计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20148367
53(9):1943-1952,2016

基于错误传播分析的 SDC 脆弱指令识别方法

马骏驰[1,2] 汪 芸[1,2] 蔡震波[3] 张庆祥[3] 王 颖[3] 胡 诚[1,2]
[1](东南大学计算机科学与工程学院 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) 南京 211189)
[3](中国空间技术研究院总体部 北京 100094)
(jcma@seu.edu.cn)

An Approach for Identifying SDC-Causing Instructions by Fault Propagation Analysis

Ma Junchi[1,2], Wang Yun[1,2], Cai Zhenbo[3], Zhang Qingxiang[3], Wang Ying[3], and Hu Cheng[1,2]
[1](*School of Computer Science & Engineering, Southeast University, Nanjing* 211189)
[2](*Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education, Nanjing* 211189)
[3](*Institute of Spacecraft System Engineering, China Academy of Space Technology, Beijing* 100094)

Abstract Single event upset (SEU) is caused by external radiation in outer space and it has a great influence on computing reliability of space devices. As process technology scales, space devices become more susceptible to SEU. SEU could result in silent data corruption (SDC), which means wrong outcomes of a program without any crash detected. SDC may lead to serious failure and hence cannot be ignored. As SDC-causing fault always propagates silently, it is very difficult to detect SDC. To develop SDC detectors, SDC-causing instructions of a program should be identified as the first step. However, this step usually needs a huge number of fault injections, which is extremely time-consuming and not achievable for most applications. In this paper, we build data dependence graph (DDG) to capture the dependencies among the values of instructions. Then the inter-function and intra-function propagation that leads to SDC is analyzed and the sufficient condition of SDC-causing instructions is demonstrated. Further, we propose a novel method of identifying SDC-causing instructions. Taking advantage of the trace files of injection, our method can detect underlying SDC-causing instructions without any expensive computations. Validation efforts show that our method yields high accuracy and coverage rate with a great reduction of injection cost.

Key words single event upset (SEU); soft error; SDC-causing instruction; fault injection; fault propagation

摘 要 单粒子软错误是高辐照空间环境下影响计算可靠性的主要因素.随着芯片晶体管数的快速增长,单粒子软错误的威胁日益严重.结果错误(silent data corruption, SDC)是单粒子软错误造成的一种故障类型.由于 SDC 是隐蔽传播的,SDC 的检测是单粒子软错误防护的难点.寻找 SDC 脆弱指令是目前检测 SDC 的重要途径.现有方法需要进行巨量的错误注入,时间代价巨大.首先根据数据关联图建立了指令的数据依赖关系,研究了函数间和函数内部错误传播过程;进而推导出判定 SDC 脆弱指令的

收稿日期:2014-12-12;**修回日期**:2015-12-14

充分条件,提出了 SDC 脆弱指令识别方法,该方法在错误注入中依据充分条件推测潜在的 SDC 脆弱指令.实验表明,在保证较高准确率和覆盖率的前提下,时间代价显著减少.

关键词 单粒子翻转;单粒子软错误;SDC 脆弱指令;错误注入;错误传播

中图法分类号 TP302.8

单粒子翻转(single event upset, SEU)是由宇宙中的质子、电子、重离子等高能粒子轰击半导体器件造成 PN 结瞬间充放电现象.单粒子翻转引起的半导体电路瞬态故障称为单粒子软错误[1-3](以下简称软错误).软错误虽然不会损坏内部硬件电路,但是有可能影响程序的正常运行,甚至导致卫星运行异常或者失控.

由于单个器件的软错误率基本保持稳定,因而处理器整体软错误率主要与集成度相关.在集成度以摩尔定律增长的趋势下,处理器的整体软错误率也会以摩尔定律增长[4],未来对于软错误的检测与加固技术的需求变得日益迫切.

软错误引起的故障类型大致可以分为 4 种[5]:屏蔽(benign)、崩溃(crash)、挂起(hang)和结果错误(silent data corruption, SDC),如表 1 所示.导致 SDC 的运行过程不会抛出异常,因此 SDC 的发生最为隐蔽.一旦发生 SDC,如果不能有效地检测,可能导致严重的后果.

Table 1 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

Category	Definition	Complete Execution	Right Output
Benign	The execution gets correct output without any symptoms.	Yes	Right
Crash	The execution is stopped abnormally.	No	No Output
Hang	Resource is exhausted but the execution can't be finished.	No	No Output
SDC	The program generates erroneous output.	Yes	Wrong

现有的软错误检测方法主要基于症状检测(symptom-based detector)[6-8].症状是指系统的一些异常特征,包括分支预测失效、cache 命中率低等.当检测器捕捉到异常特征时,就认为发生了软错误.此方法对软错误的检测率高、代价较低,但是不能检测 SDC,因为 SDC 是隐蔽传播的,不会出现常见的异常特征.

为了弥补基于症状检测方法的缺陷,近年来出现了针对 SDC 的软错误检测方法,主要包括指令级冗余和程序级断言.指令级冗余对容易受到软错误干扰的指令进行时间或空间的冗余设计[9-10];程序级断言通过对程序正常运行时为真的条件进行判断来检测软错误[11-13].2 种方法都需要针对选定的指令添加检测代码.执行检测代码会导致额外的时间代价,并且选定的指令越多,代价越高昂.为了减小时间代价,2 种方法都重点针对 SDC 脆弱指令(SDC-causing instruction)进行部署.如果指令读写的寄存器或内存发生软错误会导致 SDC,该指令称为 SDC 脆弱指令.

SDC 脆弱指令一般由软件实现的错误注入(fault injection)得出[14].错误注入通过修改寄存器和内存的二进制位来模拟软错误(二进制位为 0 时,将其改为 1;二进制位为 1 时,将其改为 0).注入错误数是修改二进制位的总次数,错误注入实验的时间代价主要与注入错误数相关.由于每条指令都需要进行错误注入,即使规模很小的程序也有巨量的注入错误数,时间代价巨大.例如,对目的操作数(32 位)进行错误注入,当程序含有 10 000 条指令时,注入错误数至少为 320 000,整个错误注入的时间代价至少为原程序单次运行的 320 000 倍.对于一般的计算设备,该时间代价难以接受.

本文提出了一种高效的 SDC 脆弱指令识别方法.该方法根据已执行的错误注入实验的信息动态推测潜在的 SDC 脆弱指令,由此减少未进行的错误注入.本文的主要贡献有:

1) 基于数据关联图(data dependence graph, DDG)建立了指令的数据依赖关系,分析了导致 SDC 的错误在函数间和函数内部的传播过程.指出关键指令(即写函数间交互数据的指令)在导致 SDC 的错误传播中的作用,由此提出并证明了判定非关键指令是 SDC 脆弱指令的充分条件.

2) 基于 SDC 脆弱指令的判定条件,提出了 SDC 脆弱指令识别方法.该方法可以在注入过程中动态推断 SDC 脆弱指令,从而有效减少注入错误数和时间代价.实验结果表明,注入错误数比现有工作 Relyzer[15] 低 25.6%,时间代价减少 27.9%.

1 相关工作

根据是否进行错误注入实验,可将现有工作中识别 SDC 脆弱指令的方法分为静态注入方法和静态分析方法 2 类,以下分类进行介绍.

1.1 静态注入方法

静态注入方法首先将需要错误注入的指令列入注入计划表(injection map),然后依次进行错误注入.在错误注入过程中,注入计划表不会发生变化.静态注入方法通过选择性的错误注入缓解了原有错误注入实验代价过高的问题.

Relyzer[15] 压缩了等价类的错误注入.等价类是指每个基本块(basic block)都相同的控制流实例组成的集合,对等价类的不同指令实例进行错误注入得到的结果是相似的,因此在等价类中只选择一个代表进行错误注入.

SmartInjector[16] 补充了 Relyzer 的方法,认为具备相同数据传播模式的数据流的注入结果是相似的,将其归为同一个等价类,并只选择一个指令实例进行错误注入.

CriticalFault[17] 通过指令级脆弱性分析来减少注入错误数.指令级脆弱性分析能够找出非敏感位.在非敏感位发生的软错误不会对程序的运行产生影响,也不会产生 SDC,因而 CriticalFault 排除掉了这些非敏感位的错误注入.

1.2 静态分析方法

静态分析方法不进行错误注入实验,而是通过分析程序特征得到 SDC 脆弱指令.

Shoestring[10] 认为所有影响全局内存或函数参数的指令都是 SDC 脆弱指令.此判定方法容易实施,但由于判定条件只考虑指令的目的操作数,而不考虑程序逻辑等其他因素,因此准确率较低.

SymPLFIED[18] 通过符号执行模拟错误传播过程来识别 SDC 脆弱指令.由于符号执行穷举了所有错误传播路径,因此不会出现漏判.但其模拟的某些错误在现实中不会发生,所以准确率较低,并且符号执行导致了状态爆炸,使得时间和空间代价极大.

综上所述,静态注入方法的优点是得到的 SDC 脆弱指令都是准确的,但错误注入导致代价较高;静态分析方法的实现简单,但准确率较低.

为了在保证高准确率情况下降低注入代价,本文提出一种新的动态注入方法.整体来看,前述静态注入方法的主要思路是在错误注入实验开始前对冗余的错误注入进行压缩;而本文是在错误注入实验开始后,根据已执行的错误注入的信息动态推测潜在的 SDC 脆弱指令,由此调整注入计划表来减少未进行的错误注入.需要指出的是,由于 2 种思路并不冲突,本文的识别方法可以与前述静态注入方法联合使用(本文中联合使用了典型的静态注入方法 Relyzer),从而进一步降低注入代价.

2 模型假设及相关概念

本文主要考虑单粒子一位翻转的情况.通过改变指令读写的寄存器或内存的一个二进制位来模拟单粒子一位翻转.

本文只研究应用程序范围内可能导致 SDC 的指令,因此不考虑在系统库函数的指令.浮点运算指令的工作方式与普通运算指令不同,本文不考虑浮点运算指令.因为指令操作码发生错误一般会导致崩溃[17],所以本文只考虑指令操作数的错误,后文中指令发生错误特指操作数发生错误.

应用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.由于输出函数是系统库函数,基于前述假设,输出函数的错误都是从其他函数传递进去的.

本文采用数据关联图[19] 对错误传播过程进行分析.数据关联图 $G(V,E)$ 是根据程序指令的读写依赖建立的有向图,其中节点集 V 代表程序运行中执行的指令,而边集 E 代表指令的数据依赖关系.例如,节点 u 读入了节点 v 写的数据,就会产生一条由 v 指向 u 的边.$\forall v_i, v_j \in V$,若 v_i 可达 v_j,则称 v_i 为 v_j 的祖先节点,v_j 为 v_i 的子孙节点.根据数据关联图能够找出错误的传入路径.在 3.3 节将通过数据关联图来分析错误传播过程.

以求和程序 *sum* 生成的数据关联图为例.*sum* 程序的代码如下所示,输入是变量 *size*,计算了从 0 到 *size*−1 的总和,并将其作为输出返回.

```
int sum(int size){
  k=0;
```

```
    for (int i=0;i<size;i++)
    {
      k=k+i;
    }
    return k;
}
```

图 1 表示了 *sum* 生成的数据关联图,表 2 显示了 *sum* 编译后的指令及其对应数据关联图中的节点. 图 1 中,节点按照指令所写变量排列. 比如变量 k 所在列的节点都是对 k 进行写操作的指令. 以节点 7 为例,节点 7 对应的指令为 add dword ptr [esp+0x24],eax,读取寄存器 eax 和内存[esp+0x24],写内存[esp+0x24]. [esp+0x24]代表寄存器 esp 与常量 0x24 相加后所对应内存地址的值. 节点 7 的父节点是上一次写 eax 的节点 6 和上一次写[esp+0x24]的节点 1,其子节点是下一次读[esp+0x24]的节点 13.

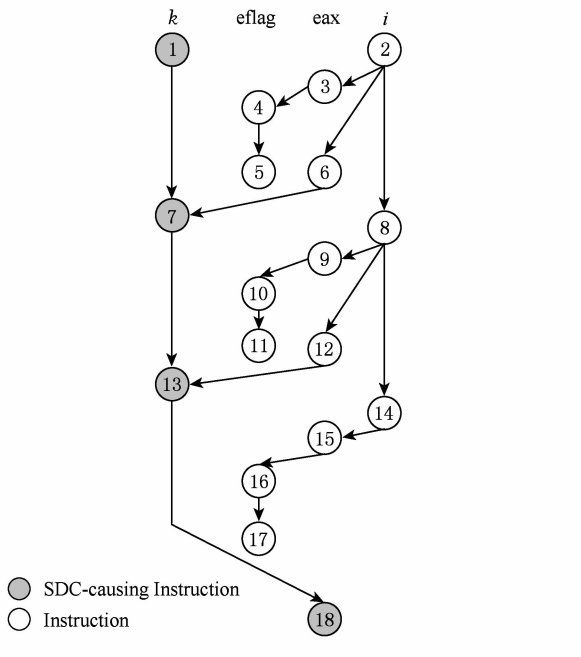

Fig. 1 The DDG of the program of *sum*.

图 1 求和程序 *sum* 的数据关联图

Table 2 The Instructions of the Program of *sum* and Their Corresponding Nodes in the DDG

表 2 求和程序 *sum* 的指令及其所对应的节点

Source Code	Instruction	Read	Write	Nodes in The DDG
$k=0$	mov dword ptr [esp+0x24], 0x0		[esp+0x24]	1
$i=0$	mov dword ptr [esp+0x20], 0x0		[esp+0x20]	2
$i<size$	mov eax, dword ptr [esp+0x20]	[esp+0x20]	eax	3, 9, 15
	cmp eax, dword ptr [esp−0x8]	eax,[esp+0x1c]	eflag	4, 10, 16
	jl 0x80486da	eflag	eip	5, 11, 17
$k=k+i$	mov eax, dword ptr [esp+0x20]	[esp+0x20]	eax	6, 12
	add dword ptr [esp+0x24], eax	[esp+0x24],eax	[esp+0x24]	7, 13
i++	add dword ptr [esp+0x20], 0x1	[esp+0x20]	[esp+0x20]	8, 14
return k	mov eax, dword ptr [esp+0x24]	[esp+0x24]	eax	18

3 导致 SDC 的错误传播分析

基于第 2 节的模型假设,本节讨论导致 SDC 的错误传播过程的特征. 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者输出内容的地址,所以导致 SDC 的直接原因是输出函数的参数发生了错误. 本文以函数为单元分析错误传播. 在传播到输出函数参数前,错误首先会在应用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点,在 3.1 节和 3.2 节分别对其进行分析. 以此为基础,在 3.3 节得出非关键指令的 SDC 脆弱性的判定定理.

3.1 函数间错误传播

函数间错误传播是指错误从函数内部指令读写的数据传播到函数间交互数据. 函数间交互数据可以被其他函数读取,错误继而传入了读取交互数据的函数. 假设有 2 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. 函数间交互数据指的是函数 A 写、函数 B 读的数据. 如图 2 所示,函数间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 读取全局变量.

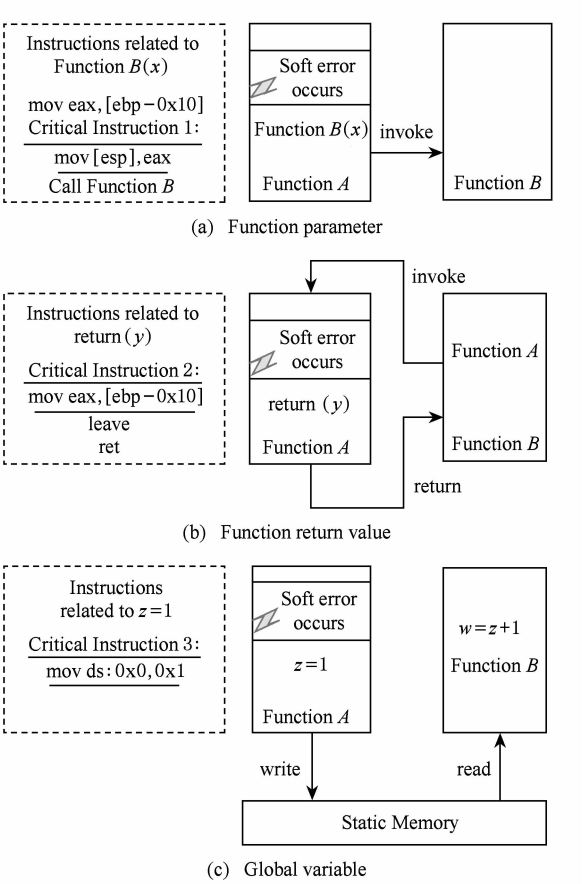

Fig. 2 The inter-function fault propagation

图 2 函数间的错误传播方式

将完成对上述函数间交互数据写操作的指令定义为关键指令,其余指令定义为非关键指令. 图 2 虚线框中的关键指令以直线划出. 对应函数间错误传播方式,关键指令包括以下 3 种类型:

1) 函数参数写指令. 该指令是调用者函数将参数写入栈的指令. 函数参数写指令执行后,被调用函数弹栈以取得参数. 如图 2(a)中的关键指令 1 将变量 x 写入栈地址[esp]中,被调用函数 B 读取[esp]的值来获取参数 x.

2) 函数返回值写指令. 该指令是被调用函数将函数返回值写入 eax 的指令. 函数返回值写指令执行后,调用者函数通过读取寄存器 eax 来获取返回值. 如图 2(b)中的关键指令 2 将变量 y 的值写入 eax,函数 B 通过读取 eax 来获取返回值.

3) 全局变量写指令. 该指令是指修改全局变量的指令. 图 2(c)中的关键指令 3 是全局变量写指令,它将 1 写入静态区中存放全局变量 z 的地址中.

定理 1. 关键指令发生错误是导致 SDC 的必要条件.

证明.

1) 证明关键指令的存在性,即导致 SDC 的过程中必然执行过关键指令. 假设在函数内部的非关键指令 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关键指令完成的,所以导致 SDC 的过程中执行过关键指令.

2) 用反证法证明导致 SDC 的过程一定存在发生错误的关键指令. 假设所有执行过的关键指令都是正确的,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错误的关键指令. 证毕.

3.2 函数内部错误传播

函数内部错误传播是指仅在某一函数内部进行读写的数据之间的传播. 函数内部错误传播是由非关键指令完成的. 包括以下 2 种方式:

1) 计算传播

计算传播是指在指令读入数据发生错误的情况下,造成了错误的写入数据. 读入数据包括源操作数、基址、状态寄存器等. 写入数据包括目的操作数、状态寄存器等. 当一条指令有多个读入数据时,任一读入数据的错误都有可能造成目的操作数的错误. 比如,指令 mov eax,DWORD PTR[ebp]需要读入的数据有 ebp 和 [ebp],不管哪一个存在错误,都有可能导致 eax 的错误.

2) 控制流传播

控制流传播是由于比较指令(cmp,test 等指令)发生错误导致程序执行错误的分支. 一般地,比较指令后的第一条指令是条件转移指令(例如 jz 指令). 比较指令影响状态寄存器的标志位,而条件转移指令根据标志位来选择究竟执行哪一个分支. 当比较指令的读入数据发生错误时,可能会导致执行错误的分支.

3.3 非关键指令的 SDC 脆弱性的判定

根据定理 1,非关键指令的错误要导致 SDC,必须要经过关键指令的传播. 进一步地,如果非关键指令的错误造成了关键指令的错误,由于两者存在的对应关系,可以通过分析关键指令错误的影响来

确定非关键指令错误的影响. 由上面的分析,本文得到了判定非关键指令的 SDC 脆弱性的充分条件.

定理 2. 对于非关键指令 I_{nc},若在同一函数中存在同时满足以下条件的关键指令 I_c,则 I_{nc} 是 SDC 脆弱指令.

1) I_{nc} 是 I_c 的祖先节点;

2) 当在 I_{nc} 发生错误时,I_c 是错误的,且是唯一错误的关键指令;

3) I_c 是 SDC 脆弱指令.

证明. 由条件 1 和条件 2,I_c 的错误是由 I_{nc} 传播的. I_c 是唯一错误的关键指令,其余关键指令是正确的,因而 I_{nc} 的错误对于其他函数的影响是通过 I_c 传递出去的,在 I_{nc} 发生的错误对其他函数的影响相当于在 I_c 发生错误对其他函数的影响. 由条件 3,根据 SDC 脆弱指令的定义,在 I_c 发生错误会产生 SDC,那么在 I_{nc} 发生错误也会产生 SDC,所以 I_{nc} 是 SDC 脆弱指令. 证毕.

4 SDC 脆弱指令识别方法

根据 3.3 节 SDC 脆弱性的判定条件,本文设计了 SDC 脆弱指令识别方法,分别对关键指令和非关键指令进行处理. 关键指令中的 SDC 脆弱指令是通过错误注入实验得到的,非关键指令中的 SDC 脆弱指令是通过错误注入实验和推断算法得到的. 由于非关键指令占注入错误数的绝大多数,因而对非关键指令的处理是识别方法的重点. 识别方法的步骤是:

1) 根据定义找出关键指令集合 Γ.

2) 通过错误注入实验判定关键指令集合 Γ 中的指令的 SDC 脆弱性. 如果错误注入得到的结果是 SDC,则将该关键指令加入 SDC 脆弱指令集合 Γ_{SDC}.

3) 识别非关键指令中的 SDC 脆弱指令. 首先使用 Relyzer 等价类注入的方法对非关键指令进行采样,将指令实例样本添加进注入计划表;然后依次进行错误注入,当得到的结果是 SDC 则运行推断算法,将推测得到的 SDC 脆弱指令从注入计划表中删去.

其中,步骤 3 的推断算法根据错误注入实验的数据对非关键指令的 SDC 脆弱性进行分析和推理.

推断算法的输入是错误注入实验中指令写入数据的记录、无错运行时指令写入数据的记录、关键指令中的 SDC 脆弱指令集合 Γ_{SDC},输出是推测出的 SDC 脆弱指令集合 S. 推断算法分为 3 个阶段,如图 3 所示. 其中,p 是指向错误注入实验的指令记录的指针,p' 是指向无错运行时指令记录的指针,$^*p$ 代表 p 所指向的指令,$p.O_d$ 代表 p 所指向指令记录的写入数据.

1) 比较阶段. 将 $p.O_d$ 与对应的 $p'.O_d$ 进行比较,若写入数据是不同的,则判定指令 $^*p$ 发生了错误. 如果 $^*p$ 属于 SDC 脆弱指令集合 Γ_{SDC},则进入推测阶段,否则进入结束阶段.

直接比较写入数据是地址的指令可能会造成误判. 由于操作系统的内存分配机制,2 次运行过程一般会分配不同的地址空间,因而写入数据是地址的指令会被认为是错误的. 本文通过错误传播分析来判断地址是否是错误的,规定当指令的读入数据是错误的、写入数据是地址时,该地址是错误的;否则是正确的.

2) 推测阶段. 深度优先搜索 $^*p$ 的祖先节点,若祖先节点是正确的,则结束该分支的搜索;否则,如果祖先节点只有一个错误的关键指令子孙节点

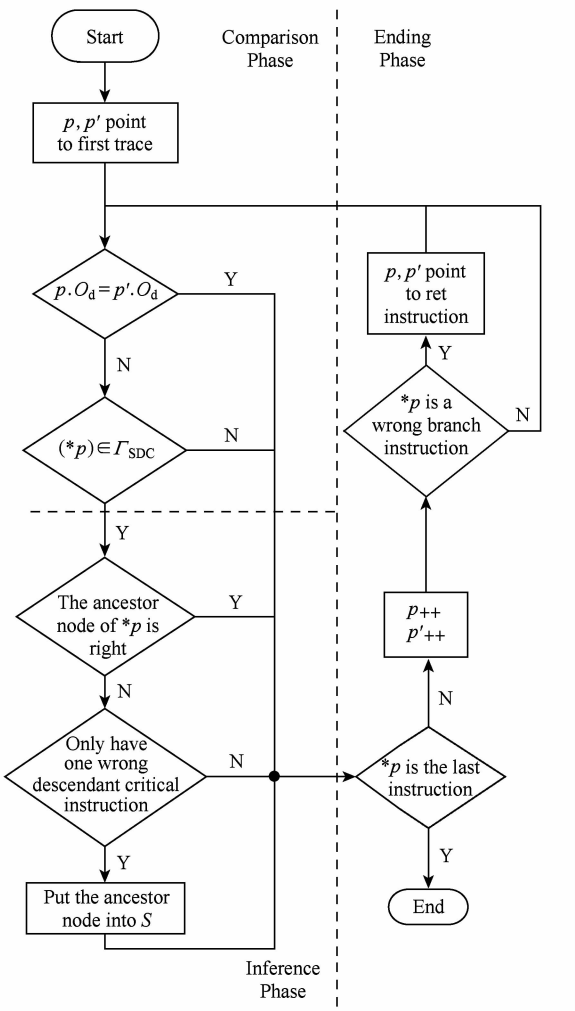

Fig. 3 The flow chart of inference algorithm.

图 3 推断算法的流程图

(即 $^*p$),则将该祖先节点加入推测出的 SDC 脆弱指令集合 S.

3) 结束阶段. p 和 p' 转到下一条指令记录. 如果 $^*p$ 是条件跳转指令,判断是否与无错运行时选择了同一分支,若选择了不同分支,则发生了控制流传播. p 和 p' 指向同一指令对应的实例才能进行比较,所以为了对齐 p 和 p',p 和 p' 指向 $^*p$ 所在函数的 ret 指令(ret 指令一般是函数最后一条指令). 如果 $^*p$ 已经是最后一条指令,则结束.

以图 1 为例来说明推断过程. 对节点 1 进行错误注入后导致了 SDC. 图 1 中灰色节点经过比较阶段后被判定发生了错误. 从节点 1 开始进行搜索,发现节点 1 的子孙节点 18 是关键指令,并且是 SDC 脆弱指令(识别方法步骤 2 后,节点 18 属于 Γ_{SDC}). 搜索节点 18 的祖先节点,发现错误节点 1,7 和 13 满足判定 SDC 脆弱指令的充分条件,推测出节点 7 和 13 为 SDC 脆弱指令.

5 验证实验

5.1 实验方法

本文的实验平台是 Ubuntu10.04,并采用工具 Pin[20] 对原有指令执行进行修改,添加错误注入的代码. 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.

错误注入是根据注入计划表来依次进行的. 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;注入位表明了对硬件位置的第几个二进制位进行注入.

测试程序是西门子标准测试集[21],包括 rep(完成字符匹配和替换)、sc 和 sc2(优先级调度算法)、pri 和 pri2(词法分析)、tot(统计数据). 对每个测试程序,本文随机选择 3 组输入进行实验.

5.2 实验结果

本文从准确率、覆盖率、注入错误数和时间代价 4 个方面来评价 SDC 脆弱指令识别方法. 其中,准确率用来衡量推断算法得到的 SDC 脆弱指令的准确程度;覆盖率用来衡量得到的 SDC 脆弱指令的完整程度.

为了计算准确率和覆盖率,对测试程序的每一条指令都进行错误注入,找出了真实的 SDC 脆弱指令集合 L_{SDC}. 以下的描述中,S 表示根据推断算法得到的 SDC 脆弱指令集合,F 表示根据错误注入得到的 SDC 脆弱指令集合,$S\cup F$ 表示本识别方法得到的 SDC 脆弱指令集合.

5.2.1 准确率

准确率的计算过程如式(1)所示. $card(S)$ 代表集合 S 元素的个数. 由于错误注入得到的 SDC 脆弱指令都是正确的,因此不需要考察集合 F. 式(1)右侧分子代表推测得到的 SDC 脆弱指令被验证为正确的指令数,分母是推测得到的 SDC 脆弱指令总数.

$$accuracy=\frac{card(S\cap L_{\mathrm{SDC}})}{card(S)}. \tag{1}$$

各测试程序的准确率如图 4 所示. 其中,每个测试程序都包含 3 组输入. 实验表明,6 个测试程序实验结果的平均准确率为 98.8%;除了 sc2 的第 1 组输入,其他测试的准确率都大于 97%;对于 pri2 和 tot 这 2 个测试程序,准确率为 100%;sc2 的第 1 组输入推测的错误主要是特殊地址相关的指令,由于该地址存放的是结构化数据的指针,改变指针导致了崩溃而没有产生 SDC,通过分析程序的数据结构可以解决这一问题,这是推断算法能够改进的一个地方.

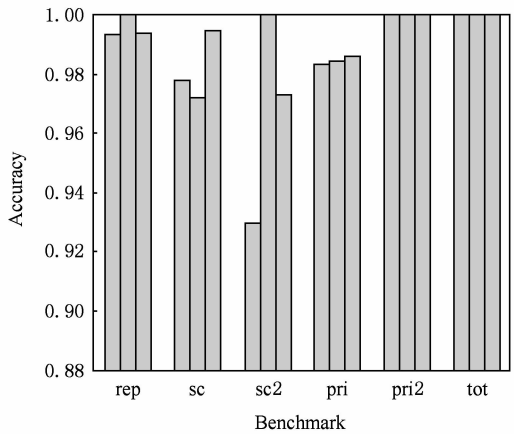

Fig. 4 Accuracy rate of our method.

图 4 本文方法的准确率

图 5 比较了本文与其他 SDC 脆弱指令识别方法的准确率,图 5 中的 SDCPredictor 为本文的识别方法. Relyzer 只通过错误注入来得到 SDC 脆弱指令,因而准确率是 100%. Shoestring 认为所有影响全局内存或函数参数的指令都是 SDC 脆弱指令,准确率为 77.2%,比本文的方法低 21.6%. 准确率低的原因是 SDC 脆弱指令与应用程序的逻辑高度

相关,影响全局内存或函数参数的指令不一定会产生 SDC.

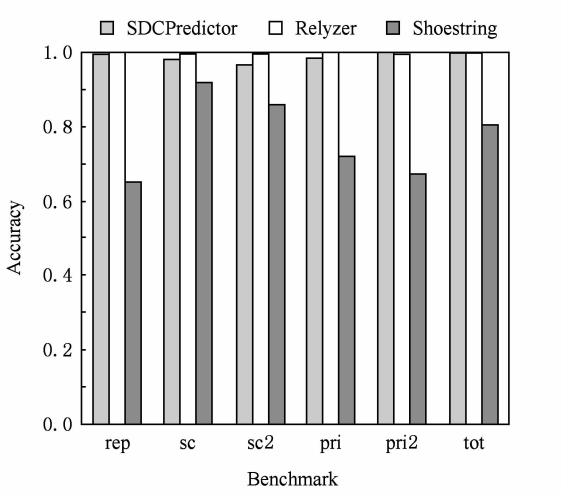

Fig. 5 The comparison of accuracy rate.

图 5 准确率比较

5.2.2 覆盖率

覆盖率的计算过程如式(2)所示.式(2)右侧分子是识别方法得到的 SDC 脆弱指令中被验证为正确的指令数,分母是真实的 SDC 脆弱指令总数.

$$coverage=\frac{card(S\bigcup F\bigcap L_{SDC})}{card(L_{SDC})}. \tag{2}$$

图 6 比较了本文识别方法与其他 SDC 脆弱指令识别方法的覆盖率.本文识别方法的平均覆盖率是 96.9%,比 Relyzer 高 1.7%.因为本文采用了与 Relyzer 类似的等价类错误注入压缩方法,所以遗漏的 SDC 脆弱指令与 Relyzer 相似,主要是地址相关的指令.这些指令经过错误注入后发生 SDC 的概率很小,甚至是否发生 SDC 与当次的地址有关,因而很难完整覆盖.

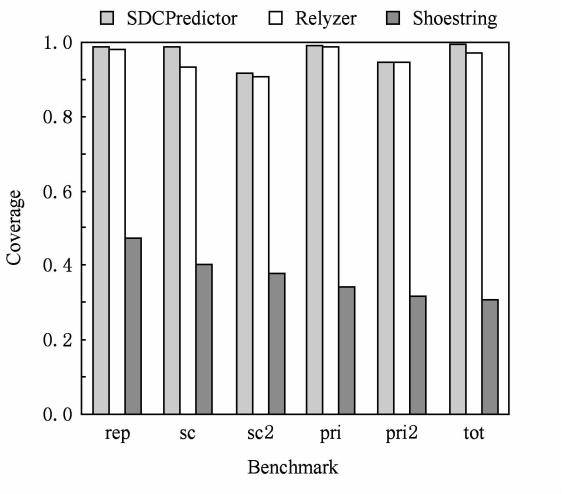

Fig. 6 The comparison of coverage rate.

图 6 覆盖率比较

本识别方法的覆盖率比静态分析方法 Shoestring 高 59.9%.Shoestring 只选择了影响全局内存和函数参数的指令作为 SDC 脆弱指令,导致其他形式的指令都没有被覆盖到.

5.2.3 注入错误数与时间代价

本节比较本识别方法、Relyzer 和全注入(每条指令都注入)的注入错误数和时间代价.图 7 显示了每个测试程序的注入错误数,图 8 显示了时间代价.其中,Full-scale 代表全注入,纵轴显示了每种方法与全注入的注入错误数或时间代价的比值.注入错误数分为关键指令和非关键指令 2 部分.实验中,非关键指令的部分占总注入错误数的 87.9%,是注入错误数的主要部分.由于采用了推断算法,非关键指令的注入错误数得到了大量削减.经过统计,本识别方法比 Relyzer 的注入错误数平均减少 25.6%,

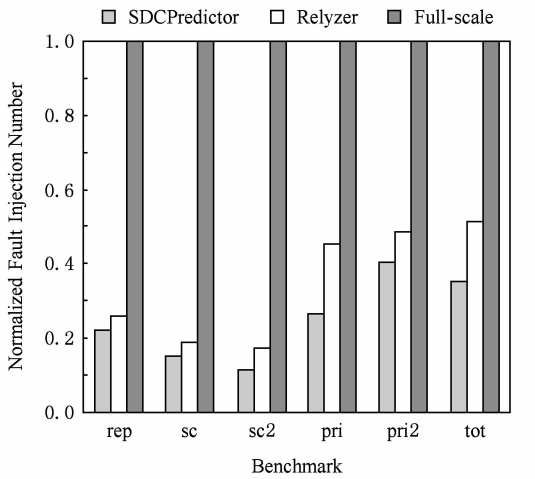

Fig. 7 The comparison of number of fault injections.

图 7 注入错误数比较

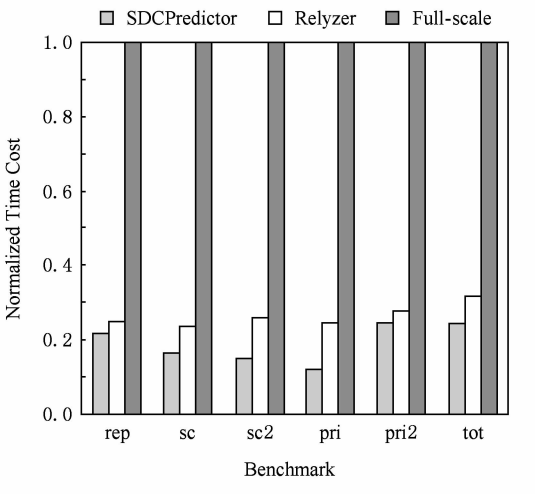

Fig. 8 The comparison of time spent on injection.

图 8 时间代价比较

时间代价平均减少 27.9%;相比于全注入,注入错误数平均减少 74.6%,时间代价平均减少 80.8%.

5.2.4 小 结

综上所述,本文的推断算法得到的 SDC 脆弱指令的准确率为 98.8%,只存在很少的误判;与典型静态注入方法 Relyzer 相比,本识别方法的覆盖率高 1.7%,时间代价低 27.9%,因此错误注入更加高效.

6 结束语

本文通过建立程序的数据关联图,分析了导致 SDC 的错误传播过程,开发了 SDC 脆弱指令识别方法.该方法在保证高准确度和高覆盖率前提下,大大降低了注入错误数和时间代价.

在得到 SDC 脆弱指令后,下一步需要进行检测机制的设计,包括检测器的位置、形式以及检测代价的评估.本文的推断算法实际上找到了导致 SDC 的错误传播路径以及路径上的关键点(关键指令节点).在未来的工作中,将研究检测器的建模以及检测代价的优化等内容.

参 考 文 献

[1] Walters J P, Zick K M, French M. A practical characterization of a NASA SpaceCube application through fault emulation and laser testing [C] //Proc of IEEE Int Conf on Dependable Systems and Networks. Piscataway, NJ: IEEE, 2013: 1-8

[2] Xing Kefei. Single event effect detection and mitigation techniques for spaceborne signal processing platform [D]. Changsha: National University of Defense Technology, 2007 (in Chinese)
(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[D]. 长沙: 国防科学技术大学, 2007)

[3] Xu Jianjun, Tan Qingping, Li Jianli, et al. An extendable control flow checking method based on formatted signatures [J]. Journal of Computer Research and Development, 2011, 48(4): 638-646 (in Chinese)
(徐建军, 谭庆平, 李建立, 等. 一种基于格式化标签的可扩展控制流检测方法[J]. 计算机研究与发展, 2011, 48(4): 638-646)

[4] Racunas P, Constantinides K, Manne S, et al. Perturbation-based fault screening [C] //Proc of the Int Symp on High Performance Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2007: 169-180

[5] Gu W, Kalbarczyk Z, Iyer R K, et al. Characterization of Linux kernel behavior under errors [C] //Proc of IEEE Int Conf on Dependable Systems and Networks. Piscataway, NJ: IEEE, 2003: 22-25

[6] Wang N J, Patel S J. ReStore: Symptom-based soft error detection in microprocessors [J]. IEEE Trans on Dependable and Secure Computing, 2006, 3(3): 188-201

[7] Li M L, Ramachandran P, Sahoo S K, et al. Understanding the propagation of hard errors to software and implications for resilient system design [C] //Proc of Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM, 2008: 265-276

[8] Dimitrov M, Zhou H. Unified architectural support for soft-error protection or software bug detection [C] //Proc of Int Conf on Parallel Architecture and Compilation Techniques. Los Alamitos, CA: IEEE Computer Society, 2007: 73-82

[9] Reis G A, Chang J, August D I. Automatic instruction-level software-only recovery [J]. IEEE Micro, 2007, 27(1): 36-47

[10] Feng S, Gupta S, Ansari A, et al. Shoestring: Probabilistic soft error reliability on the cheap [C] //Proc of Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM, 2010: 385-396

[11] Hari S K S, Adve S V, Naeimi H. Low-cost program-level detectors for reducing silent data corruptions [C] //Proc of IEEE Int Conf on Dependable Systems and Networks. Piscataway, NJ: IEEE, 2012: 1-12

[12] Yim K S, Pham C, Saleheen M, et al. Hauberk: Lightweight silent data corruption error detector for GPGPU [C] //Proc of IEEE Int Symp on Parallel & Distributed Processing Symp. Piscataway, NJ: IEEE, 2011: 287-300

[13] Pattabiraman K, Kalbarczyk Z, Iyer R K. Application-based metrics for strategic placement of detectors [C] //Proc of IEEE Pacific Rim Int Symp on Dependable Computing. Piscataway, NJ: IEEE, 2005: 75-82

[14] Pradeep R, Prabhakar K, Jeffrey W K. Statistical fault injection [C] //Proc of Dependable Systems and Networks. Piscataway, NJ: IEEE, 2008: 122-127

[15] Hari S K S, Adve S V, Naeimi H, et al. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults [C] //Proc of ASPLOS. New York: ACM, 2012: 123-134

[16] Li J, Tan Q. SmartInjector: Exploiting intelligent fault injection for SDC rate analysis [C] //Proc of IEEE Int Symp on Defect and Fault Tolerance in VLSI and Nanotechnology Systems. Piscataway, NJ: IEEE, 2013: 236-242

[17] Xu X, Li M L. Understanding soft error propagation using efficient vulnerability-driven fault injection [C] //Proc of IEEE Int Conf on Dependable Systems and Networks. Piscataway, NJ: IEEE, 2012: 1-12

[18] Pattabiraman K, Nakka N, Kalbarczyk Z, et al. SymPLFIED: Symbolic program-level fault injection and error detection framework [C] //Proc of Dependable Systems and Networks. Piscataway, NJ: IEEE, 2008: 472-481

[19] Nethercote N, Mycroft A. Redux: A dynamic dataflow tracer [J]. Electronic Notes in Theoretical Computer Science, 2003, 89(2): 149-170

[20] Luk C K, Cohn R, Muth R, et al. Pin: Building customized program analysis tools with dynamic instrumentation [J]. ACM Sigplan Notices, 2005, 40(6): 190-200

[21] Hutchins M, Foster H, Goradia T, et al. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria [C] //Proc of the 16th Int Conf on Software Engineering. Washington, DC: IEEE Computer Society, 1994: 191-200

Ma Junchi, born in 1988. PhD candidate. His current research interests include soft error mitigation and software reliability.

Wang Yun, born in 1967. Professor and PhD supervisor. Member of China Computer Federation. Her current research interests include distributed computing, fault tolerance and wireless sensor network.

Cai Zhenbo, born in 1971. Research fellow. His current research interests include spacecraft environment engineering.

Zhang Qingxiang, born in 1971. Research fellow. His current research interests include spacecraft environment engineering.

Wang Ying, born in 1982. Engineer. Her current research interests include spacecraft environment engineering.

Hu Cheng, born in 1988. PhD candidate. His current research interests include distributed computing and wireless rechargeable sensor network.