基于程序特征谱整数溢出错误定位技术研究

惠战伟 黄 松 嵇孟雨
(解放军理工大学指挥自动化学院 南京 210007)
(全军军事训练软件测评中心 南京 210007)

摘 要 随着软件业的飞速发展,人们对软件质量的要求也越来越高.整数溢出错误以其高危性和隐蔽性成为影响软件安全性和可靠性的重要因素之一.如何准确定位整数溢出错误是软件安全领域研究的热点.论文改进了现有错误定位模型,构建了整数溢出错误定位模型INTRank.实验结果表明:基于INTRank模型的语句可疑度估计方法可以较为准确地计算语句可疑度,使得程序员能够按照基于语句可疑度的优先级顺序检查源代码,找出导致整数溢出错误的原因,同时本文方法具有较低的漏报率.

关键词 整数溢出错误定位;程序特征谱;定义使用对覆盖;分支覆盖
中图法分类号 TP311 **DOI号**: 10.3724/SP.J.1016.2012.02204

Research on Spectra-Based Integer Bug Localization

HUI Zhan-Wei HUANG Song JI Meng-Yu
(*College of Command Automation, PLAUST, Nanjing* 210007)
(*PLA Software Testing and Evaluation Center for Military Training, Nanjing* 210007)

Abstract With the development of the software industry, people require higher quality of software. Integer bug is considered to be one of the main factors for the lacking of safety and reliability in computer systems, since it is elusive and of high risk. How to locate the integer bug accurately is a hotspot in the field of software security testing. We propose a fault localization model INTRank, which improves the traditional model. The experiment result shows that the INTRank based approach could estimate the suspicious score of every statement under test in the program more accurately than the traditional ones. And programmers can examine the code based on the suspicious score of the priority to find out the root cause of the integer bug. Meanwhile our approach can reduce the false negative effectively.

Keywords integer bug localization; program spectra; define-use pair coverage; branch coverage

1 引 言

对于应用于航空航天、过程控制、核子能源、交通运输和医疗卫生等任务关键领域的软件,其计算结果的正确性是可靠性和安全性的关键部分.整数溢出错误是导致软件计算出错的原因之一.由于检测每个算数操作结果的代价太大,无法承受,许多商业软件中的整数溢出错误基本上没有得到有效检测.若程序对一个整数求出了一个非期望值,并且这些非期望值被用于数组索引或者循环变量等情形时,就会产生软件安全漏洞.这些安全漏洞通常会导致灾难性的后果.如1996年6月4日欧洲Ariane 5火箭爆炸事故[1]和2004年12月25日Comair航空

收稿日期:2012-06-30;最终修改稿收到日期:2012-08-09.本课题得到国家"八六三"高技术研究发展计划项目基金(2009AA01Z402)、江苏省自然科学基金(BK2012059,BK2012060)资助.**惠战伟**,男,1983年生,博士研究生,研究方向为软件测试、软件故障定位.E-mail:hzw_1983821@163.com.**黄 松**,男,1970年生,教授,博士生导师,研究领域为系统仿真、软件测试.**嵇孟雨**,男,1987年生,硕士研究生,中国计算机学会(CCF)学生会员,研究方向为软件测试.E-mail:jmypla@163.com.

公司的机组调度软件崩溃[1]都是由整数溢出错误引发的.

整数溢出错误在程序运行时存在数据流和控制流两种传播形式,并且在一组测试执行中,程序正常执行和异常终止执行有可能同时存在.本文基于上述两方面的考虑实现了对现有错误定位模型的改进,提出了一种新的错误定位模型 INTRank.

本文通过实验表明:使用基于 INTRank 的方法有助于程序员按照一定的优先级顺序检查代码,可以为其提供有效的诊断线索;基于 INTRank 模型的语句可疑度估计方法提高了定位整数溢出错误的准确率,从而提升了软件开发和维护的效率,同时整数溢出错误的漏报率也显著降低.

2　相关工作

2.1　整数溢出错误定位

目前针对整数溢出错误定位的检测和定位方法主要有以下 3 种类型:

(1)监控系统运行时的指令操作.按照整数溢出错误的特征制定一定的判定准则,在程序运行时监控整数溢出错误是否发生.

例如南京大学陈平等人[2]研究的 BRICK 可以实时监控程序运行中所有运算操作,利用一定的判定机制检测并定位整数溢出错误;

国防科学技术大学卢锡城等人[3]利用程序语言的抽象语义设计实现了一个自动化整数出错误测试系统.

(2)利用污点分析技术或者类型推理技术找到发生整数溢出错误的位置.

例如 Ceesay 等人[4]通过一个基于 Cqual[5]的静态工具来追踪不可信数据;Ashcraft 等人[6]采用基于程序员编写的编译扩展来追踪不可信数据,并在追踪过程中检测程序错误.以 valgrind[7]、memcheck[8]和 Aftersight[9]等系统为代表的动态污点追踪与传播技术,可以在不需要任何源码的情况下,运行时检测程序执行路径中是否存在如下错误:对未初始化内存的引用、悬挂指针和内存泄漏等.

(3)安全整数类、C 编译器扩展等整数溢出错误防护机制.

例如 IntSafe、SafeInt[3]等整数溢出编程防护机制被集成到编译器中帮助程序员避免不安全的整数操作.

以上这些技术实质上都是用整数溢出的静态或者动态规约作为判定依据直接定位整数出错误发生的位置,这种方法有明显的局限性,如定位错误类型有限,存在不同程度的漏报率以及无法向程序员提供诊断线索等问题.

2.2　软件错误定位方法

为了解决错误定位过程中消耗时间多的问题,研究人员提出了多种自动化的错误定位技术,目前有多种自动化软件错误定位技术分类方法.按照获取错误定位所需数据方法的不同,可以分为基于程序特征谱(Spectra-Based Fault Localization,SBFL)的定位技术和基于模型(Model-Based Fault Localization,MBFL)的定位技术.本文采用 Wong 等人[10]的分类方法,基于程序特征谱错误定位技术中与本文相关的两种代表性的错误定位技术包括:

(1)基于程序特征谱覆盖统计的错误定位技术

Tarantula[11]是一种基于可执行语句命中谱的自动化错误定位技术.Tarantula 利用测试用例产生的执行追踪信息(程序特征谱)和执行结果信息(失效或通过),计算每个语句的可疑度.

虞凯等人[12]利用多种程序特征谱模型,考虑控制与数据依赖关系捕捉程序的异常行为,在可疑度估计阶段,虞凯等针对两种缺陷类型分别建立可疑度计算模型 CDBug 和 DDBug;最后将这两种计算模型整合,可以在预先不知道程序所包含缺陷类型的情况下确定语句的可疑度.

(2)基于程序特征谱比较的错误定位技术

NNQ[13](Nearest Neighbor Queries)是 Renieris 和 Reiss 提出的基于程序特征谱比较的错误定位方法.NNQ 方法假设存在有一个失效运行和许多成功运行,运用一种距离度量方法,搜索与失效运行最"相似"的成功运行,比较这两种运行,排除同时被成功运行和失效运行所覆盖的语句,最后产生一个包含可疑语句集的报告.NNQ 采用 Hamming 距离和排列距离两种方法度量两个程序特征谱之间的相似度.程序员检查生成的可疑语句集,如果错误语句确实存在于可疑语句集,那么错误定位成功;如果可疑语句集中没有错误语句,那么首先通过构造程序依赖图(图的节点是语句,边包括数据依赖边和控制依赖边),检查其它节点中最近节点,直至找到错误语句.这种方法的优点是可以利用最近邻居模型有目的地选取用于错误定位的测试用例,而缺点是往往错误语句不会被可疑语句集包括,程序员需要检查更多的代码才能定位错误语句.

基于程序特征谱软件错误定位技术是一种利用

程序的运行覆盖信息和运行结果反向推理程序失效原因的错误定位方法.本文将基于程序特征谱的错误定位用于整数溢出错误定位,目的是解决现有整数溢出错误定位技术不能有效地支持程序员调试修改程序的问题.

3 基于程序特征谱整数溢出的错误定位方法

3.1 问题分析

首先本文通过一个引例(表1)说明现有基于程序特征谱的错误定位模型被用于定位整数溢出错误时,存在准确率低的问题,引例来自乔治亚工学院①.用LOUPE模型[12]和CP模型[14]分别计算引例中语句的可疑度,本文发现:分别按照两个模型计算错误语句7的可疑度时,语句7的可疑度都不为最大值.

下面分别分析LOUPE模型和CP模型不能准确定位整数溢出错误的原因:

(1)LOUPE模型基于如下假设,即程序错误发生后,其直接控制依赖或者直接数据依赖的语句"立刻"导致程序状态发生偏差.LOUPE模型中的非分支判断语句的可疑度值和包含该语句的定义使用对可疑度的平均值有关;而分支判断语句的可疑度计算还需考虑其分支的可疑度.如表2所示,语句14

表1 引例 mid.c

程序	语句	T1	T2	T3	T4	T5	CP	LOUPE	INTRank
#include〈*stdio.h*〉	1								
main (int *argc*, char **argv*[])	2								
{ int x, y, z, m;	3								
if (*argc* <4)	4	•	•	•	•	•	0	0.632	0
{*fprintf*(*stderr*,"*Error*\n");	5						−1	0	0
exit (1); }	6						−1	0	0
x=(short int)*atoi* (*argv*[1]); //截断错误	7	•	•	•	•	•	0	0.422	0.421
y=*atoi* (*argv*[2]);	8	•	•	•	•	•	0	0.408	0.313
z=*atoi* (*argv*[3]);	9	•	•	•	•	•	0	0.394	0.160
m=*z*;	10	•	•	•	•	•	0	0.632	0
if (*y*<*z*)	11	•	•	•	•	•	−0.086	0.632	0.622
{if (*x*<*y*)	12		•		•	•	2	0.707	0.371
m=*y*;	13		•		•		0	0	0
else if (*x*<*z*)	14					•	0	0.707	0.354
m=*x*; }	15					•	0	0.707	0
else if (*x*>*y*)	16	•		•			1	0.500	0.250
m=*y*;	17	•		•			0	0.500	0.250
else if (*x*>*z*)	18						−1	0	0
m=*x*;	19						−1	0	0
printf ("%d\n", *m*); }	20	•	•	•	•	•	0	0.371	0
*T*1:*x*=7,*y*=5,*z*=2 *T*2:*x*=1,*y*=2,*z*=3 *T*3:*x*=−317 696,*y*=2,*z*=−63 579 *T*4:*x*=2,*y*=5,*z*=7 *T*5:*x*=−327 696,*y*=−65 579,*z*=−3	Result	P	P	F	P	F	INTRank:*rank*=2 CP:*rank*=14 LOUPE:*rank*=10 注:"•"表示程序语句被测试用例覆盖.		

表2 LOUPE和CP计算分支可疑度和定义使用对可疑度

控制依赖(分支)	T1	T2	T3	T4	T5	LOUPE模型分支可疑度/CP模型分支可疑度
(4,5)						0/−1
(4,7)	•	•	•	•	•	0.632/0
(11,12)		•		•	•	0.408/−0.143
(11,16)	•		•			0.5/0.2
(12,13)		•		•		0/−1
(12,14)					•	0.707/1
(14,15)					•	0.707/1
(16,17)	•		•			0.5/0.2
(16,18)						0/−1
(18,19)						0/−1

① http://www.static.cc.gatech.edu/aristotle/Tools/Aristotle/samples/mid.c.

（续　表）

数据依赖(定义使用对)	T1	T2	T3	T4	T5	LOUPE 模型定义使用对可疑度
(7,12,x)		•		•	•	0.408
(7,14,x)					•	0.707
(7,15,x)					•	0.707
(7,16,x)	•		•			0.707
(7,18,x)						0
(7,19,x)						0
(8,11,y)	•	•	•	•	•	0.632
(8,12,y)		•		•	•	0.408
(8,13,y)		•		•		0
(8,16,y)	•		•			0.5
(8,17,y)	•		•			0.5
(9,10,z)	•	•	•	•	•	0.632
(9,11,z)	•	•	•	•	•	0.632
(9,10,z)						0
(9,14,z)					•	0.707
(9,18,z)						0
(10,20,m)	•	•	•	•	•	0.632
(13,20,m)		•		•		0
(15,20,m)					•	0.707
(17,20,m)	•		•			0.5
(19,20,m)						0
Result	P	P	F	P	F	注：“•”表示程序分支或定义使用对被测试用例覆盖. 测试用例与表 1 相同.

(s_{14}表示语句 14)所在定义使用对(9,14,z)和(7,14,x)的可疑度值都为 0.707，按照 LOUPE 模型计算 $susp_d(s_{14})=(0.707+0.707)/2=0.707$，另外分支判断语句 14 仅存在分支(14,15)，且分支(14,15)的可疑度为 0.707，所以按照 LOUPE 计算模型 $susp_c(s_{14})=0.707-0=0.707$.

$susp(s_{14})=max(susp_c(s_{14}),susp_d(s_{14}))=0.707$. 因为包含语句 7($s_7$表示语句 7)的定义使用对(7,18,$x$)、(7,19,$x$)的可疑度都为 0，在取平均数时对语句 7 可疑度值的计算产生干扰，所以真正的错误语句 s_7 的可疑度值为 $susp_d(s_7)=(0.707+0.707+0.707+0.408+0+0)/6=0.422$(因为 s_7 为非分支判断语句，所以仅考虑 $susp_d(s_7)$)，0.422 并不是最大可疑度值.

(2) CP 模型仅考虑控制流传播，导致模型只能分析到基本块层次，对于基本块中的语句则不能很好地区分可疑度. 如表 1 所示，语句 7～语句 10 属于同一个基本块，所以使用 CP 模型计算这些语句的可疑度值的结果相同. 并且因为 CP 模型仅考虑影响控制流的错误，所以影响控制流的分支判断语句 12 的可疑度为 1(为最大值)，如表 3 所示，程序正是执行到语句 12 才与正确的控制流发生偏离.

3.2　前提假设

在程序执行过程中，错误语句会影响程序的某种状态(值或控制流)，然后这种错误的转换会通过程序的继续执行，将受影响的程序状态传播至程序最终产生失效的位置[15].

如表 1 所示，T5：$x=-327\,696$，$y=-65\,579$，$z=-3$ 为失效测试用例，原因是在程序入口处对变量 x 赋值时，由于使用强制类型转换($atoi()$函数返回值为 int 型)，将输入参数由 int 型转换为 short int，由于输入参数 $x=-327696<-32\,768$(short int 型的最小值)，满足了触发截断错误的约束条件，程序发生截断错误，使得 x 实际接受的值为 −16(被影响的程序状态). 错误的 x 值被用于 if 条件语句作为控制流的判断条件，最终导致程序失效.

表 3　输入为 T5 程序错误传播示例

	正确执行的分支	错误执行的分支
控制流分析	(4,7) (11,12) (12,13)	(4,7) (11,12) (12,14) (14,15)

	正确执行的定义使用对			错误执行的定义使用对		
	变量名	定义语句	使用语句	变量名	定义语句	使用语句
定义使用分析	x	7	12	x	7	12、14、15
	y	8	11、12	y	8	11、12
	z	9	10、11	z	9	10、11、14

如表 3 所示，当程序执行失效测试用例 T5 时，控制流与预期的正确执行之间发生偏差，正确执行应该是(4,7)→(11,12)→(12,13)，但是实际执行为(4,7)→(11,12)→(12,14)→(14,5)；同时变量 x、z 的使用语句也与正确的执行不一致，如表 3 所示.

因此整数溢出错误发生后控制流和数据流都有

可能发生偏差,而且程序错误语句往往与程序发生状态偏差的位置存在一定的“传播距离”.

因此可以得出以下结论:

(1)在程序执行过程中,整数溢出错误语句会对程序状态产生影响,而错误状态的传播则会通过数据流或者控制流.

(2)失效测试用例所覆盖的分支或者定义使用对中,存在某个分支或者定义使用对的可疑度为最大值.如表2所示,定义使用对(9,14,z)和分支(14,15)分别为失效测试用例T5所覆盖的定义使用对和分支,同时他们的可疑度值都为最大值1.

基于以上两点,可以确定整数溢出错误定位模型的前提条件:

(1)程序错误状态传播方式包括数据流传播和控制流传播.所以本文同时建立基于定义使用对覆盖特征谱和基于分支覆盖特征谱的错误定位模型,最后将两种模型整合.

(2)在程序执行过程中,由于程序的错误状态在语句之间通过数据流或控制流传播,因此程序语句可疑度之间存在相关性.

同时,可以得到本文方法的应用范围:

(1)如果整数溢出错误发生后程序仍得出正确结果,则此方法无效.

(2)如果整数溢出错误发生后立刻对程序执行状态产生影响,或者错误语句相对独立(即与其他语句不存在依赖关系),则此方法与现有方法相比没有明显优势.

3.3 本文方法

在3.2节基础上本文提出一种面向整数溢出错误的错误定位模型INTRank,如图1所示.

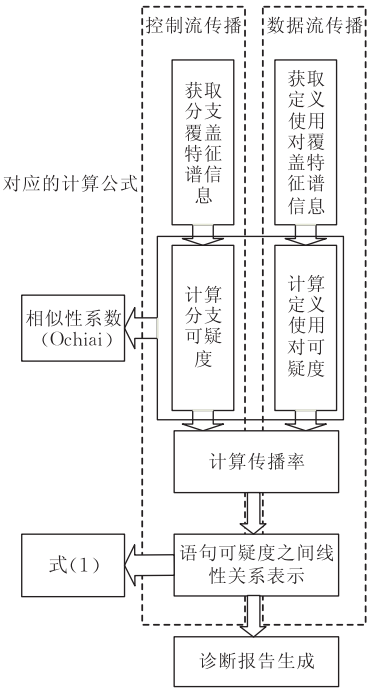


图1 INTRank模型概况

INTRank模型具体分为以下5个步骤:

(1)收集程序特征谱信息.

(2)计算分支和定义使用对的可疑度.

我们选取Ochiai公式[16]作为分支和定义使用对的可疑度计算公式,如下所示:

$$sim(e)=\frac{a_{10}}{\sqrt{(a_{10}+a_{11})\times(a_{10}+a_{00})}}.$$

其中,a_{10}表示覆盖到边e的失效测试用例数,a_{11}表示覆盖边e的成功测试用例数,a_{00}表示未覆盖边e的失效测试用例数.

相似性系数公式Ochiai原本被运用于植物学分类领域,文献[16]中Abreu等人将现有的相似性系数公式用于软件错误定位,比较发现Ochiai的性能一直优于其它相似性系数公式.

(3)计算错误传播率.

错误传播率是指程序错误状态经过某分支(或者定义使用对)传播影响程序状态的可能性.如表2中,定义使用对(8,11,y)和定义使用对(9,11,z)的可疑度都为0.632,所以定义使用对(8,11,y)的错误传播率:

$$prob(8,11,y)=\frac{susp_{du\text{-}pair}(8,11,y)}{\sum_{\forall(*,11,*)}susp_{du\text{-}pair}(*,11,*)}=0.632/(0.632+0.632)=0.5.$$

(4)建立语句可疑度之间的线性关系.

本文借鉴CP模型的方法用线性关系近似表示语句可疑度之间存在的相关性.但是与CP模型不同,本文考虑在程序执行中程序错误状态终止传播情形(例如程序异常终止),因此对程序中任意语句s在整数溢出错误传播过程中的情形做以下假设:

情形1. 程序执行过程中,程序的错误状态经语句s传播至下一个语句s',语句s'为语句s在分支覆盖图或者定义使用对覆盖图中的子节点.

情形2. 程序执行过程中,程序的错误状态在语句s处终止.

本文将两种情形的概率之和作为语句s的可疑度值.

本文将程序分支覆盖图和定义使用对覆盖图抽象为一个有向无环图G,将错误状态传播抽象为错误状态在G中节点向相邻节点迁移.

假设本文得到节点 A 所有出边和入边的可疑度，分别是入边 $susp_{edge}(e_1)$，$susp_{edge}(e_2)$ 和出边 $susp_{edge}(e_3)$，$susp_{edge}(e_4)$，如图 2 所示.

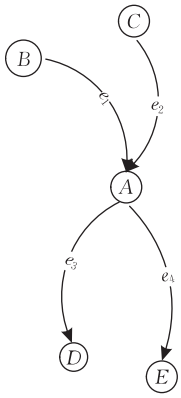


图 2　错误状态传播图例

对于情景 1，则用全概率公式表示节点 A 的可疑度 $susp_s^{(1)}(A)=P(A|D)susp_s(D)+P(A|E)susp_s(E)$，这里 $P(A|D)$ 表示节点 D 被错误状态影响时，错误状态由 A 迁移至 D（即通过 e_3 传播）的概率，即错误传播率. $P(A|D)susp_s(D)$ 表示节点 D 被错误状态影响时，节点 A 包含缺陷的可能性，同理 $P(A|E)susp_s(E)$ 表示节点 E 被错误状态影响时，节点 A 包含缺陷的可能性. 这里 $P(A|D)=prob(e_3)$，$P(A|E)=prob(e_4)$.

因此对于情形 1，$susp_s^{(1)}(A)=prob(e_3)susp_s(D)+prob(e_4)susp_s(E)$.

对于情形 2，由于节点 A 无子节点，本文直接将包含节点 A 的所有入边的可疑度求和作为节点 A 的可疑度，$susp_s^{(2)}(A)=susp_{edge}(e_1)+susp_{edge}(e_2)$.

直觉上在有向无环图 G 中，包含节点 A 的所有边中，入边数越多，则发生情形 2 的可能性越大，同样出边数越多情形 1 发生的可能性越大，故本文得到式(1)：

$$susp_s(A)=\frac{|e_{in}|}{|e_{in}|+|e_{out}|}susp_s^{(2)}(A)+\frac{|e_{out}|}{|e_{in}|+|e_{out}|}susp_s^{(1)}(A) \quad (1)$$

其中情形 1 的可能性为 $\frac{|e_{out}|}{|e_{in}|+|e_{out}|}$，表示包括节点 A 的出边数与包括节点 A 的所有边数之比；情形 2 的可能性为 $\frac{|e_{in}|}{|e_{in}|+|e_{out}|}$，表示包括节点 A 的入边数与包括节点 A 的所有边数之比.

当 $\frac{|e_{out}|}{|e_{in}|+|e_{out}|}=0$ 时表示节点 A 无子节点，例如节点 A 为程序出口语句，或者在执行所有测试用例时程序都异常退出，当 $\frac{|e_{in}|}{|e_{in}|+|e_{out}|}=0$ 时表示节点 A 无入边，例如节点 A 为程序入口语句.

按照式(1)，本文将程序中每个语句的可疑度写成方程组的形式. 分别在数据流和控制流情形下建立关于语句可疑度的线性方程组，并将可疑度计算过程转化为求解 n 元线性方程组问题，使用高斯消元法求解方程组.

(5) 诊断报告.

① 模型整合

由于不知道整数溢出错误发生后错误状态是通过控制流还是数据流传播，对于语句 n_i，本文使用式(2)计算最终语句 n_i 的可疑度.

$$susp(n_i)=\max(susp_{du}(n_i),susp_b(n_i)) \quad (2)$$

式(2)表示语句 n_i 的可疑度为 $susp_{du}(n_1)$（数据流传播）和 $susp_b(n_i)$（控制流传播）中可疑度较大的值.

② 特殊情况处理

若语句 n_i 不属于任何定义使用对，则 $susp_{du}(n_i)=0$；若语句 n_i 不属于任何分支，则其 $susp_b(n_i)=0$.

③ 语句排序

本文用秩（$rank$）表示语句在诊断报告中的排名，秩越小表示被程序员检查的优先级越高.

本文采用 last-line 策略处理语句可疑度值相同的情况，即对于所有可疑度值相同的语句在取初始排序中最后一个语句的秩作为这些语句的秩. 表 1 中本文使用 INTRank 模型计算语句可疑度，错误语句 7 的可疑度为 0.421，为第二大的可疑度值.

4　实验评价

4.1　实验概述

本文选择两个任务关键软件（tcas 和 schedule）作为被测程序，tcas 和 schedule 均来自 SIR 的西门子测试套件①.

实验的软件环境是 CentOS4.8 操作系统，JDK1.6.0_13，glibc 2.2.3；硬件环境是 Dell OPIPLEX 755，Intel 双核处理器.

本实验环境中整型变量取值范围如表 4 所示.

我们使用 WET② 运行架构收集分支和定义使用对覆盖信息.

① http://sir.unl.edu/portal/index.html.
② http://wet.cs.ucr.edu/index.html.

表 4 本实验环境中整型变量的取值范围

整数类型	范围
int	[−2147483648,+2147483647]
short int	[−32768,+32767]

4.2 实验数据

4.2.1 故障注入

由于 tcas 程序中主要是逻辑运算,无内存使用函数,并且只有少量算数运算,因此本文在 tcas 中仅注入截断错误,即将 int 型强制转换为 short int. 由于变量 *Own_Tracked_Alt* 和 *Other_Tracked_Alt* 分别表示为己方飞机所在海拔高度和被侦测飞机所在海拔高度,获取极端数据的可能性大(整数溢出错误的发生往往与输入极端数据有关[3]),所以本文分别在这两个变量的定义语句和使用语句注入截断错误. 注入故障的策略主要分为单故障注入和多故障注入. v1～v6 为单故障版本,v7～v19 为多故障版本.

schedule 中使用了动态链表数据结构,所以本文选择最容易引发内存安全问题的 *malloc*() 函数和最容易引发算数溢出的计数器变量(*n* 变量和 *mem_count* 变量)注入整数溢出错误. 本文在 schedule 中注入了算数溢出和符号错误,v1～v6 为单故障程序版本,v7 和 v8 为多故障程序版本.

4.2.2 测试数据

本文从 tcas/testplans. alt/testplans-bigcov 中选择 suite. 271 作为初始测试用例,将每个测试用例中的第 4 个和第 6 个参数同时加上 32767,例如 976 1 1 5378 390 1000 2 641 741 1 0 0 转化为 976 1 1 38145 390 33767 2 641 741 1 0 0. 这样参数变量 *Own_Tracked_Alt* 和 *Other_Tracked_Alt* 的取值范围正好控制在[32767,2147483647],满足截断错误的触发条件. 另外本文通过代码覆盖率检测工具 gcov 验证,转化后的测试用例集分支覆盖率为 100%,语句覆盖率为 98.63%,如图 3 所示. 由于 tcas 中存在不可达语句,所以语句覆盖率不能达到 100%,此为软件设计错误,在本实验中不予考虑.

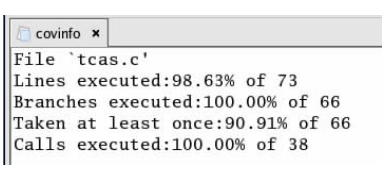


图 3 TCAS gcov 执行结果

schedule 的测试用例本文选择/testplans. alt/testplans-bigcov/suite1000 作为测试用例以保证最大覆盖率如图 4 所示.

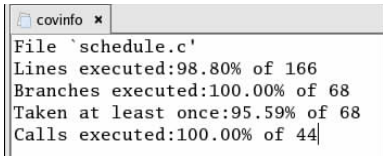


图 4 schedule gcov 执行结果

4.3 评价分析

4.3.1 评价方法

本文使用文献[17]中的评价方法对错误定位的准确性进行评估.

错误定位方法的准确性可以表述为,按照错误定位模型对语句的可疑度排名顺序检查代码,程序员找到错误语句之前所必须检查的代码数量. 程序员检查的语句数越少,说明错误定位方法准确性越高,同时也说明无需检查的代码数越多. 本文用无需检查代码占可执行语句数的比例定量表示错误定位方法的准确性.

定义错误定位方法的准确性为

$$q = 1 - \frac{rank}{total} \tag{3}$$

其中,*rank* 表示诊断报告中语句的秩,*total* 表示可执行语句总数.

对于多故障版本程序前提下的错误定位方法的准确性本文使用文献[18]的评价方法.

多故障版本程序前提下的错误定位方法准确率为找到首个错误语句前必须检查的语句个数,占可执行语句总数的百分比.

定义漏报率为

$$NP = 1 - \frac{D}{E} \tag{4}$$

其中,D 为诊断报告中注入错误的数量,E 为注入错误的总数量.

本文定义整数溢出错误定位方法定位整数溢出错误的能力 C 为 $NP=0$ 的程序版本数占所有注入相同整数溢出错误类型的程序版本数的百分比

$$C = \frac{num(versions(NP = 0))}{num(versions(all))} \tag{5}$$

4.3.2 实验结果

(1)单故障程序版本

本文列出了 tcas 和 schedule 程序各个单故障版本的错误定位准确率的对比图如图 5 和图 6 所示,另外本文将 q 值划分为 5 个区间,统计了每种方法的 q 值在这 5 个区间中的分布如表 5 和表 6 所示.

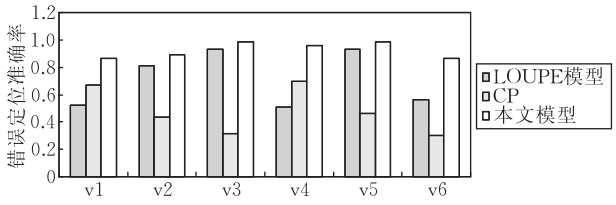


图 5　tcas 错误定位方法准确率对比(单故障版本)

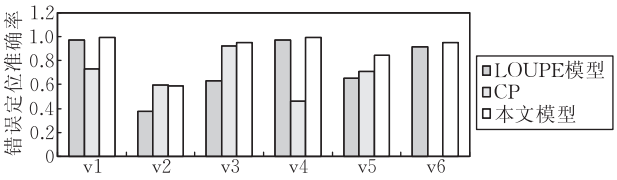


图 6　schedule 错误定位方法准确率对比(单故障版本)

表 5　tcas 各单故障版本准确率 q 的区间分布

q 的划分区间	LOUPE 模型准确率/%	CP 模型准确率/%	本文模型准确率/%
0.8～1.0	50	0	100
0.6～0.8	0	33.3	0
0.4～0.6	50	33.3	0
0.2～0.4	0	33.3	0
0～0.2	0	0	0

表 6　schedule 单故障版本准确率 q 的区间分布

q 的划分区间	LOUPE 模型准确率/%	CP 模型准确率/%	本文模型准确率/%
0.8～1.0	50	16.7	83.3
0.6～0.8	33.3	33.3	0
0.4～0.6	0	33.3	16.7
0.2～0.4	16.7	16.7	0
0～0.2	0	0	0

(2) 多故障程序版本

tcas 和 schedule 程序各个多故障版本的错误定位准确率如图 7、图 8 所示. 另外与单故障程序版本类似,本文分别将 q 的值划分为 5 个区间,统计了每种方法 q 的值在这 5 个区间中的分布如表 7 所示.

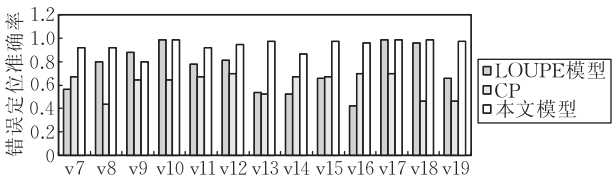


图 7　tcas 错误定位方法准确率对比
(多故障版本:发现首个错误语句的 q 值)

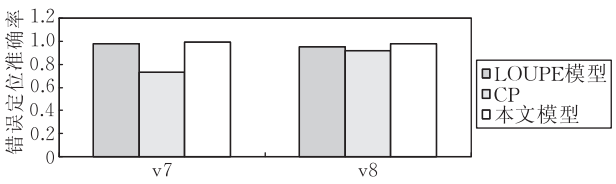


图 8　schedule 错误定位方法准确率对比
(多故障版本:发现首个错误语句的 q 值)

表 7　tcas 多故障版本程序错误定位准确率 q 的区间分布

q 的划分区间	LOUPE 模型准确率/%	CP 模型准确率/%	本文模型准确率/%
0.8～1.0	38.4	0	92.3
0.6～0.8	30.8	69	7.7
0.4～0.6	30.8	31	0
0.2～0.4	0	0	0
0～0.2	0	0	0

(3) 现有整数溢出错误检测工具结果

本文选择现有的两种典型整数溢出错误检测工具作为研究对象. splint 为静态分析工具,可以使用类型推理方法检测源代码中的类型冲突. valgrind-memcheck 为动态检测工具,运用动态追踪污点传播技术监控不信任数据,若不信任数据在内存分配函数使用中造成空指针使用、内存泄露等错误,则工具报错. 两个工具的检测结果如下.

表 8　splint 检测结果(tcas)

版本号	注入错误数	报告错误数	NP/%	版本号	注入错误数	报告错误数	NP/%
v1	1	4	100	v11	2	5	50
v2	1	5	0	v12	2	4	100
v3	1	4	100	v13	2	4	100
v4	1	4	100	v14	2	5	50
v5	1	4	100	v15	2	4	100
v6	1	5	0	v16	2	5	50
v7	2	4	100	v17	2	4	100
v8	2	6	0	v18	2	5	50
v9	2	5	50	v19	2	5	50
v10	2	4	100				

表 9　splint 检测结果(schedule)

版本号	注入错误数	报告错误数	NP/%	版本号	注入错误数	报告错误数	NP/%
v1	1	39	100	v5	1	39	100
v2	1	40	0	v6	1	40	0
v3	1	40	0	v7	2	39	100
v4	1	39	100	v8	2	41	0

表 10　valgrind-memcheck 检测结果(tcas)

版本号	注入错误数	报告错误数	NP/%	版本号	注入错误数	报告错误数	NP/%
v1	1	0	100	v11	2	0	100
v2	1	0	100	v12	2	0	100
v3	1	0	100	v13	2	0	100
v4	1	0	100	v14	2	0	100
v5	1	0	100	v15	2	0	100
v6	1	0	100	v16	2	0	100
v7	2	0	100	v17	2	0	100
v8	2	0	100	v18	2	0	100
v9	2	0	100	v19	2	0	100
v10	2	0	100				

表 11　valgrind-memcheck 检测结果(schedule)

版本号	注入错误数	报告错误数	NP/%	版本号	注入错误数	报告错误数	NP/%
v1	1	0	100	v5	1	1	0
v2	1	0	100	v6	1	1	0
v3	1	1	0	v7	2	0	100
v4	1	0	100	v8	2	1	50

4.3.3 实验讨论

(1)错误定位模型有效性对比

tcas 程序中如图 5 和表 6 所示,对于本实验中注入的 6 个单故障版本程序,本文方法优于现有错误定位方法.

将 q 的范围划分为 5 个区间,统计每个方法的准确率在各个区间的个数,结果发现 6 个单故障版本程序中本文方法的准确率 100% 在 0.8～1.0 之间,LOUPE 仅有 50%,CP 模型的准确率则均匀分布在中间 3 个区间内.

而在 schedule 的单故障程序中,本文方法有 83.3% 的 q 值在 0.8～1.0 之间,如表 6 所示.

如图 7 和图 8 所示本文方法对于 tcas 多故障程序版本情况下发现首个错误语句的效率明显优于现有方法.虽然 v9 程序 LOUPE 的准确率比本文方法高,但是其平均准确率却低于本文方法.v10 和 v17 版本程序,LOUPE 的准确率与本文方法相同,但是平均准确率同样低于本文方法.所以本文方法对于实验中的多故障程序版本准确率总体上优于 LOUPE 和 CP.

在 13 个多故障版本的 tcas 程序中,本文方法有 92.3% 的 q 值在 0.8～1.0 之间(如表 7),说明本文方法定位多故障的准确率优于现有方法.

(2)与现有整数溢出错误定位工具比较

从本实验结果发现 splint 可以检测符号错误和截断错误,但是无法检测算数溢出.而 valgrind-memcheck 可以检测发生在 malloc 函数使用中的整数溢出错误,但是错误报告仅仅定位到错误语句所在函数(如图 9),并不能指导程序员有效地调试程序.并且在 tcas 的检测过程中,由于 tcas 没有内存使用函数,因此 valgrind-memcheck 不报错.

```
==995== Invalid write of size 4
==995==    at 0x80484DC: new_list (in /root/Desktop/tui/schedule/schedule)
==995==    by 0x8048967: init_prio_queue (in /root/Desktop/tui/schedule/
schedule)
==995==    by 0x8048A43: main (in /root/Desktop/tui/schedule/schedule)
```

图 9 valgrind-memcheck 的错误报告

本文分别列出本实验的每种方法对 3 种整数溢出的定位能力 C,即 $NP=0$ 的程序版本数占所有注入相同整数溢出错误类型的程序版本数的百分比,如表 12 所示.从中可以发现静态分析工具可以有效地定位符号错误;基于程序特征谱的整数溢出错误定位方法可以定位全部算数溢出错误,而 splint 和 valgrind-memcheck 无法定位算数溢出错误.

(3)进一步分析

整数溢出错误在程序运行中通过数据流和控制流传播,而现有的 CP 模型只考虑通过控制流传播的程序错误,因此使用本文提出的 INTRank 模型定位整数溢出错误的准确性比 CP 模型高,如图 5～图 8 所示.

由于程序语句之间存在数据或者控制依赖关系,因此程序语句的可疑度之间同样存在相关性.现有 LOUPE 模型虽然考虑到语句之间存在控制依赖和数据依赖关系,但是该模型将分支和定义使用对的可疑度直接作为语句可疑度的作法忽略了整数溢出错误在程序运行时存在一定的"传播距离".因此 INTRank 模型的准确性比 LOUPE 模型高,如图 5～图 8 所示.

在与现有整数溢出错误检测方法的对比中,本文方法和现有基于程序特征谱错误定位方法基本上能够定位所有整数溢出错误定位类型(CP 除外,如表 12 所示),漏报率极低.这是由于基于程序特征谱错误定位方法是从程序运行结果反向推理程序失效的原因.无论程序中存在何种类型整数溢出错误,只要导致程序失效,基于程序特征谱错误定位方法就能定位该错误.

表 12 检测 3 种整数溢出错误的能力 C 的对比

类型	检测能力 C				
	Splint/%	Valgrind-memcheck/%	本文模型/%	LOUPE 模型/%	CP 模型/%
截断错误	15.80	0	100	100	100
符号错误	100	37.5	100	100	75
算数溢出	0	0	100	100	100

5 结论与展望

本文在分析整数溢出错误在程序运行时的传播特征基础上建立了错误定位模型 INTRank,提出了基于 INTRank 模型的语句可疑度方法,最后实现了原型工具 INT-LM,运行界面如图 10 所示.

通过对 tcas 和 schedule 进行故障注入,生成 27 个故障版本程序,故障类型涵盖截断错误、符号错误和算数溢出这 3 种基本类型.实验结果显示:

(1)在满足测试用例最大覆盖率前提下,本文方法可以较准确地定位注入的整数溢出错误语句.

(2)单故障程序前提下,INTRank 模型定位 tcas 6 个故障版本程序中整数溢出错误的准确率 100% 在 0.8～1.0 之间,schedule 的 6 个故障程序中有 83.3% 的准确率在 0.8～1.0 之间.而多故障

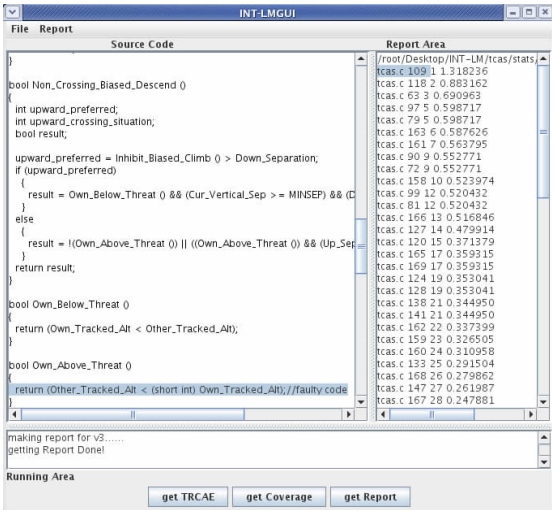


图 10　INT-LM 运行界面

前提下 INTRank 模型的准确率总体上优于 LOUPE 和 CP 模型，这也进一步说明在满足测试用例最大覆盖率前提下，直接将定义使用对或者分支的可疑度赋值于语句的方法不能实现对整数溢出错误的准确定位. 程序语句不能视为相互独立的个体，他们之间的可疑度值可以近似为一种线性关系.

(3) 基于程序特征谱整数溢出错误定位技术的漏报率比 splint 和 valgrind-memcheck 低. 实验结果表明，splint 和 valgrind-memcheck 都存在漏报现象，特别是算数溢出的漏报率达到 100%；而本文方法可以定位所有注入的整数溢出错误，漏报率为 0.

本文在某些方面还有待进一步的改进和探索. 由于本文实验中的初始测试输入数据均来自西门子测试套件①，因此下一步应该研究面向整数溢出错误定位的测试输入数据生成方法. 本文实现的原型工具使用 WET 运行架构获取覆盖信息，但是 WET 运行架构的运行开销比较大，会产生大量冗余信息，目前此工具只能运用于中小规模程序，所以如何高效地获取程序运行信息将是下一步的研究重点.

参　考　文　献

[1] Sercord R C. Secure Coding in C and C++. USA: Addison-Wesley, 2006

[2] Chen Ping, Wang Yi, Xin Zhi, Mao Bing, Xie Li. BRICK: A binary tool for run-time detecting and locating integer-based vulnerability//Proceedings of the International Conference on Availability, Reliability and Security. Fukuoka Institute of Technology, Fukuoka, Japan, 2009: 208-215

[3] Lu Xi-Cheng, Li Gen, Lu Kai et al. High-trusted-software-oriented automatic testing for integer overflow bugs. Journal of Software, 2010, 21(2): 179-193(in Chinese)
(卢锡城，李根，卢凯等. 面向高可信软件的整数溢出错误的自动化测试. 软件学报，2010，21(2)：179-193)

[4] Ceesay E N, Zhou J, Gertz M, Levitt K, Bishop M. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs//Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment. Berlin, Germany, 2006: 1-16

[5] Zhang X, Edwards A, Jaeger T. Using CQUAL for static analysis of authorization hook placement//Proceedings of the 11th Usenix Security Symposium. San Francisco, USA, 2002: 33-48

[6] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes//Proceedings of the IEEE Symposium on Security and Privacy. Washington, USA, 2002: 143-159

[7] Nethercote N, Seward J. Valgrind: A framework for heavy-weight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, 42(6): 89-100

[8] Nethercote N, Seward J. How to shadow every byte of memory used by a program//Proceedings of the ACM/Usenix International Conference on Virtual Execution Environments. New York, USA, 2007: 65-74

[9] Chow J, Garfinkel T, Chen P M. Decoupling dynamic program analysis from execution in virtual environments//Proceedings of the 2008 Usenix Annual Technical Conference. Boston, USA, 2008: 1-14

[10] Wong W E, Debroy V. Software fault localization. IEEE Transactions on Reliability, 2010, 59(3): 449-482

[11] Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique//Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering. Long Beach, California, USA, 2005: 273-282

[12] Yu Kai, Lin Meng-Xiang, Gao Qing, Zhang Hui, Zhang Xiang-Yu. Locating faults using multiple spectra-specific models//Proceedings of the 2011 ACM Symposium on Applied Computing. TaiChung, China, 2011: 1404-1410

[13] Renieris M, Reiss S P. Fault localization with nearest neighbor queries//Proceedings of the 18th IEEE International Conference on Automated Software Engineering. IEEE Computer Society, Montreal, Canada, 2003: 30-39

[14] Zhang Zhen-Yu, Chan W K, Tse T H, Jiang Bo, Wang Xin-Ming. Capturing propagation of infected program states//Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Amsterdam, Netherlands, 2009: 43-52

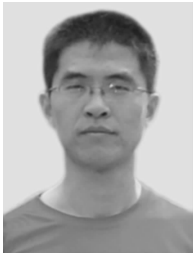
[15] Voas J M. PIE: A dynamic failure-based technique. IEEE Transactions on Software Engineering, 1992, 18(8): 717-727

[16] Abreu R, Zoeteweij P, Golsteijn R et al. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 2009, 82(11): 1780-1792

① http://sir.unl.edu/portal/index.html.

[17] Abreu R, Zoeteweij P, Gemund A J C van. On the accuracy of spectrum-based fault localization//Proceedings of Testing: Academic and Industrial Conference Practice and Research Technique. IEEE Computer Society, Cumberland Lodge, UK, 2007: 89-98

[18] Wong E, Wei T, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization//Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation. Lillehammer, Norway, 2008: 42-51



HUI Zhan-Wei, born in 1983, Ph. D. candidate. His research interests include software testing and software defect detection.

HUANG Song, born in 1970, Ph. D., professor, Ph. D. supervisor. His research interests include system simulation and software testing.

JI Meng-Yu, born in 1987, M. S. candidate. His research interest is software testing.

Background

Our research is funded by the National High Technology Research and Development Program (863 Program) of China (No. 2009AA01Z402), the Natural Science Foundation of Jiangsu Province, China (Grant Nos. BK2012059, BK2012060). In this paper, we study how to apply spectra-based fault localization technique to locate integer bug which is seldom focused by other experts. Testing for debug is a meaningful study field. The security or safety problems are often caused by the flaws in source code which are introduced during the development period.

In the past, most of the experts research how to find the inputs which trigger the integer overflow, however ignoring the root cause of integer overflow. The root cause of integer bug is that the presentation range of value by machine can't match range of value in the arithmetic operation. The traditional integer bug localization technology is of high false negative. Meanwhile they can't help programmer examine the code by the order of the priority.

In this paper, we use the spectra-based fault localization technique to locate integer bug. Based on investigating the characteristics of the error state propagation during the program running, we propose a fault localization model INTRank.

Based on the INTRank approach, programmers could estimate the suspicious score of the statements of the program more accurately. They could also examine the code by the suspicious score based on the order of the priority to find out the root cause of the integer bugs.

In previous researches, we have studied the approach to derive the requirements of security testing and evaluated the security quality of the software under test.