软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(2):219−230 [doi: 10.13328/j.cnki.jos.004915]

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

一种基于不变量的软错误检测方法*

马骏驰[1,2], 汪 芸[1,2]

[1](东南大学 计算机科学与工程学院,江苏 南京 211189)

[2](计算机网络和信息集成教育部重点实验室(东南大学),江苏 南京 211189)

通讯作者: 汪芸, E-mail: yunwang@seu.edu.cn

摘 要: 软错误是高辐照空间环境下影响计算可靠性的主要因素,结果错误(silent data corruption,简称 SDC)是软错误造成的一种特殊的故障类型.针对 SDC 难以检测的问题,提出了一种基于不变量的检测方法.不变量是运行时刻保持不变的程序特征.在软错误发生后,由于程序受到影响,不变量一般不再满足.根据该原理,在源代码中插入以不变量为内容的断言,利用发生软错误后断言报错来检测软错误.首先,根据错误传播分析确定了检测位置,提取了检测位置的不变量;定义了表征不变量检测能力的渗透率,在同一检测位置依据渗透率将不变量转化为断言.通过错误注入实验,验证了该检测方法的有效性.实验结果表明:该检测方法具备较高的检出率和较低的检测代价,为星载系统的软错误防护提供了新的解决思路.

关键词: 单粒子翻转;结果错误;错误检测;不变量

中图法分类号: TP302

中文引用格式: 马骏驰,汪芸.一种基于不变量的软错误检测方法.软件学报,2016,27(2):219−230. http://www.jos.org.cn/1000-9825/4915.htm

英文引用格式: Ma JC, Wang Y. Approach for detecting soft error by using program invariant. Ruan Jian Xue Bao/Journal of Software, 2016,27(2):219−230 (in Chinese). http://www.jos.org.cn/1000-9825/4915.htm

Approach for Detecting Soft Error by Using Program Invariant

MA Jun-Chi[1,2], WANG Yun[1,2]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

[2](Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education, Nanjing 211189, China)

Abstract: Soft error has a great influence on computing reliability of space devices and could result in silent data corruption (SDC), which means wrong outcomes of a program without any crash detected. As SDC-causing fault always propagates silently, it is very difficult to detect SDC. In this paper, an approach for detecting SDC is proposed by using program invariant. A program invariant is a set of properties of program. Normally, the invariant holds during runtime. But when soft error occurs, the invariant is often violated due to the impact of soft error. Based on this principle, invariant-based asserts are inserted into source code. Once an exception is thrown by an assert, it indicates that soft error is detected. By analyzing the propagation of the fault that leads to SDC, the locations where asserts are embedded are selected and then invariants are extracted. Some of the invariants are converted to asserts based on their permeability, which indicates the capabilities of detecting soft error. The proposed approach is evaluated by fault injection experiment which shows that it achieves high coverage with low overhead. The approach broadens the ways of protecting satellite system from soft error.

Key words: single event upset; silent data corruption; error detection; program invariant

单粒子翻转(single event upset)是由于高能粒子轰击半导体电路导致的 PN 结瞬间充放电现象,常发生于大

* 收稿时间: 2015-02-11; 修改时间: 2015-06-03, 2015-07-12, 2015-07-30; 采用时间: 2015-09-10; jos 在线出版时间: 2015-11-12
CNKI 网络优先出版: 2015-11-11 17:04:04, http://www.cnki.net/kcms/detail/11.2560.TP.20151111.1704.006.html

气层外的电子设备中[1,2].单粒子翻转会造成半导体电路的逻辑跳变.例如,7(二进制 00000111b)由于其中第 0 位的 1 变 0,数值变为 6(二进制 00000110b).这种由单粒子翻转造成的电路瞬态故障称为软错误.软错误会影响系统的正常运行,最严重的情况下甚至导致卫星失控.1997 年,美国 AT&T 公司的 Telstar 401 卫星因软错误而停止服务,波及了金融、股票市场的运行[3].近年来,芯片集成晶体管数呈指数级增长,在性能得到大幅提升的同时,软错误率也按照摩尔定律快速增长[4].软错误防护已经成为星载系统必须要解决的关键问题.

软错误引起的故障类型大致可以分为 4 种[5],包括屏蔽(benign)、崩溃(crash)、挂起(hang)和结果错误(silent data corruption,简称 SDC),见表 1.其中,SDC 的错误传播过程最隐蔽,整个过程不会显示出任何异常表现,只是程序运行的最终结果发生了错误.

Table1 Error categories caused by soft error and their differences

表 1 软错误引起的故障类型及其区别

类型	完整运行	结果正确
屏蔽	是	是
崩溃	否(程序直接退出)	无结果
挂起	否(系统资源耗尽)	无结果
SDC	是	否

软错误防护的过程一般分为检测和恢复两个阶段:检测阶段是通过检测器捕获到软错误的过程,而恢复阶段是将系统调整到正确状态的过程.检测是防护方法的第 1 步,也是一把双刃剑,一方面所起的作用至关重要,另一方面又会降低系统性能.由于隐蔽性的特点,SDC 是最难检测的故障类型.

软错误检测大致可以分为冗余、现象捕获和程序级断言这 3 种方法.冗余方法[6,7]设置了运算副本,通过比较副本的运算结果来检测软错误.冗余方法的检出率较高,但检测代价也很高.随着星载系统对性能、功耗的要求越来越高,低检测代价的技术成为研究的热点,现象捕获方法[8,9]是其中的代表.现象是指系统的异常表现,包括分支预测失效、cache 命中率低等.当捕捉到异常表现时,就认为发生了软错误.现象捕获方法的检出率高、代价低,但明显的不足是不能检测 SDC,因为 SDC 不会出现异常表现.程序级断言方法[4,10−13]通过执行断言(assertion)来检测软错误.断言的内容是程序正常运行时的逻辑或数值特征.现有工作大多采用数值区间作为断言,检测代价低,但检出率不高.

针对 SDC 难以检测的问题,本文提出了一种基于不变量的检测方法.不变量是运行时刻保持不变的程序特征[14,15].比如,$x=4\times y$ 是关于变量 x,y 的一个不变量,代表 x,y 满足一种线性关系.在程序正常运行的情况下,不变量是满足的;但在发生软错误后,由于变量的数值、程序的执行路径等可能会偏离正常运行时的状态,造成了不变量不再满足.利用这种性质,本文在源代码中插入由不变量转化成的断言(如 assert(x==4×y)).在执行断言时,一旦发现不变量不满足,就认为检测到软错误.

本文所提出的方法按照上述分类属于程序级断言方法,但断言内容的形式更丰富、检出率更高,是对现有方法的重要补充.本文的检测方法在保证对 SDC 较高的检出率的同时,检测代价较低,因而系统因检测而造成的性能损失较小;在不修改底层硬件的前提下,只需添加少量代码就能完成系统加固,易于在星载系统中实现.本文的主要贡献有:

1) 以函数为基本单元,分析了 SDC 的错误传播过程,得出函数间和函数内部的传播特征.指出接口指令(见定义 1)在导致 SDC 的错误传播中的作用,由此提出并证明了导致 SDC 的必要条件.
2) 提出了基于不变量的检测方法.根据 SDC 的错误传播特征,设计了不变量的提取方法.针对因提取的不变量数量较多而导致检测代价很高的问题,定义了表征不变量检测能力的渗透率,并研究了渗透率的计算方法.通过渗透率对不变量进行筛选,极大地降低了检测代价,并取得了较高的检出率.
3) 基于检测方法设计并实现了程序加固系统 Radish.以本文的检测方法为基础,Radish 能够自动完成对 c 语言程序的不变量提取、断言添加,从而提高程序的可靠性.

1 相关工作

程序级断言通过执行断言来检测软错误.断言的内容是程序正常运行时的逻辑或数值特征.特征提取的方式可以分为自动提取和人工提取.自动提取的特征较为简单,通常是数值的合法区间.在程序运行时获取变量的数值,根据数值的分布得出合法区间.人工提取的特征包括变量的关系、函数等,形式较为复杂.通过程序员分析函数功能、逻辑得出.以下分类进行介绍.

1.1 自动提取

iSWAT[10]采用了最简单的单个合法区间的形式,直接判断变量的值是否位于合法区间内.FaultScreening[4]和 ASED[11]采用了双合法区间.对于同一变量的值分布,双合法区间比单个合法区间的合法数值更少,断言满足的概率更小.因而理论上讲,双合法区间的检测能力更强.

FaultScreening 生成双合法区间的过程见表 2.每次读入一个变量的值,并修改区间.当新值包含于双合法区间中时,区间不变;否则,将新值并入某个区间中,使得新的双合法区间总长度最短.例如表 2,当读入 5,72 时,分别填入空白的区间.当读入 6 时,并入区间 1,因为并入区间 1 的长度更短.当读入 5 004 时,由于 5 004 比 72 大很多,5 004 独占区间 2.最终,区间 1 为[5,72],区间 2 为[5004,5004].如果变量的值不在这两个区间中,就认为发生了软错误.

Table 2 Process of generating intervals of FaultScreening

表 2 FaultScreening 的区间生成过程

新值	区间 1	区间 2
5	–	–
72	5-5	–
6	5-5	72-72
5 004	5-6	72-72
–	5-72	5 004

1.2 人工提取

LPD[12]将函数的功能描述、变量的关系等特征作为断言.如,以 e 为底的指数函数 exp 的两组输入 x,y 满足 $\exp(x)\times\exp(y)=\exp(x+y)$,以该等式作为断言,可以检出函数输出 $\exp(x),\exp(y)$的错误.LPD 的检出率较高,但设计断言需要对程序有深入了解.如,设计 exp 的断言需要程序员了解函数 exp 的功能和其背后所隐含的数学原理.

EA[13]将信号的属性特征作为断言.信号被分为离散信号和连续信号:连续信号的属性特征包括合法区间、变化率范围等;离散信号的属性特征包括合法取值集合、状态转移图等.EA 的检出率较高,但程序员需要熟知控制系统的模块功能、各信号的特征.

1.3 小 结

综上所述,人工提取方式生成的断言检出率较高,但是对程序员有较高的要求.尤其对于复杂程序,特征提取的难度很大.自动提取的特征以变量的合法区间为主,形式比较单一,检出率相对较低.在实际应用中,人工提取需要程序员大量阅读文档,工作难度大、完成周期长;自动提取具备更强的可操作性、可扩展性.

为了改进自动提取关系单一的不足,本文引入不变量技术.不变量是运行时刻保持不变的程序特征.不变量技术主要应用于软件测试领域.通过不变量自动生成文档,定义在软件测试和维护中不能轻易改变的关系[14].通过比对提取的不变量和设计目标,还能找出软件设计的缺陷和程序 bug[16].

2 SDC 的错误传播分析

为了设计针对 SDC 的检测方法,本节在指令级对 SDC 的错误传播进行分析,找出错误传播的特征.程序的结果一般是通过 printf,cout 等系统库函数进行输出.本文将进行结果输出的函数称为输出函数.首先对错误模型做如下假设:

(1) 操作码发生错误一般会导致崩溃[17],因而本文只考虑操作数的错误,后文中,指令发生错误特指操作数发生错误.

(2) 本文不考虑输出函数内部指令产生的错误.输出函数的代码一般不能由用户修改,所以无法插入断言进行保护.本文将输出函数看作黑箱,不考虑其内部的传播过程.

基于假设(2),输出函数的错误是从外部传入的.导致 SDC 的执行过程没有抛出异常,每条指令都是合法的,但结果是错误的.由于结果是由输出函数输出的,而输出函数的参数是输出的内容或地址,所以导致SDC的直接原因是输出函数的参数发生了错误.在传播到输出函数参数前,错误首先会在程序的不同函数间和函数内部传播.函数间和函数内部错误传播的方式具有不同的特征,分别对其进行分析.

2.1 函数间错误传播

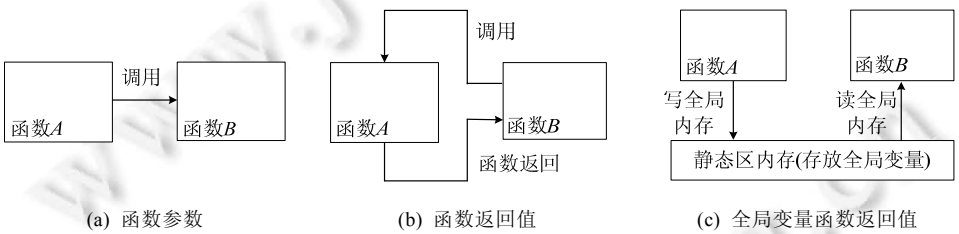
函数间的错误传播是通过数据交互完成的.假设有两个函数 *A*,*B*,函数 *B* 读取函数 *A* 写的数据,使得函数 *A* 将错误传播给函数 *B*.如图 1 所示,函数间的错误传播方式有以下 3 种:

(1) 函数 *A* 调用函数 *B*,函数 *A* 把错误的数据通过调用函数参数传递给函数 *B*;

(2) 函数 *A* 被函数 *B* 调用,函数 *A* 把错误的数据通过函数返回值传递给函数 *B*;

(3) 函数 *A* 将错误的数据写入全局变量,函数 *B* 读取错误的全局变量.

定义 1. 将完成对上述函数间交互数据写操作的指令称为接口指令.对应函数间错误传播方式,接口指令包括函数参数写指令、函数返回值写指令和全局变量写指令这 3 种类型.



(a) 函数参数 (b) 函数返回值 (c) 全局变量函数返回值

Fig.1 Fault propagation patterns between functions

图 1 函数间的错误传播方式

2.2 函数内部错误传播

函数内部错误传播是指函数内部软错误发生的指令到接口指令的传播过程.传播方式分为数据流错误和控制流错误.当发生数据流错误时,执行的指令与无软错误时相同,但指令所读写的寄存器和内存数据发生了错误.当发生控制流错误时,程序执行了与无软错误时不同的指令.

控制流错误一般是由于比较指令(cmp,test 等指令)发生错误导致的.比较指令会影响状态寄存器的标志位,而其后的条件转移指令根据标志位来选择究竟执行哪一个分支.因而,比较指令发生错误会导致条件转移指令执行与无软错误不同的指令.

定理 1. 接口指令发生数据流错误或控制流错误是导致 SDC 的必要条件.

证明:首先证明接口指令的存在性.假设在函数内部指令 I_k 发生的软错误最终导致 SDC,其传播过程中必须经过函数间错误传播.因为导致 SDC 一定要输出函数输出错误的内容,而发生软错误的指令 I_k 在输出函数外部,只有通过函数间的错误传播才能把错误传递到输出函数的参数.函数间的错误传播是由接口指令完成的,所以导致 SDC 的过程中执行过接口指令.

其次,用反证法证明导致 SDC 的过程一定存在发生数据流错误或控制流错误的接口指令.假设所有执行过的接口指令都是正确的,那么与 I_k 所在函数交互的函数没有受到错误的影响,得到的输出结果是正确的,不会产生 SDC.综上,导致 SDC 的执行过程存在发生数据流错误或控制流错误的接口指令. □

3 基于不变量的检测方法

基于不变量的检测方法通过判断不变量形式的断言是否满足来检测软错误.检测方法分为检测位置确定和断言生成两个部分.检测位置是断言放置的源代码位置.

检测位置是根据 SDC 错误传播分析确定的.根据定理 1,SDC 一定要通过接口指令的传播,而接口指令发生错误分为数据流错误和控制流错误两种情况.假设函数 f 有两个接口指令 I_c,I_b,I_c 的正常数据为 d_c,I_b 的正常数据为 d_b.无软错误时,函数执行接口指令 I_c,接口指令的数据为 d_c.当发生数据流错误时,程序依然执行了接口指令 I_c,只是数据发生了变化,变为 $d'_c(d'_c \neq d_c)$.由于 d'_c 偏离了正常的数据,检测位置确定在接口指令 I_c.当发生控制流错误时,执行了接口指令 I_b 而不是 I_c.接口指令 I_b 的数据 d'_b 有可能仍等于 d_b.在 I_b 处添加断言有较大可能检测不到错误,所以将检测位置设置在比较指令来及时发现错误的跳转.综上,为了覆盖两种情况,本文将比较指令和接口指令作为检测位置.

断言生成分为不变量提取和检测代价优化两个部分.不变量提取采用关系匹配的方式进行.在检测位置一般能够提取到大量的不变量,不变量的检测能力也有较大差异.本文通过定义渗透率来衡量不变量的检测能力.根据渗透率,挑选出检测能力较强的不变量作为最终的断言.第 3.1 节讨论不变量提取,第 3.2 节讨论检测代价优化.其中,检测代价优化对断言的检出率和检测代价影响较大,是本节讨论的重点.

3.1 不变量提取

定义 2. 将不变量定义为三元组(P,V,R).P 代表程序点(program point),用指令的静态地址表示;V 代表程序变量序偶;R 代表关系.例如不变量 $q_1=(p_1,v_1,r_1)$,$p_1=0x10$,$v_1=\langle tmp1,tmp2\rangle$,$r_1=\{\langle x,y\rangle|y=x+1\}$,$q_1$ 表示在程序点 $0x10$,程序中的变量序偶$\langle tmp1,tmp2\rangle \in r_1$,即 $tmp1$,$tmp2$ 满足 $tmp2=tmp1+1$.按照关系包含的变量数量,本文考虑的关系分为一元关系、二元关系、三元关系和数组这 4 种类型,其中的主要关系见表 3.其中,x,y,z 是变量,a,b,c 是计算得出的参数.

Table 3 Relations of invariant

表 3 不变量的关系

类型	表达式
一元关系	$x=a$, $x>a$, $x<a$, $a<x<b$, $x\%a=0$
二元关系	$y=ax+b$, $x<y$, $x\neq y$
三元关系	$z=ax+by+c$
数组	数组最大值、最小值为定值,元素按升序或降序排列

提取不变量就是确定程序点、变量序偶和关系的过程.程序点就是上述分析得到的检测位置.接下来,在每个程序点找出变量序偶满足的关系.提取不变量的过程主要分为 5 步.

(1) 为了防止关系是碰巧满足的,首先选取两组输入数据:一组作为提取组,用来提取不变量;一组作为验证组,用来验证提取的不变量.提取组和验证组不包含相同的输入数据.

(2) 以提取组为输入运行程序,得到程序的 trace 文件,trace 文件中包含每个程序点运行时刻变量的数值.

(3) 在每个程序点,将 trace 文件中变量序偶的数值依次代入表 3 所列的关系的约束条件中,验证关系是否成立.以一元关系为例,需要验证表 3 一元关系一行的 5 个关系,即

$$r_1=\{x|x=a\},r_2=\{x|x>a\},r_3=\{x|x<a\},r_4=\{x|a<x<b\},r_5=\{x|x\%a=0\}.$$

只要一组数值不满足关系 $r_i(1\leqslant i\leqslant 5)$,变量序偶就不满足关系 r_i.若所有数值都满足关系 r_i,则变量序偶满足关系 r_i,将该三元组列入待选的不变量集合.对于参数未定的关系,首先根据一组或几组数值计算参数,确定关系表达式,再进行验证.以关系 r_2 的验证过程为例,假设变量 x 在 trace 文件中的数值是 {1,3,5,7,2,18,25,37,40},首先计算表达式中的 a.以 $x_{\min}$ 表示 trace 文件中变量的最小值,则 $a=x_{\min}-1=0$.接着,将 trace 中变量 x 的数值代入 $r_2=\{x|x>0\}$中,所有数值均满足,因而 r_2 是满足的.

(4) 在待选的不变量集合中,根据不变量的可信度进行筛选.可信度主要和满足关系的 trace 数量有关.如果 trace 数量太少,不变量可能只是碰巧满足.假设一条 trace 碰巧满足关系 r 的概率为 p_r,n 条 trace 都

满足关系 r 的概率为 $P=1-p_r^n$.若 p_r=0.5,当 n=7 时,P>0.99.实际中,取 n=7 作为阈值,满足关系的 trace 条数小于 7 的不变量被删除.

(5) 最后,在验证组检验不变量是否满足.检验的方法是:将所有第(4)步中输出的不变量以断言的形式插入源程序中;然后,以验证组为输入运行程序.如果程序的断言报错,证明该不变量在验证组不满足,则将其删除.

3.2 检测代价优化

根据上述步骤得到的不变量通常数量很多.假设程序点有 n 个变量,即使只提取一元关系,也会提取出 5n 个不变量(一元关系有 5 种).当 n=5 时,最多提取到 25 个不变量.如果把所有的不变量都作为断言,则检测代价很高,因而,本文在同一检测位置中挑选出检测能力最强的不变量作为最终的断言.

为了衡量不变量的检测能力,本文定义了渗透率.渗透率是指关系所包含的变量发生软错误后,关系依然满足的概率.

定义 3. 关系 r 包含变量 $u_1,u_2,\ldots,u_i,\ldots,u_n$.假设 $u_i(i\in[1,n])$发生软错误变为 u_i',渗透率.

$$\gamma(r)\overset{\text{def}}{=}P(\langle u_1,u_2,...,u_i',...,u_n\rangle\in r).$$

例如,对于恒等关系 r_c={x|x=1},假设 x 发生软错误后变为 x',渗透率$\gamma(r_c)$=$P(x'\in r_c)$=$P(x'$=1).不变量的渗透率越低,发生软错误后,不变量不满足的概率越高,因而断言的检测能力越强.

下面介绍渗透率的计算方法.以无符号整型变量为例,无符号整型变量包含 32 位,数值范围为[0,2^{32}−1],共包含 2^{32} 个数值.以 $V(x)$表示变量 x 的取值集合,例如对于 r_c,$V(x)$={1}.本文假设 x 的取值服从 $V(x)$内的均匀分布.当计算渗透率时,样本空间Ω为 $V(x)$集合内所有数值的总位数,因而 $N(\Omega)$=|$V(x)$|×32.以下计算中,用 x_k 表示 x 第 k 位,用 x'表示发生软错误后的变量.x_k 取 0 或 1.若 x_k 发生软错误,当 x_k=0 时,x'=x+2^k;当 x_k=1 时,x'=x−2^k.

将关系分为等于关系(包含=,≠的关系)和范围关系(包含<,>的关系).范围关系的讨论较为复杂,本节先讨论等于关系,再讨论范围关系.

3.2.1 等于关系

本节讨论 $x=a$,$x\%a=0$,$x\neq y$ 的渗透率.

(1) $x=a$

$x'=x\pm 2^k\neq a$,所以 $x'=a$ 不可能满足,$\gamma(x=a)=P(x'=a)=0$.

同理,$\gamma(y=ax+b)=\gamma(z=ax+by+c)=0$.

(2) $x\%a=0$

$x'=x\pm 2^k$,要满足$(x\pm 2^k)\%a=0$,根据同余定理,得出 $2^k\%a=0$.分 $a\neq 2^q$ 和 $a=2^q(q\in\mathbf{Z},0\leqslant q\leqslant k)$两种情况来讨论:

a) 若 $a\neq 2^q$,$2^k\%a\neq 0$,则 $P(x'\%a=0|a\neq 2^q)=0$。

b) 若 $a=2^q$,要满足 $2^k\%a=2^k\%2^q=0$,则 $q\leqslant k\leqslant 31$.因而 $P(x'\%a=0\mid a=2^q)=\dfrac{32-q}{32}$.

综上, $\gamma(x\%a=0)=\begin{cases}0, & a\neq 2^q\\ \dfrac{32-q}{32}, & a=2^q\end{cases}$.

由于 $a=2^q$ 的概率很小,对于绝大部分的 a,$\gamma(x\%a=0)=0$.

(3) $x\neq y$

考虑与 $x'\neq y$ 互斥的事件 $x'=y$.只有 x 与 y 有一位不同并在该位发生翻转时,$x'=y$ 才会满足,因而,

$$\gamma(x\neq y)=1-P(x'=y)=1-\frac{C_{2^{32}}^1.32}{C_{2^{32}}^2.32}=1-4.7\times 10^{-10}.$$

3.2.2 范围关系

范围关系的渗透率与 x_k、关系类型(大于、小于)等因素有关.比如对于 $x<a$,当 x_k=1 时,x'=x−2^k,$x<a$ 依然满

足;而当 x_k=0 时,$x'=x+2^k$,$x<a$ 只有在 $x<a-2^k$ 的情况下才能满足.计算渗透率需要得出满足 x_k=1∧$x<a$ 和满足 x_k=0∧$x<a-2^k$ 的 x_k 个数.由此定义了函数 $f_0^k(w_1,w_2)$,$f_1^k(w_1,w_2)$,如公式(1)、公式(2)所示.

$$f_0^k(w_1,w_2)\overset{\text{def}}{=}\sum_{x=w_1}^{w_2-1}(1-x_k) \tag{1}$$

$$f_1^k(w_1,w_2)\overset{\text{def}}{=}\sum_{x=w_1}^{w_2-1}x_k \tag{2}$$

$f_0^k(w_1,w_2)$ 计算了从 w_1 到 w_2-1 的整数中,$(w_1,w_1+1,\ldots,w_2-1)$第 k 位是 0 的个数,$f_1^k(w_1,w_2)$ 计算了从 w_1 到 w_2-1 的整数中,第 k 位是 1 的个数.其中,$0\leqslant w_1<w_2\leqslant 2^{32}$,$0\leqslant k\leqslant 31$,$a,k\in\mathbf{Z}$.

易得 $f_0^k(w_1,w_2)=f_0^k(0,w_2)-f_0^k(0,w_1)$,$f_1^k(w_1,w_2)=w_2-w_1-f_0^k(0,w_2)+f_0^k(0,w_1)$,只要求出 $f_0^k(0,w_1)$,$f_0^k(0,w_2)$ 就能求出 $f_0^k(w_1,w_2)$,$f_1^k(w_1,w_2)$.$f_0^k(0,w_1)$,$f_0^k(0,w_2)$ 可以通过公式(3)计算.

$$f_0^k(0,w)=p_1+p_2+p_3 \tag{3}$$

$$p_1=\left\lfloor\frac{w}{2^{k+1}}\right\rfloor\cdot 2^k \tag{4}$$

$$p_2=w\%2^{k+1}-w\%2^k \tag{5}$$

$$p_3=\left(1-\frac{w\%2^{k+1}-w\%2^k}{2^k}\right)\cdot(w\%2^k) \tag{6}$$

证明:从 0 到 $w-1$ 的第 k 位是以 2^{k+1} 为周期循环出现的.以从 0 到 8 为例,表 4 显示了从 0 到 8 的低 4 位,第 0 位一列是以 2 为周期,第 1 位一列以 4 为周期,以此类推.将 2^{k+1} 周期以外的剩余部分称为剩余串.设剩余串的总长度为 m,剩余串中 0 的个数为 n.剩余串分为两种形式:

- 一种是以 0 串结尾,即 $\underbrace{00\ldots011\ldots1}_{2^{k+1}}\underbrace{00\ldots011\ldots1}_{2^{k+1}}\ldots\underbrace{00\ldots0}_{\leqslant 2^k}$,此时 $n=m$;
- 另一种是以 1 串结尾,即 $\underbrace{00\ldots011\ldots1}_{2^{k+1}}\underbrace{00\ldots011\ldots1}_{2^{k+1}}\ldots\underbrace{00\ldots0}_{2^k}\underbrace{11\ldots1}_{\leqslant 2^k}$,此时 $m=2^k+n$.

Table 4 Bit representation of 0~8

表 4 数值 0~8 的位表示

数值	位			
	3	2	1	0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

公式(3)中,p_1 等于以 2^{k+1} 为周期的 0 的个数.其中,$\left\lfloor\frac{w}{2^{k+1}}\right\rfloor$ 得出了周期个数,每个周期的 0 的个数为 2^k;p_2 和 p_3 合起来计算了剩余串中 0 的个数.接下来,在以 0 串结尾和以 1 串结尾两种情况下证明 $p_2+p_3=n$.

(1) 对于以 0 串结尾的情况,$n=m$.$w\%2^{k+1}=w\%2^k=m$ ∴$p_2=0$,$p_3=m$,$p_2+p_3=m=n$;

(2) 对于以 1 串结尾的情况,$n=2^k$.$w\%2^{k+1}=m$,$w\%2^k=m-2^k$ ∴$p_2=2^k$,$p_3=0$,$p_2+p_3=2^k=n$.

公式得证. □

利用所定义的函数 $f_0^k(w_1,w_2)$ 计算范围关系的渗透率.计算思路是分 x_k=0 和 x_k=1 两种情况进行分析,求出满足条件的 x_k 的个数,然后求出样本空间 Ω,得出渗透率.

(1) $x<a$

a) 当 x_k=1 时,由于 $x'=x-2^k<x<a$,因而 $x'<a$ 成立;

b) 当 x_k=0 时,$x'=x+2^k$.要满足 $x'<a$,则 $x<a-2^k$.即当 x 取 0,1,…,$a-2^k-1$ 时,$x'<a$ 成立.满足条件的 x_k 的个数:

$$N(x_k=0\wedge x<a-2^k)=\sum_{k=0}^{31}f_0^k(0,a-2^k).$$

样本空间Ω为符合 $x<a$ 的 x 的总位数:$|V(x)|=a$,$N(\Omega)=|V(x)|\times 32=32a$.

$$\text{综上, }\gamma(x<a)=P(x'<a)=\frac{N(x_k=1)+N(x_k=0\wedge x<a-2^k)}{N(\Omega)}=\frac{\sum_{k=0}^{31}f_1^k(0,a)+\sum_{k=0}^{31}f_0^k(0,a-2^k)}{32a}.$$

(2) $x>a$

a) 当 x_k=0 时,由于 $x'=x+2^k>x>a$,所以 $x'>a$ 成立;

b) 当 x_k=1 时,$x'=x-2^k$.若要满足 $x'>a$,则 $x>2^k+a$.

样本空间Ω为符合 $x>a$ 的 x 的总位数:$|V(x)|=2^{32}-a$,$N(\Omega)=32(2^{32}-a)$.

$$\text{综上, }\gamma(x>a)=P(x'>a)=\frac{\sum_{k=0}^{31}f_0^k(a+1,2^{32})+\sum_{k=0}^{31}f_1^k(a+2^k+1,2^{32})}{32(2^{32}-a)}.$$

(3) $a<x<b$

a) 当 x_k=1 时,由于 $x'=x-2^k<x<b$,要让 $a<x'<b$ 成立,只需 $a<x'$,即 $a+2^k<x$ 成立.

b) 当 x_k=0 时,$x'=x+2^k>a$.要让 $a<x'<b$ 成立,只需 $x'<b$,即 $x<b-2^k$ 成立.

样本空间Ω为符合 $a<x<b$ 的 x 的总位数:$|V(x)|=b-a-1$,$N(\Omega)=32(b-a-1)$.

$$\text{综上, }\gamma(a<x<b)=P(a<x'<b)=\frac{\sum_{k=0}^{31}f_0^k(a+2^k+1,b)+\sum_{k=0}^{31}f_1^k(a+1,b-2^k)}{32(b-a-1)}.$$

其他范围关系的计算方法类似,这里不再赘述.

4 系统实现

根据以上检测方法,本文实现了程序加固系统 Radish.Radish 通过添加基于不变量的断言对 c 程序源文件进行加固.Radish 的输入是 c 程序源文件,输出是加入不变量形式断言的 c 程序源文件.当加固后的源文件运行时,假如检测到不变量不满足会报错并终止运行,以防止错误继续传播.

Radish 包括 galar,matinv 和 addinv 这 3 个模块.galar 负责 trace 提取模块,matinv 进行关系匹配和不变量输出,addinv 计算渗透率并将不变量转化为断言.下面以 my.c 的例子介绍 Radish 的工作流程,如图 2 所示.

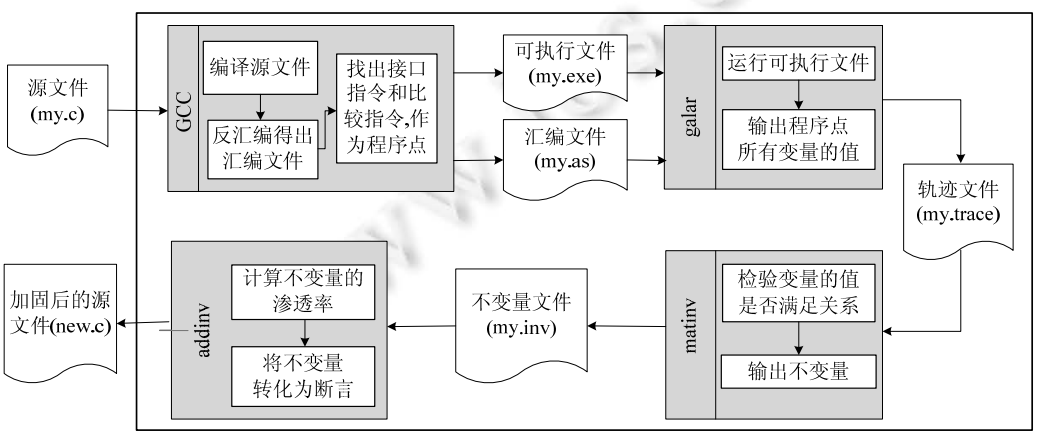


Fig.2 Work flow chart of Radish

图 2 Radish 工作流程图

(1) 通过 GCC 编译源文件 my.c,生成可执行文件 my.exe.通过 objdump 命令选项得到 my.c 的汇编文件 my.as.在 my.as 中寻找所有的接口指令和比较指令,将其静态地址作为 galar 的程序点.

(2) galar运行可执行文件my.exe,并在程序点位置记录下所有可读变量的值(包括全局变量以及作用域包含程序点的局部变量),输出到轨迹文件 my.trace.

(3) matinv 检验 my.trace 中变量的值是否满足关系,将不变量输出到不变量文件 my.inv 中.

(4) addinv 计算 my.inv 中不变量的渗透率.在每个程序点,addinv 将渗透率最低的不变量转化为 c 语言中的断言,并插入回原程序 my.c 相应位置.最终生成加固后的文件 new.c.

5 验证实验

本文通过两轮错误注入实验来验证断言的有效性:第 1 轮错误注入实验针对原始程序,第 2 轮针对添加断言后的程序.将两轮实验的结果进行对比,计算出断言的检出率和检测代价.检测代价通过添加断言后执行的指令增量与原指令总量之比来衡量.

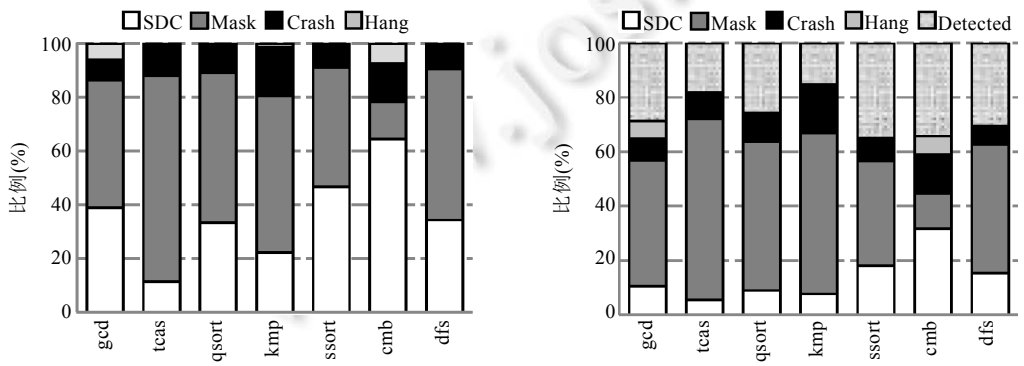
本文的实验平台是 Ubuntu10.04,采用工具 Pin[18]进行错误注入.Pin 是针对 IA-32 和 x86-64 指令集的动态二进制插桩工具,能够对任意指令插入用户自定义的分析代码.

实验中,对每一条指令都进行了错误注入.错误注入通过 Pin 改变指令读写的寄存器或内存的一位来实现.错误注入根据注入计划表来依次进行.注入计划表包含注入指令的静态地址、实例序号、注入硬件位置和注入位.静态地址是指令相对于代码段开始位置的偏移量.实例序号是被注入错误的实例在所有相同静态地址运行实例中的序号.注入硬件位置是注入错误的具体位置.注入位表明对硬件位置的第几位进行注入.

测试程序是一些常用的算法和数值计算程序,包括 qsort(快速排序)、kmp(字符串匹配)、gcd(求最大公约数)、cmb(求组合数)、dfs(深度优先搜索)、ssort(希尔排序)、tcas(飞行安全判定).当注入错误后的结果与无错误注入时的结果不同时,就认为发生了 SDC.经过测算,测试程序执行时间最长不超过 0.9s.当执行时间超过 30s 时,就认为发生了挂起,结束程序的运行.提取不变量时,提取组和验证组都包含 100 组输入数据.测试程序原始提取的不变量数量为 2 734 个,经过渗透率计算后筛选至 41 个.筛选后的不变量占原有的不变量的 1.5%,极大地减少了插入的断言数量,从而降低了检测代价.

5.1 原始程序的错误注入结果

对原始程序的错误注入结果如图 3(a)所示,测试程序平均的 SDC 比例是 36%.SDC 比例的范围是 11%~64%,不同程序的 SDC 比例的差异度较大.SDC 比例与程序采用的数据结构有关,比如,cmb 的 SDC 比例最高,因为其采用递归的方式计算组合数.只要一次递归时发生错误,组合数就会错误.



(a) 原始程序错误注入结果 (b) 加固后程序错误注入结果

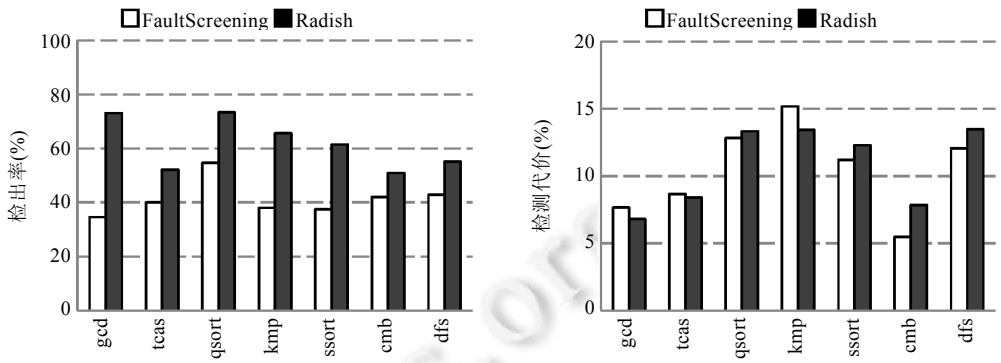
Fig.3 Experimental results of fault injections

图 3 错误注入实验结果

5.2 加固后的错误注入结果

对加固后的程序的错误注入结果如图 3(b)所示.SDC 的平均比例是 14%,cmb 的 SDC 比例依然最高.SDC

的平均检出率为 62%,平均检测代价为 11%.检测代价的范围是 7%~13%.检出率和检测代价如图 4(a)、图 4(b)所示.由于断言一般都是简单的语句,编译后单次执行增加的指令在 10 条左右.检测代价主要与断言的执行次数有关.如果断言位于循环体内部,则检测代价会比较高.例如,kmp 的断言多数位于 while 循环中,其检测代价较高.



(a) SDC 检出率 (b) 检测代价

Fig.4 SDC coverage rate and detection overhead obtained by FaultScreening and Radish

图 4 FaultScreening 和 Radish 的 SDC 检出率、检出代价比较

虽然本文的断言是针对 SDC 进行设计的,但对其他 3 种故障类型也具备一定的检测能力.如图 5 所示,屏蔽、崩溃和挂起的平均检出率分别为 8%,8%和 42%.屏蔽和崩溃的检出率远低于 SDC.屏蔽的检出率低是因为程序的运行没有受到软错误的影响或者影响很小,不变量一般仍然满足.崩溃的检出率较低是因为发生软错误后很快就会导致崩溃.经过统计,57%导致崩溃的情况只执行不超过 10 条指令,程序还没有执行断言就已经结束运行了.今后将在 Radish 系统基础上集成基于现象捕获方法的检测器,以对其他故障类型进行检测.

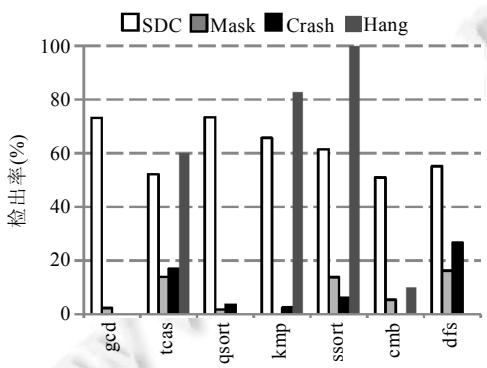


Fig.5 Coverage rate of four error categories

图 5 4 种故障类型的检出率

5.3 与FaultScreening比较

图 4 比较了本文的 Radish 与 FaultScreening[4]对 SDC 的检出率和检测代价.FaultScreening 通过判断变量的值是否位于合法区间来检测软错误,区间的生成过程见第 1.1 节.实验中,FaultScreening 的检测位置与本文相同,但断言不同.FaultScreening 的平均检出率为 41%,比本文低 21%.主要原因是:FaultScreening 的断言只包含范围关系,检测能力不如本文的断言.如图 6 所示,以 dfs 程序某一处的断言为例,FaultScreening 的断言是 0<*stackPtr*<7,本文的断言是 *stackPtr*–*x*–*y*=0.从渗透率来看,0<*stackPtr*<7 的渗透率为 0.0625,*stackPtr*–*x*–*y*=0 的渗透率为 0.

stackPtr−*x*−*y*=0 的渗透率低于 0<*stackPtr*<7,所以检测能力更强.当发生软错误后,如果 *stackPtr* 仍在 0~7 范围内,则通过判断 0<*stackPtr*<7 是检测不到的;而通过判断 *stackPtr*−*x*−*y*=0 不仅能够检测 *stackPtr* 发生的软错误,还可以检测到 *x* 和 *y* 发生的软错误.

```
If (y<size−1 && maze[x][y+1])
{
        ...
}
If (x<size−1 && maze[x+1][y])
{
        ...
}
stackPtr−−;
assert(stackPtr>0 && stackPtr<7);
```

(a) FaultScreening

```
If (y<size−1 && maze[x][y+1])
{
        ...
}
If (x<size−1 && maze[x+1][y])
{
        ...
}
stackPtr−−;
assert(stackPtr−x−y==0);
```

(b) Radish

Fig.6 Asserts in dfs generated by FaultScreening and Radish

图 6 FaultScreening 和 Radish 对 dfs 程序生成的断言

FaultScreening 的检测代价是 10%,与本文相差不到 1%.检测代价接近是因为 FaultScreening 和本文采用了相同的检测位置,并且断言的指令条数接近.

5.4 小 结

综上所述,本文的平均检出率为 62%,平均检测代价为 11%.以较低的检测代价获取了较高的检出率.与程序级断言的典型工作 FaultScreening 相比,检出代价基本相同,检出率却提高了 21%.

6 结束语

针对 SDC 难以检测的问题,本文研究了基于不变量的检测方法.本文的检测方法并不需要了解程序的逻辑,因而具备很好的通用性和扩展性.在保证较低的检测代价前提下,实现了较高的检出率.下一步需要改进渗透率的计算.现阶段对所有变量是平等对待的.然而变量对于错误传播的重要程度是不同的,今后可以对变量的重要程度进行建模和分析,以进一步提高检出率.

References:

[1] Walters JP, Zick KM, French M. A practical characterization of a NASA SpaceCube application through fault emulation and laser testing. In: Proc. of the Dependable Systems and Networks. Washington: IEEE Computer Society, 2013. 1−8. [doi: 10.1109/DSN.2013.6575354]

[2] Xing KF. Single event effect detection and mitigation techniques for spaceborne signal processing platform [Ph.D. Thesis]. Changsha: National University of Defense Technology, 2011 (in Chinese with English abstract).

[3] Xu JJ. Research on compile techniques of fault tolerance for soft errors [Ph.D. Thesis]. Changsha: National University of Defense Technology, 2010 (in Chinese with English abstract).

[4] Racunas P, Constantinides K, Manne S. Perturbation-Based fault screening. In: Proc. of the Int'l Conf. on High-Performance Computer Architecture. Washington: IEEE Computer Society, 2007. 169−180. [doi: 10.1109/HPCA.2007.346195]

[5] Gu W, Kalbarczyk Z, Iyer RK. Characterization of Linux kernel behavior under errors. In: Proc. of the Dependable Systems and Networks. Washington: IEEE Computer Society, 2003. 22−25. [doi: 10.1109/DSN.2003.1209956]

[6] Shafique M, Rehman S, Aceituno PV. Exploiting program-level masking and error propagation for constrained reliability optimization. In: Proc. of the 50th Annual Design Automation Conf. New York: ACM Press, 2013. 17. [doi: 10.1145/2463209.2488755]

[7] Rehman S, Shafique M, Aceituno PV. Leveraging variable function resilience for selective software reliability on unreliable hardware. In: Proc. of the Conf. on Design, Automation and Test in Europe. Washington: IEEE Computer Society, 2013. 1759−1764. [doi: 10.7873/DATE.2013.354]

[8] Wang NJ, Patel SJ. ReStore: Symptom-Based soft error detection in microprocessors. IEEE Trans. on Dependable and Secure Computing, 2006,3(3):188−201. [doi: 10.1109/TDSC.2006.40]

[9] Li ML, Ramachandran P, Sahoo SK. Understanding the propagation of hard errors to software and implications for resilient system design. ACM SIGARCH Computer Architecture News, 2008,36(1):265−276. [doi: 10.1145/1346281.1346315]

[10] Sahoo SK, Li ML, Ramachandran P. Using likely program invariants to detect hardware errors. In: Proc. of the Dependable Systems and Networks with FTCS and DCC. Washington: IEEE Computer Society, 2008. 70−79. [doi: 10.1109/DSN.2008.4630072]

[11] Pattabiraman K, Saggese GP, Chen D. Dynamic derivation of application-specific error detectors and their implementation in hardware. In: Proc. of the Dependable Computing Conf. Washington: IEEE Computer Society, 2006. 97−108. [doi: 10.1109/EDCC.2006.9]

[12] Hari S, Adve S, Naeimi H. Low-Cost program-level detectors for reducing silent data corruptions. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. Washington: IEEE Computer Society, 2012. 1−12. [doi: 10.1109/DSN.2012.6263960]

[13] Matin H. Executable assertions for detecting data errors in embedded control systems. In: Proc. of the 30th Int'l Conf. on Dependable Systems and Networks. Washington: IEEE Computer Society, 2000. 24−36. [doi: 10.1109/ICDSN.2000.857510]

[14] Ernst MD, Perkins JH, Guo PJ. The Daikon system for dynamic detection of likely invariants. Science of Computer Programming, 2007,69(1):35−45. [doi: 10.1016/j.scico.2007.01.015]

[15] Sudheendra H, Monica S. Tracking down software bugs using automatic anomaly detection. In: Proc. of the Int'l Conf. on Software Engineering. New York: ACM Press, 2002. 291−301. [doi: 10.1145/581339.581377]

[16] Boshernitsan M, Doong R, Savoia A. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In: Proc. of the Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2006. 169−180. [doi: 10.1145/1146238.1146258]

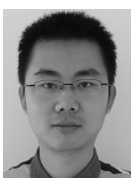
[17] Xu X, Li ML. Understanding soft error propagation using efficient vulnerability-driven fault injection. In: Proc. of the Dependable Systems and Networks. Washington: IEEE Computer Society, 2012. 1−12. [doi: 10.1109/DSN.2012.6263923]

[18] Luk CK, Cohn R, Muth R. Pin: Building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices, 2005,40(6):190−200. [doi: 10.1145/1065010.1065034]

附中文参考文献:

[2] 邢克飞.星载信号处理平台单粒子效应检测与加固技术研究[博士学位论文].长沙:国防科学技术大学,2007.

[3] 徐建军.面向寄存器软错误的容错编译技术研究[博士学位论文].长沙:国防科学技术大学,2010.



马骏驰(1988－),男,陕西西安人,博士生,主要研究领域为软错误防护,软件可靠性.

汪芸(1967－),女,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布计算,容错计算,传感器网络.