound and the compared tools

表 2 Carraybound 与对比工具的警报统计

程序	规模 (KLOC)	CAB-simple		CAB-Z3		Cppcheck		Checkmarx		Fortify	
		W	*T*	*W*	*T*	*W*	*T*	*W*	*T*	*W*	*T*
libco-v1.0	6.0	3	2	3	2	0	0	10	0	0	0
libfreenect-0.5.7	34.7	10	10	10	10	0	0	10	0	–	0
vips-8.7.4	167.7	49	42	47	42	0	0	100	0	5	0
coreutils-8.30	206.8	12	5	10	5	0	0	305	1	35	2
curl-7.63.0	233.7	30	16	15	15	0	0	144	4	88	3
libxml2-2.9.9	2 302.4	9	5	6	5	0	0	96	0	–	0
总计	4 772.2	113	80	92	79	0	0	617	5	128	5

注:*W* 表示相应工具报告的警报数目.*T* 表示工具报告的警报中真警报的数目.–表示该项数据不可用,因为使用 Fortify 扫描这些程序时出错

Table 3 Time and memory consumption of Carraybound and the compared tools

表 3 Carraybound 与对比工具的时间和内存开销

程序	规模 (KLOC)	AST 文件数	CAB-simple 时间(s)	CAB-simple 内存(MB)	CAB-Z3 时间(s)	CAB-Z3 内存(MB)	Cppcheck 时间(s)	Checkmarx 时间(s)	Fortify 时间(s)
libco-v1.0	6.0	6	0.2	33	0.3	44	0.8	84	22
libfreenect-0.5.7	34.7	17	0.6	55	0.8	66	14.5	828	–
vips-8.7.4	167.7	411	110	3 866	117	3874	4 633	1 364	504
coreutils-8.30	206.8	393	39	1 225	36	1234	5 646	6 001	976
curl-7.63.0	233.7	179	11	556	12	565	466	1 191	429
vim-8.1.0818	838.6	81	142	1 785	140	1797	434	1 483	–
espruino-2.01	1 141.6	97	5	336	9	349	539	1 801	3 403
libxml2-2.9.9	2 302.4	50	171	2 093	171	2105	30	12 720	108

注:–表示该项数据不可用,因为使用 Fortify 扫描这些程序时出错

Table 4 Results of checking programs with known out-of-bound CVEs and the repaired programs

表 4 对包含已知数组越界 CVE 漏洞程序和修复后程序的检查结果

程序	CVE	修复版本	CAB-Z3		Cppcheck		Checkmarx		Fortify	
			缺陷版本	修复版本	缺陷版本	修复版本	缺陷版本	修复版本	缺陷版本	修复版本
file(1)(9611f3)	CVE-2017-1000249	(35c94d)	Yes	No	No	No	No	No	No	No
openjpeg-1.5.0	CVE-2012-3535	1.5.1	Yes	No	No	No	No	No	Yes	Yes
sendmail-8.12.7	CVE-2002-1337	8.12.8	Yes	No	No	No	No	No	Yes	No

注: (9611f3)和(35c94d)表示程序 git 的提交序号

Q1 有效性评估.我们分别统计了赋值语句简单匹配处理方法(CAB-simple)和约束求解处理方法(CAB-Z3)的误报情况,发现前者的平均误报率为 29.2%,后者的平均误报率为 16.3%.导致约束求解匹配处理方法误报的主要原因是我们无法处理一些库函数调用,如果在条件语句或者赋值语句中存在库函数调用并保证了数组边界,但是我们却无法判断从而导致误报.而赋值语句简单匹配处理方法相比约束求解匹配处理方法存在更多误报的原因是前者只简单匹配一些固定格式的语句并要求语句中特定位置为常量,很多复杂情形无法处理导致误报;而通过约束求解我们可以增强条件判断的能力,除了能处理更多的线性约束,甚至可以处理非线性约束.

Q2 效率评估.为了评估 Carraybound 的分析效率,我们统计了如表 3 所示的程序的分析时间和内存消耗.约束求解匹配处理方法由于调用约束求解器,总体会比赋值语句简单匹配处理方法消耗更多的时间和内存.但是时间平均增加了 1.53%,内存平均增加了 0.86%.使用约束求解方法并未造成明显的时间和内存消耗增加.这是因为,一方面,我们存储和复用了约束求解的结果,避免了冗余的约束求解;同时,我们针对求解 $expr1 \rightarrow expr2$ 约束进行了专门的优化,减少了对约束求解器的调用;另一方面,由于约束求解可以精确地判断程序语句是否对数组边界进行了检查,可以尽早地移除已经被满足的数组边界检查,从而从总体上能够节约开销.如表 3 所示,我们可以发现赋值语句简单匹配处理方法和约束求解匹配处理方法在时间和内存消耗上都随着程序规模的扩大,呈现接近线性趋势的增长,因此我们的方法具有很好的可扩展性.

Q3 与已有方法的比较.如表 2 和表 3 所示,有些工具无法指定只检测数组越界缺陷,会有更高的时间开销,我们不与其进行效率比较,这里列出来仅供参考.

• Cppcheck:如图 3 所示的示例,Cppcheck 未报告任何数组越界相关的警报.而经过实验,类似于“char a[5]; a[5]=0;”这样简单的数组越界,该工具是可以报告的.这表明,该工具可能存在一些漏报.同时,针对表 2 所列的被测程序,该工具均未报告数组越界警报.如表 3 所示,该工具由于不能指定单独检查越界问题,会消耗较长的时间进行检查其所支持的缺陷类型.

• Checkmarx:如图 3 所示的示例,Checkmarx 未报告任何数组越界相关的警报.表 2 中所列的被测程序一共报告了 617 个数组越界相关的警报,其中高风险警报 25 个,经人工确认有 1 个为可疑的数组越界缺陷,其余为低风险警报,经人工确认有 4 个为可疑的数组越界缺陷.在人工确认其报告时,发现该工具无法处理和循环相关的数组下标访问问题,即使数组下标为循环变量,当循环上界为数组上界时,仍然会误报数组越界.如表 3 所示,该工具在只选定几个与数组越界相关的缺陷类型进行检测时,仍会消耗较长的时间.我们使用时发现,该工具会消耗很长的时间对源码进行解析以生成该工具的一种中间表示,比如逻辑图,然后再在该图上进行查询以检查缺陷.虽然我们只指定了数组越界相关的缺陷类型,但是该构建逻辑图的过程是针对所有类型的缺陷的,因此会消耗较长时间.

• HP Fortify:如图 3 所示的示例,Fortify 只报告 test.c 中第 11 行 p.noisy[n]为缓冲区溢出警报点,而该警报实际为误报;并且漏报了 test.c 中第 15 行 arr[i]会因为第 12 行的循环导致数组越界/缓冲区溢出.表 3 所示的被测程序,该工具报告了大量缓冲区溢出警报,我们人工筛选出其中与数组越界相关的警报,经过人工确认发现多数为误报,并且多数情况下,无法处理和循环相关的数组下标访问问题.

Q4 对已知 CVE 缺陷的报告比较.见表 4,Carraybound 可以对缺陷版本的程序报告相应的数组越界警报,而对修复后的版本的程序将不再报告数组越界警报.Cppcheck 和 Checkmarx 对修复前后的版本均未报告相应的数组越界警报.而 HP Fortify 对 Sendmail 程序的修复前后的版本的报告是正确的,对 file 程序的前后版本均未报告,对 openjpeg 程序未检查出该缺陷已被修复.

3.3 实验讨论

上述实验结果表明,这些已有的开源和商业静态分析工具均不是专门针对数组越界检查的工具,它们的存在可以帮助程序员发现程序中的各种类型的缺陷.但是这些工具未进行精确的数据流分析和约束求解,因此对于数组越界这一类型的缺陷存在大量误报和漏报.

我们的工具采用数组长度区间分析、按需污点分析、精确的数据流分析和约束求解,所以有相对更少的误报和漏报.但在人工确认这些静态分析工具报告的警报时,我们也发现了本文工具 Carraybound 在实现上的一些不足,主要包括可扩展性和准确性两方面的不足.

可扩展性.由于约束求解比较耗时,尤其是求解一些复杂约束,比如一些按位运算时,约束求解器将无法在较短的时间内给出求解结果.这将会限制我们工具的可扩展性.在实验时只能通过设置 timeout 时间来跳过一些复杂的约束求解,但是这样做又可能会导致该工具的误报和漏报.

准确性.影响本文工具 Carraybound 准确性的问题主要包括:

(1) 类型转换:C 语言程序中经常存在类型转换,我们目前的工具实现中未能很好地处理该问题,因此可能会导致工具的误报和漏报.

(2) 复杂循环越界:有些情形下很难分析出数组下标和循环上界的关系,这会导致工具的误报和漏报.

(3) 库函数:由于采用静态分析方法,在不能获得库函数的源码实现的情况下,我们将无法判断这些库函数的功能和作用,但是可能这些库函数实现了对数组边界的检查和保证,因此会导致我们的工具产生误报.

(4) 复杂数组下标:程序中存在一些使用复杂表达式作为数组下标的情形,会导致工具产生误报.尤其是对于简单匹配方法,容易因为无法匹配越界检查条件,产生误报.比如数组下标 $2\times i+j$,而程序中可能是分别对 i 和 j 的范围约束,无法直接匹配到如 $2\times i+j<xx$ 的检查语句,这样就会导致误报.目前的工具出于可扩展性考虑,对约束求解设置了 timeout 时间,因此复杂数组下标也会导致约束求解方法的误报和漏报.比如数组下标为包含按位运

算的表达式,将会导致判定准则中的约束更为复杂,从而无法在指定时间内求解,进而导致误报.

4 相关工作

4.1 污点分析

动态污点分析是一种当前流行的软件分析方法.当前有很多工作通过进行动态污点分析来追踪软件中的隐藏漏洞[10,21−23].污点分析将可能包含恶意数据的外部输入作为污点源,比如网络数据包;然后,跟踪这些污点数据如何在整个程序执行过程中传播;当敏感数据(如堆栈中的返回地址或用户特权设置)被污点数据污染时执行相应的处理操作.

与动态污点分析相比,静态污点分析以静态方式追踪源码或二进制文件中的污点信息.STILL[24]是一个基于静态污染和初始化分析的防御机制,可以在各种互联网服务中(例如 Web 服务),检测嵌入在数据流中的恶意代码.为了减少污点分析的开销,TaintPipe[9]借助轻量级的运行时日志记录来生成紧凑的控制流信息,并产生多个线程,以流水线的方式并行地执行符号化污点分析.静态污点分析面对的另一个问题是耗费人力.大多数现有的静态污点分析工具会在潜在的易受攻击的程序位置报错.这会导致需要开发者人工确认报告,为此需要耗费大量的人工成本.Ceara 等人[25]提出了一种污点依赖序列算子,主要基于细粒度的数据流和控制流污点分析,为程序员提供需要被分析的路径的一些相关信息.

4.2 指针分析

Andersen 算法[26]和 Steensgaard 算法[27]是最具代表性的流不敏感的指针分析算法.Andersen 指针分析方法[26]是一种基于包含的经典的 C 语言指针指向分析算法,该算法被认为是最精确的流不敏感、上下文不敏感的指针指向分析算法.该算法将程序中的直接指向关系描述为变量与对象之间的约束关系集合,再通过求解约束关系集合的传递闭包,计算得到间接的指向关系,从而获得所有变量的、完整的指向关系集合.这种基于包含的思想被广泛应用在后续的指针指向分析工作中[28].Steensgaard 算法[27]是一种基于等价的指针分析算法,其复杂度接近于线性复杂度.但是流不敏感的指针分析将会影响后续静态分析的精度.目前也有一些流敏感的指针分析算法,这些方法通常是基于数据流分析[29],比如 Emami 算法[30]、Lam 算法[31]、Chase 算法[32]等.本文的流敏感、上下文敏感指针分析是按需分析,即只针对所关心的数组名进行指针分析.

4.3 数组越界检查

Xu 等人[33]提出了一种可以直接在不受信任的机器码上进行分析的方法,该方法依赖于这些程序的初始输入的类型状态信息和线性约束.Detlefs 等人[34]提出了一种针对常见程序错误的静态检查器,包括数组下标越界、空指针解引用和多线程程序中的并发类错误.该方法利用线性约束,自动合成循环不变量用于边界检查.Leroy 和 Rouaix[35]提出了一个理论模型,系统地把基于类型的运行时检查放入主机代码的接口程序中.Kellogg 等人[36]提出了一种在编译时检测数组越界的轻量级验证方法,但是为了达到线性验证时间,该方法需要开发者预先注释相关信息,例如程序边界信息.相比之下,我们的方法能够分析出程序边界.ABCD[37]用于按需消除无用的数组边界检查.它平均能够删除 45%的动态边界检查指令,有时可以实现接近理想化的优化.

当前也存在许多静态的数组越界检测工具[38−40].Chimdyalwar[8]对 5 种用于检查数组越界的静态分析工具进行了评估.其中,商业工具包括 Polyspace 和 Coverity;学术工具为 ARCHER,其他两个是开源工具 UNO 和 CBMC.Polyspace 是唯一无漏报的工具,但是由于它是内存密集型分析,不能以同样的高精度扩展到大规模程序上.相比之下,Coverity 可以支持百万行级别的代码分析,但存在大量误报.UNO 同时有误报和漏报,并且不能应用在大规模程序上.ARCHER 宣称可以运行在百万行级别的代码上,但是分析并不完善.CBMC 模型检查器进行了精确的分析,无法在大规模程序上达到相同精度.Nguyen 等人[39]提出了针对 Fortran 语言的数组越界静态检查方法.Arnaud 等人[38]提出了针对嵌入式程序中数组越界检查的静态分析方法,该方法处理的程序规模为 20+万行,我们的方法可以处理百万行程序.由于该文献提供的工具 CGS 是根据 NASA 程序定制的闭源工具,并且使用了 NASA 闭源程序作为被测对象,因此,我们在实验中没有与该方法进行比较.

4.4 针对数组越界的缓冲区溢出检测

当前有很多关于缓冲区溢出检测的工作.大多数工作在检测缓冲区溢出的同时也能检测数组越界缺陷.

Tance[41]提出了一种黑盒组合测试方法检测缓冲区溢出漏洞.Dinakar 等人[42]提出通过对内存进行细粒度的划分等技术来降低 C/C++程序动态数组越界检查的运行开销.如 Loginov[43]和 rtcc[44]等基于插桩的方法能在运行时检测是否出现缓冲区溢出.但是这些方法会引入额外的运行时开销,从而降低测试的效率.例如,Loginov 工具的额外开销高达 900%.SafeC[45]、Cyclone[46]和 DangDone[47]使用扩展的指针表示,这些扩展包含每个指针值的合法目标对象的对象基础信息和大小.使用这些指针要对程序进行大量修改以使用外部库,这些外部库函数通常是被包装好的用于转换指针的方法.此外,编写这样的封装对于间接函数调用以及访问全局变量或存储器中其他指针的函数来说可能是难以实现的.

预防技术是一种用于防止数组下标越界被利用的方法.例如,StackGuard[2]可能在检测到堆栈上的返回地址被覆盖后终止进程.运行时预防的现有方法具有显著的运行时开销.除此之外,这些方法在可能有漏洞的程序部署完成后生效.CFI[48]检查程序的控制流程是否在执行期间被劫持.这与我们的工作形成对比,我们的工作目的是在部署之前发现程序中的数组越界缺陷.

4.5 针对数组越界的模糊测试

模糊(fuzzing)测试是安全测试中使用最为广泛的黑盒测试方法之一.该方法在检测数组越界或缓冲区溢出问题中也发挥着重要作用[49−57].它主要通过程序崩溃检测数组越界缺陷.模糊测试通常从一个或多个合法输入开始,然后随机改变这些输入以获得新的测试输入.高级模糊测试技术[50]是基于生成的模糊测试技术,它为了解决具有复杂输入结构的程序的输入生成问题,通过基于语法的输入归约定义有效输入.Godefroid 等人[51]提出了一种替代的白盒模糊测试方法,结合符号执行和动态测试生成.虽然模糊测试可以检测到数组越界错误,但一个主要限制是代码覆盖率低.此外,一些数组越界的错误可能只读取越界的区域,因此不会导致崩溃,这样,模糊测试中的监视器可能无法检测到这种情况[52].我们的方法基于静态方法,可以实现高代码覆盖,并且可以检测不同类型的数组下标越界.

5 结论和未来工作

本文提出了一种基于污点分析的数组越界缺陷的静态检测方法,并实现了一个可以在 Windows 和 Linux 操作系统上运行的自动化静态分析工具——Carraybound.如果程序中存在数组越界缺陷,我们将报告相应的数组位置和待添加的数组边界条件.我们通过扫描一些真实程序的源代码来评估 Carraybound 工具.实验数据表明,Carraybound 可以快速报告在程序中没有进行数组边界检查的数组下标,在使用约束求解方法时,误报率大约为 16.3%.尽管 Carraybound 有一些误报和漏报,但它可以有效地减少程序员人工审查工作.我们的方法可以提供待增加的数组边界检查条件和位置,可以帮助程序员更加方便、快速地定位和确认所报告的数组越界警报,也可以作为修复推荐建议提供给程序员来参考.

目前,Carraybound 由于库函数等原因,可能导致误报,对于库函数有源码的情况,可以考虑通过函数摘要等技术完成更高精度的数组越界缺陷检测,对于库函数无源码的情况,可以考虑结合动态测试的方法来检测;另一方面,数组越界缺陷是一类特殊的缓冲区溢出缺陷,我们可以考虑扩展到对缓冲区溢出缺陷的检测,对于常见的缓冲区溢出相关 API,如 strcpy、memcpy,可以通过定义总结其溢出条件,建立缓冲区溢出模型,再利用数据流分析检测程序中是否有相应的越界检查语句,从而检测缓冲区溢出缺陷.

References:

[1] CWE. Improper validation of array index. https://cwe.mitre.org/data/definitions/129.html

[2] Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the USENIX Security Symp. 1998,98:63–78.

[3] CVE. http://www.cvedetails.com/vulnerabilities-by-types.php

[4] Ye T, Zhang L, Wang L, Li X. An empirical study on detecting and fixing buffer overflow bugs. In: Proc. of the IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). IEEE, 2016. 91–101.

[5] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2016. 786–791.

[6] Bao T, Gao F, Zhou Y, Li Y, Wang L, Li X. Automatically validating static buffer overflow warnings based on guided symbolic execution. Journal of Cyber Security, 2016,(2):46–60 (in Chinese with English abstract).

[7] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):860–882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[8] Chimdyalwar B. Survey of array out of bound access checkers for C code. In: Proc. of the 5th India Software Engineering Conf. ACM, 2012. 45–48.

[9] Ming J, Wu D, Xiao G, Wang J, Liu P. TaintPipe: Pipelined symbolic taint analysis. In: Proc. of the 24th {USENIX} Security Symp. ({USENIX} Security 15). 2015. 65–80.

[10] Newsome J, Song DX. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2005,5:3–4.

[11] Khedker U, Sanyal A, Sathe B. Data Flow Analysis: Theory and Practice. CRC Press, 2009.

[12] Kildall GA. A unified approach to global program optimization. In: Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. ACM, 1973. 194–206.

[13] Galler SJ, Aichernig BK. Survey on test data generation tools. Int'l Journal on Software Tools for Technology Transfer, 2014,16(6):727–751.

[14] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 337–340.

[15] Z3 theorem prover. https://z3.codeplex.com/

[16] Gao F, Chen T, Wang Y, Situ L, Wang L, Li X. Carraybound: Static array bounds checking in C programs based on taint analysis. In: Proc. of the 8th Asia-Pacific Symp. on Internetware. ACM, 2016. 81–90.

[17] Zhou Y. Extensible framework for static vulnerability detection based on taint analysis [Ph.D. Thesis]. Nanjing: Nanjing University, 2017 (in Chinese with English abstract).

[18] Cppcheck. http://cppcheck.net/

[19] Checkmarx. https://www.checkmarx.com/

[20] Fortify static code analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[21] Costa M, Crowcroft J, Castro M, Rowstron A, Zhou L, Zhang L, Barham P. Vigilante: End-to-end containment of Internet worms. ACM SIGOPS Operating Systems Review, 2005,39(5):133–147.

[22] Crandall JR, Su Z, Wu SF, Chong FT. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS). ACM, 2005. 235–248.

[23] Suh GE, Lee JW, Zhang D, Devadas S. Secure program execution via dynamic information flow tracking. ACM SIGPLAN Notices, 2004,39(11):85–96.

[24] Wang X, Jhi YC, Zhu S, Liu P. Still: Exploit code detection via static taint and initialization analyses. In: Proc. of the 2008 Annual Computer Security Applications Conf. (ACSAC). IEEE, 2008. 289–298.

[25] Ceara D, Mounier L, Potet ML. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In: Proc. of the 3rd Int'l Conf. on Software Testing, Verification, and Validation Workshops. IEEE, 2010. 371–380.

[26] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. University of Cophenhagen, 1994.

[27] Steensgaard B. Points-to analysis in almost linear time. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 1996. 32–41.

[28] Chen C, Huo W, Yu H, Feng X. A survey of optimization technology of inclusion-based pointer analysis. Jisuanji Xuebao/Chinese Journal of Computers, 2011,34(7):1224–1238 (in Chinese with English abstract).

[29] Pang L, Su X, Ma P, Zhao L. Research on flow sensitive demand driven alias analysis. Journal of Computer Research and Development, 2015,52(7):1620–1630 (in Chinese with English abstract).

[30] Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACM SIGPLAN Notices, 1994,29(6):242–256.

[31] Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for C programs. ACM SIGPLAN Notices, 1995,30(6):1–12.

[32] Chase DR, Wegman MN, Zadeck FK. Analysis of pointers and structures. ACM SIGPLAN Notices, 1990,39(4):343–359.

[33] Xu Z, Miller BP, Reps T. Safety checking of machine code. ACM SIGPLAN Notices, 2000,35(5):70–82.

[34] Detlefs DL, Leino KRM, Nelson G, Saxe JB. Extended Static Checking. 1998. [doi: 10.1007/978-0-387-35358-6_1]

[35] Leroy X, Rouaix F. Security properties of typed applets. In: Secure Internet Programming. Berlin, Heidelberg: Springer-Verlag, 1999. 147–182.

[36] Kellogg M, Dort V, Millstein S, Ernst MD. Lightweight verification of array indexing. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2018. 3–14.

[37] Bodík R, Gupta R, Sarkar V. ABCD: Eliminating array bounds checks on demand. ACM SIGPLAN Notices, 2000,35(5):321–333.

[38] Venet A, Brat G. Precise and efficient static array bound checking for large embedded C programs. ACM SIGPLAN Notices, 2004,39(6):231–242.

[39] Nguyen TVN, Irigoin F. Efficient and effective array bound checking. ACM Trans. on Programming Languages and Systems (TOPLAS), 2005,27(3):527–570.

[40] Popeea C, Xu DN, Chin WN. A practical and precise inference and specializer for array bound checks elimination. In: Proc. of the 2008 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation. ACM, 2008. 177–187.

[41] Wang W, Lei Y, Liu D, Kung D, Csallner C, Zhang D, Kacker R, Kuhn R. A combinatorial approach to detecting buffer overflow vulnerabilities. In: Proc. of the 41st IEEE/IFIP Int'l Conf. on Dependable Systems & Networks (DSN). IEEE, 2011. 269–278.

[42] Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. In: Proc. of the 28th Int'l Conf. on Software Engineering (ICSE). ACM, 2006. 162–171.

[43] Loginov A, Yong SH, Horwitz S, Reps T. Debugging via run-time type checking. In: Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer-Verlag, 2001. 217–232.

[44] Steffen JL. Adding run-time checking to the portable C compiler. Software: Practice and Experience, 1992,22(4):305–316.

[45] Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. In: Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation (PLDI). ACM, 1994. 290–301.

[46] Hicks M, Morrisett G, Grossman D, Jim T. Experience with safe manual memory-management in cyclone. In: Proc. of the 4th Int'l Symp. on Memory Management. ACM, 2004. 73–84.

[47] Wang Y, Gao F, Situ L, Wang L, Chen B, Liu Y, Zhao J, Li X. DangDone: Eliminating dangling pointers via intermediate pointers. In: Proc. of the 10th Asia-Pacific Symp. on Internetware. ACM, 2018. 6.

[48] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security (TISSEC), 2009,13(1):4.

[49] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, 2007.

[50] Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. ACM SIGPLAN Notices, 2008,43(6):206–215.

[51] Godefroid P, Levin MY, Molnar DA. Automated Whitebox fuzz testing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2008,8:151–166.

[52] McNally R, Yiu K, Grove D, Gerhardy D. Fuzzing: The state of the art. Defence Science and Technology Organisation Edinburgh, 2012. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=15CF9A7FD272D62D76FB5ED26DA3808F?doi=10.1.1.461.4627&rep=rep1& type=pdf

[53] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. ACM Trans. on Information and System Security (TISSEC), 2008,12(2):1–38.

[54] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. ACM SIGPLAN Notices, 2005,40(6):213–223.

[55] Xu RG, Godefroid P, Majumdar R. Testing for buffer overflows with length abstraction. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2008. 27–38.

[56] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2016,16(2016): 1–16.

[57] Pak BS. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution [Ph.D. Thesis]. School of Computer Science, Carnegie Mellon University, 2012.

附中文参考文献:

[6] 鲍铁匀,高凤娟,周严,李游,王林章,李宣东.基于目标制导符号执行的静态缓冲区溢出警报自动确认技术.信息安全学报,2016,(2):46–60.

[7] 王蕾,李丰,李炼,冯晓兵.污点分析技术的原理和实践应用.软件学报,2017,28(4):860–882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

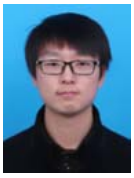
[17] 周严.基于污点分析的静态漏洞检测可扩展框架[博士学位论文].南京:南京大学,2017.

[28] 陈聪明,霍玮,于洪涛,冯晓兵.基于包含的指针分析优化技术综述.计算机学报,2011,34(7):1224–1238.

[29] 逄龙,苏小红,马培军,赵玲玲.流敏感按需指针别名分析算法.计算机研究与发展,2015,52(7):1620–1630.

高凤娟(1991－),女,学士,主要研究领域为软件工程,程序分析,软件测试,软件安全.

王豫(1991－),男,学士,主要研究领域为软件工程,程序分析,软件测试,软件安全.



陈天骄(1992－),男,硕士,主要研究领域为软件工程.

司徒凌云(1988－),男,博士,助理研究员,CCF 学生会员,主要研究领域为软件工程,信息安全,静态分析,模糊测试.

王林章(1973－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件工程,软件测试,软件安全.

李宣东(1963－),男,博士,博士生导师,CCF 会士,主要研究领域为复杂软件建模与分析,软件测试与验证.