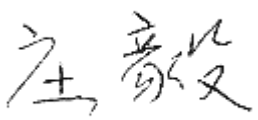
中图分类号：TP311　　论文编号：1028716 20-S078
学科分类号：083500

硕士学位论文

面向辐射环境的软错误检测技术研究

研究生姓名　　郑伟宁
学科、专业　　软件工程
研 究 方 向　　软件可靠性
指 导 教 师　　庄　毅　教授



南京航空航天大学
研究生院　计算机科学与技术学院
二〇二〇年三月

Nanjing University of Aeronautics and Astronautics
The Graduate School
College of Computer Science and Technology

Research on Soft Error Detection Technology for Radiation Environment

A Thesis in

Software Engineering

by

Zheng Weining

Advised by

Prof. Zhuang Yi

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Engineering

March, 2020

承诺书

本人声明所呈交的硕士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：郑伟宁

日　　期：2020.04.19

摘 要

由于太空辐射的作用，产生的单粒子翻转对计算机等电子设备造成的影响日益严重。单粒子翻转是指由辐射产生的高能粒子使得电子设备内部电路受到干扰，造成信息逻辑位翻转的现象。由于单粒子翻转造成的软错误具有位置随机性、高可传播性、强隐蔽性等特点，在辐射环境下如何提高卫星、飞机等星载和机载计算机的可靠性是一项挑战。因此，研究面向单粒子翻转的错误检测及加固方法具有重要的意义。

论文工作深入研究了由于单粒子翻转造成的软错误的检测及加固技术。具体研究工作如下：（1）研究了单粒子翻转的损伤机理，进行二次开发，搭建了模拟单粒子翻转的故障注入工具及实验验证平台。（2）分析了指令的脆弱性及相关特征，提出了基于 SDC 多位脆弱性分析的数据流错误检测及加固方法。该方法可通过指令特征提取和指令识别加固两个阶段对程序自动进行加固，得到具有数据流检错能力的加固程序。使用数据流故障注入工具对该方法的性能进行了对比实验。实验结果表明，该方法具有更准确的脆弱性指令识别能力，更高的错误检错率以及更低的时空开销。（3）分析了程序控制流相关信息，提出了基于多层分段标签的控制流错误检测方法。该方法可通过对多层分段标签的更新和检查，在线检测出程序的控制流错误。我们设计了新颖的标签结构与计算方法，可降低控制流错误检测的时空开销，并具有处理复杂程序以及检测控制流错误的能力。对比实验结果表明，基于多层分段标签的控制流错误检测方法具有检错能力高，时空开销低的优点。（4）给出了数据流错误检测和控制流错误检测的综合错误检测方法，并对该方法进行了实验评估。（5）设计并初步实现了面向单粒子翻转的错误检测及加固软件。主要包含了三种错误检测加固功能：数据流错误检测功能、控制流错误检测功能和综合错误检测功能。

使用单粒子故障注入工具对实现的软件进行测试。测试实验结果表明，设计并初步实现的面向单粒子翻转的错误检测及加固软件具有数据流和控制流错误检测功能，可自动实现错误检测及加固功能。

关键词：单粒子翻转，数据流检错，指令脆弱性分析，控制流检错，LLVM

ABSTRACT

The impact of single-event upset(SEU) due to space radiation on electronic devices such as computers is increasingly profound. SEU refers to the phenomenon that high-energy particles generated by radiation cause interference in the internal circuits of electronic equipment, which eventually causes state change of the logic bits of information. Soft errors caused by SEU feature position randomness, wide spreadability and high concealment. Therefore, it is challenging to improve the reliability of computers such as satellites and airplanes in a radiant environment. It is of great significance to study software error detection and consolidation methods for radiation environments.

This paper studies software error detection and consolidation techniques caused by SEU. The contributions of this work are as follows: (1) The damage mechanism of SEU was studied. Through secondary development, a fault injection tool and experimental verification platform for simulating SEU was established. (2) The vulnerability and related characteristics of the instruction was analyzed and SDCVA-OCSVM (Method for detecting SDC error using SDC vulnerability analysis and single-class support vector machine) was proposed. SDCVA-OCSVM can strengthen the program through two stages of instruction feature extraction and instruction recognition and reinforcement, and obtain a reinforcement program with data flow error detection capability. The performance of the method is compared with the data flow fault injection tool. Experimental results show that compared with similar methods, SDCVA-OCSVM has more accurate instruction recognition ability, higher error detection rate, and lower time and space overhead. (3) The information about program control flow was analyzed and CFMSL(A control flow detection method based on Muti-layer Segmented Labels) was proposed. CFMSL can detect the control flow errors of the program online by updating and checking the multi-layer segment labels. We designed a novel label structure and calculation method that can reduce the time and space overhead of the method. CFMSL has the ability to handle complex programs and detect subtle control flow errors. Comparative experimental results show that CFMSL has the advantages of high error detection capability and low time and space overhead. (4) A comprehensive error detection method for data flow error detection and control flow error detection was proposed, and the method is experimentally evaluated. (5) Software error detection and hardening software for single event upset was designed and initially implemented. It mainly involves three types of error detection and consolidation functions: data flow error detection, control flow error detection and comprehensive error detection.

We used single particle fault injection tools to test the implemented software. The test results

show that the software error detection and hardening software for single event upset designed and initially implemented has data flow and control flow error detection functions and can automatically implement soft consolidation functions.

Key words： Single-event upset，Data flow error detection，Instruction vulnerability analysis，Control flow error detection， Low Level Virtual Machine

目 录

图表清单

注释表

I_{ecSDC}	易引发 SDC 错误的指令	I_{sset}	程序指令集合
I_i	第 i 个静态指令	$size$	集合内元素个数
I_{dset}	程序动态指令集合	n_i	第 i 条指令的执行次数
P_1	单位错误翻转概率	P_2	二位横向错误翻转概率
P_3	二位竖向错误翻转概率	P_4	二位交叉错误翻转概率
P_5	三位横向错误翻转概率	P_6	三位竖向错误翻转概率
P_7	三位 L 型错误翻转概率	P_{ecSDC}	指令多位 SDC 脆弱性
R_j	故障注入实验总次数	T_i	故障注入造成 SDC 次数
L_i	样本类型标签	$\vec{F}_i$	指令的特征向量
S	脆弱性阈值	$life(num)$	数据变量生命周期
$width(num)$	数据变量的二进制位数	F_{data}	数据变量传播性特征
P_{mask}	屏蔽概率参数	$f(x)$	二进制数 x 的总位数
$f_1^x(b_1,b_2)$	x 中b_1至b_2位中 1 的总位数	$f_0^x(b_1,b_2)$	x 中b_1至b_2位中 0 的总位数
$Z_2^0(x)$	2 位以上均为 0 的位数集合	$Z_3^0(x)$	3 位以上均为 0 的位数集合
$Z_2^1(x)$	2 位以上均为 1 的位数集合	$Z_3^1(x)$	3 位以上均为 1 的位数集合
f_i	特征向量中第 i 个特征	$feature_i$	未经过处理的第 i 项指令特征
b	程序的基本块	B	程序基本块集合
E	程序控制流跳转边集合	e	跳转边
$pred$	基本块的前驱集合	PG	程序控制流程图
TO	O 型基本块	$succ$	基本块的后继集合
GS	动态全局标签	TM	M 型基本块
GS_1	动态全局标签层号段	BS	基本块标签
BS_1	基本块标签层号段	GS_2	动态全局标签标签值段
d	差值参数	BS_2	基本块标签标签值段
d_1	差值参数层号段	Lv	基本块层级
1_{set}	1 位集合	d_2	差值参数标签值段

缩略词

缩略词	英文全称
SEU	Single Event Upset
FPGA	Field－Programmable Gate Array
SDC	Silent Data Corruption
SIHFT	Software-Implemented Hardware Fault Tolerance
SWIFT	Software Implemented Fault Tolerance
I_{ecSDC}	Instructions that easily cause SDC errors
ECCA	Enhanced Control-Flow Checking Using Assertions

CFCSS	Control-Flow Checking by Software Signatures
RCFC	Regularized Control Flow Checking Algorithm
CEDA	Control-Flow Error Detection Using Assertions
HETA	Hybrid error-detection technique using assertions
S-SETA	Selective software-only error-detection technique using assertions
RSCFC	Relationship Signatures for Control Flow Checking
CEDBR	Control-flow Error Detection based on Basic-block Repartition
GDB	GNU Project debugger
SEDHS-SEU	Software error detection and hardening software for single event upset
SDCVA-OCSVM	Method for detecting SDC error using SDC vulnerability analysis and one-class support vector machine
IFE	Instruction feature extraction
I-I_{ecSDC}	Identification of instructions that easily cause SDC errors
VAR	Variables technology
SSA	Static Single Assignment
CFMSL	Control Flow detection method based on Muti-layer Segmented Labels
IR	Intermediate Representation
CFE	Control Flow Error
LLVM	Low Level Virtual Machine
OCSVM	One Class Support Vector Machine
IRVAR	Intermediate Representation VAR

第一章 绪论

1.1 课题背景与研究意义

1.1.1 太空辐射引发的单粒子效应

人类对太空的探索已有半个多世纪的历程，随着探索的深入，航天技术也不断进步。我国虽然在航天领域起步较晚，但经过全体人民的努力奋斗，已经相继完成了卫星发射、载人航天和月球探测等项目，成为了世界先进水平的航天强国。在众多的航天器中，人造卫星是应用较广泛的一种[1]。据美国卫星目录统计，自 1957 年苏联发射了世界上第一颗卫星后，截至 2019 年 6 月 23 日，全世界已经发射了 8558 颗人造卫星[2]，而且考虑到各国军事保密问题，未来卫星数量还会大幅度增加。如我国计划于 2020 年发射的嫦娥 5 号探测器，北斗系导航卫星等。然而，由于宇宙环境的复杂性，卫星运行往往会遭受严苛的挑战。在人造卫星被广泛应用的今天，卫星内部电路集成度不断提高，电子元件不断减小，对辐射也越来越敏感。

太空环境中，电子设备受到辐射影响的概率极高，而在这其中单粒子效应是太空辐射环境中造成设备损坏的主要原因之一[3]。单粒子效应是指当辐射引发电子元件的逻辑错误，使元件内部受到干扰而出现故障，轻则导致设备出现功能中断，重则直接烧毁元件造成巨大损失！1990 年，我国研制的第一代准极地太阳同步轨道气象卫星风云一号中的 FY-1B 卫星升空。在正常运行 165 天后，星载计算机开始频繁受到单粒子翻转影响，工作出现不稳定征兆。最终 FY-1B 卫星没有达到预期的设计寿命要求，因单粒子翻转影响而于 1992 年提早失效[80]。1994 年，我国自主研发的地球同步转移轨道卫星实践四号(SJ-4)于西昌卫星发射中心发射成功。实践四号升空后，其上搭载的两台用于单粒子事件测量的监测装置，在入轨后短短的 19 天内就发生了 65 次翻转[81]。2017 年，中国首颗暗物质粒子探测卫星“悟空”因意外导致接收数据量锐减，后经调查，其故障原因主要就是单粒子效应。2019 年，亚特兰大新闻报道，一对夫妇驾驶的丰田汽车突然加速导致二人受伤，专家经过调查发现事故的起因是汽车控制系统受到单粒子翻转影响而突然失控。

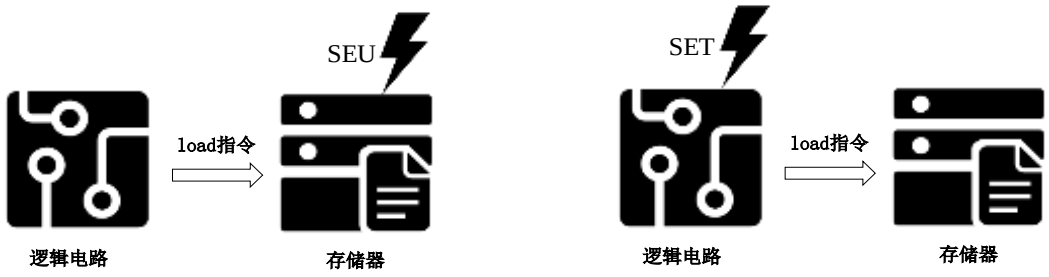
太空环境中存在多种高能粒子，当它们撞击航天控制计算机中的敏感区域时，单粒子效应就会发生。根据一些物理条件不同，如入射角度，撞击位置等，单粒子效应最终所造成的后果也不同。随着深入的研究，已发现单粒子效应包括单粒子翻转和单粒子瞬态脉冲等多种类型。到目前为止，已被发现并且定义的单粒子效应类型如表 1.1 单粒子效应类型 [59]所示。

表 1.1 单粒子效应类型

类型	英文缩写	定义
单粒子翻转	SEU(single event upset)	存储信息发生改变
单粒子瞬态脉冲	SET(single event transient)	瞬态电流在逻辑电路中传播，导致输出错误
单粒子闩锁	SEL(single event latchup)	电路中的产生大电流反馈状态
单粒子烧毁	SEB(single event burnout)	大电流下的二次击穿导致器件不可恢复的永久短路
单粒子栅穿	SEGR(single event gate rupture)	强电场导致的局部电介质击穿
单粒子扰动	SED(single event disturb)	数字电路逻辑状态短时间变化，但未导致翻转
单粒子功能中断	SEFI(single event function interrupt)	控制部件出错导致控制单元功能中断
单粒子位移损伤	SPDD(single particle displacement damage)	因位移效应造成的永久损伤
单粒子快速反向	SES(single event snapback)	在器件中产生大电流再生状态

从表 1.1 可以看出，太空环境中的单粒子效应会导致航天设备发生多种故障。其中单粒子闩锁、单粒子烧毁、单粒子栅穿、单粒子功能中断、单粒子位移损伤、单粒子快速反向等单粒子效应容易造成设备损毁，这些现象从所造成的结果来说较为明显，很容易确定故障的发生[4][5][6]。

单粒子翻转、单粒子瞬态脉冲和单粒子扰动会影响设备的正确运行，但又不会对设备造成永久性破坏。参考相关文献，本文以逻辑电路和存储器为例，说明这几种单粒子效应的原理[60]。当高能带电粒子撞击逻辑电路和存储器时，由于粒子携带大量的电荷，直接改变了此时电路的运行状态，这种现象就是单粒子翻转。如图 1.1 (a)所示，单粒子翻转直接发生在存储器中，改变了存储器内数据，导致其无法正确工作。另一方面如图 1.1 (b)所示，若带电粒子影响的是逻



(a) 单粒子翻转示意图　　(b) 单粒子瞬态脉冲示意图

图 1.1 单粒子翻转与单粒子瞬态脉冲影响示意图

辑电路，此时逻辑电路出现了位翻转故障，这种故障又通过某种方式传播到了存储器中，如“load”指令。这种现象也同样改变了存储器内数据，但此时发生的现象属于单粒子瞬态脉冲。若逻辑电路受单粒子翻转发生了变化，但是存在时间较短，且未对存储器产生较大影响，则发生的是单粒子扰动现象。

1.1.2 单粒子翻转问题与软错误

在计算机运行过程中，因为信号或者数据不正确造成的故障被称为软错误[7]。导致软错误的原因有多种，从电路设计出现问题到元件之间相互干扰均有可能，但相比单粒子翻转所引起的软错误事故相比，其他原因造成软错误事故的占比较小[8][9]。软错误会改变寄存器或数据存储器中的数值，从而导致处理器将程序带入错误循环或者直接进入死循环无法跳出。除此之外，这种故障还可以修改某些计算数据值，造成数据结果的错误。由于软错误发生时缺乏足够“明显”的信息，使用者往往在错误发生了很久之后才能察觉，甚至始终没有意识到发生过软错误。

单粒子效应有众多分类，这其中单粒子翻转所造成软错误属于软件错误，其他诸如单粒子栅穿，单粒子烧毁等造成的绝对故障则属于硬件错误。单粒子翻转所引起的软错误虽然不一定造成绝对故障，但是仍可造成严重后果。而且和其他错误相比，软错误的传播过程十分隐蔽。首先程序很有可能是正常执行的，不一定存在系统错误的提示，因而很难被使用者及时捕获。其次是单粒子翻转的产生具备很强的随机性，会根据受创部位不同而产生不同的结果，很难使用特别固定的模式去判断它。单粒子翻转所造成的位翻转发生在不同部件，就会造成不同的结果，具备随机性和偶然性。在本文的研究工作中将所有由单粒子翻转引起的软错误分为两类：数据流错误和控制流错误，具体分类情况如图 1.2 单粒子翻转造成软错误类型分类所示。

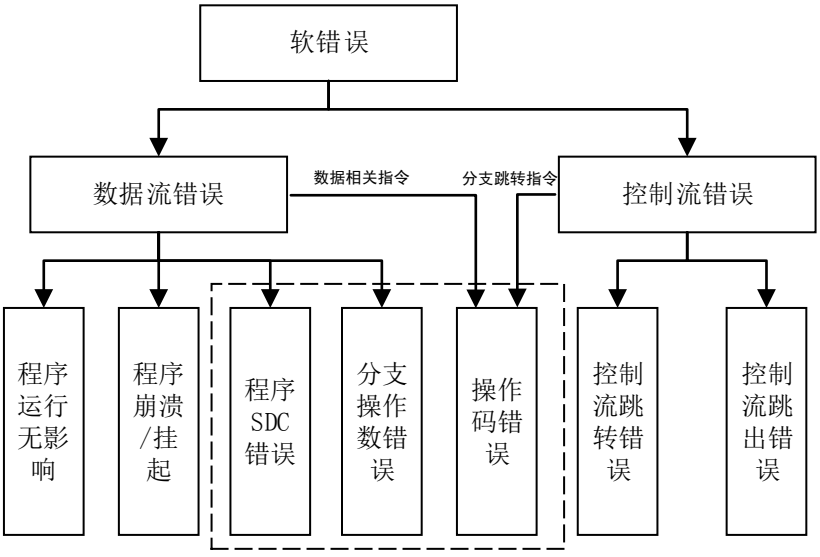


图 1.2 单粒子翻转造成软错误类型分类

数据流错误一般较为明显，当单粒子翻转发生在程序使用的数据位置上时，就会导致数据

流错误。如内存和寄存器中的数据单元，总线中的数据总线，当高能粒子在这些位置上引发了单粒子翻转时，就很容易改变程序正在使用或者将要使用的数据，这往往会导致程序发生数据流错误。

程序发生数据流错误时，根据其运行结果可分为三类。(1)程序运行无影响。尽管数据出现了错误，但在程序运行过程中错误被屏蔽了。(2)程序崩溃/挂起。关键位置的数据出现错误，并且错误不断传播，最终导致程序崩溃或进入死循环，无法跳出。(3)无记载数据损坏（Silent Data Corruption，SDC）。这类错误难以检测，具体表现为程序似乎正常执行，但是最终数据结果或者中间过程的数据出现了错误。

另一方面，若程序执行时，单粒子翻转改变了内存指令或程序寄存器 PC 中操作码部分的值，或在程序取指令操作执行之前影响了跳转指令中地址所存储的区域，就容易导致控制流错误。此时程序因单粒子翻转跳转到了一个错误的地址，从而影响了程序的正确执行。本文也将这种错误称为跳转目标地址错误，它是控制流错误中最常见的一种。

程序发生控制流错误时，根据其跳转结果可分为两类：(1)控制流跳转，是指程序控制流执行时发生控制流错误，控制流发生错误跳转，但仍然在程序内部。(2)控制流错误跳出，是指程序控制流执行时发生控制流错误，使控制流出错，跳出至程序外。

除了上述两种一般情况外，还有几种较为特殊的情况需要单独说明。

有一种单粒子翻转引发的错误较为特殊，本文称之为条件分支操作数错误。当程序执行时，影响分支跳转方向的操作数所存储的寄存器发生单粒子翻转后，其数据可能发生变化，这就引入了错误的操作数，导致条件分支指令的跳转发生了错误，没有按预期的执行路径执行。从表面上来看，这是一种控制流跳转错误，但是它的诱因却是数据出现了错误。从这一点考虑，又应该将其归为数据流错误。在本文中，我们将其归为数据流错误的一种。一方面是因为在条件分支操作数错误中，单粒子翻转所造成的最直接后果是改变了数据。另一方面是目前本文所提出的控制流检错方法无法检测这一错误，反倒是数据流错误检测方法可以检测出来。

还有一种情况，当程序执行时，计算机中存放代码段的存储区域或指令缓存区域发生单粒子翻转后，指令在被取值执行时，其操作码可能发生变化，可能会将原有的指令转换成另一条功能不一样的指令或无效的指令。本文将这种错误称为操作码错误。操作码错误的分类和原有指令类型有很大的关系，当原有指令为分支跳转指令时，则可将之归为控制流错误。若原有指令是和数据相关的指令，如读，写或运算等操作指令，则此时的单粒子翻转影响的是程序的数据处理，则可将之分为数据流错误。

1.1.3 论文研究意义

尽管单粒子翻转所引起的软错误通常不会对硬件电路造成永久性损坏，但仍会造成很严重

的后果。尽管目前已有不少针对单粒子翻转的研究成果，但仍有航天器受单粒子翻转引发的错误影响无法正常工作，这证明了仍然还有一些问题需要研究解决。

2011 年，我国首颗火星探测卫星“萤火一号”搭载在在俄罗斯“福布斯-土壤”探测器内部，由俄罗斯“天顶号”运载火箭在哈萨克斯坦拜科努尔发射场点火发射。升空后，由于负责运载的俄方 Fregat 上级控制系统遭受空间粒子辐射诱发软错误，在升空的第二天，俄方便宣布福布斯－土壤号火星探测器变轨失败，这也重挫了我国火星探测任务[11]。2019 年西班牙公布了第一颗开源卫星 FossaSat-1。这款卫星总成本不到 3 万欧元，但却着重加装了防护单粒子翻转的 MAX6369 部件。

除去航天事业外，在航空领域单粒子翻转现象也造成了重大损失。2008 年，一架 A330-303 空客受到单粒子翻转的影响，连续快速俯冲两次，造成了空乘人员和 11 名乘客受伤[10]。2017 年，科学家 lan Johnston 公开警告称，单粒子翻转会对飞机造成重大影响，甚至导致其坠落。美国联邦航空局在民用航空发动机的适航审定中明确提出申请人须评估控制系统电子器件对大气中子单粒子效应的敏感性。同样，欧洲航空安全局也在民用航空发动机的适航审定中提出了单粒子效应相关的要求。欧空局认为单粒子效应是一个基本问题，申请人有必要解决单粒子效应对机载电子设备的影响和飞机/发动机级潜在的安全性影响。

在其他领域如工业生产、民用设施，单粒子翻转引发的软错误也在不同程度上影响了电子设备的正常运行，降低了系统的可靠性。如在比利时斯哈尔贝克镇，电子器件的单位翻转为某一位候选人额外增加了 4096 个选票。由于这个候选人得到的选票超过了逻辑可能性，这个错误才被检查到，调查人员最终追查到了机器中的某个寄存器收到了单粒子翻转的影响；

在设计抗单粒子翻转错误检测与加固方案时需要综合考虑应用场景、加固效果、需求成本等多方面因素。由于人造卫星特殊的应用环境，星载控制系统中的计算机对可靠性与性能均有较高的需求[12]。 所以在性能受限的平台上，及时有效的对单粒子翻转造成的软错误进行检测恢复，针对计算机系统进行软加固，减少故障的发生以提高系统的可靠性具有重要的意义。开发一种能够有效检测单粒子翻转引发的软错误，同时具有检测成本低，检错率高等优点，具有重要意义。

1.2 国内外研究现状

针对单粒子翻转的故障检测技术从手段上来区分，可以分为基于硬件的错误检测技术和基于软件的错误检测技术两种[13]。其中基于软件的错误检测技术为本文的研究重点。基于软件的错误检测技术又可分为两类：数据流错误检测技术和控制流错误检测技术。本节会针对这两类错误检测技术，系统的分析目前国内外较为前沿的软错误检测方法。

1.2.1 错误检测技术的分类

目前针对单粒子效应的故障检测技术已有了多年的发展，美国斯坦福大学、密歇根州立大学、伊利诺斯州立大学、亚利桑那州利大学、欧洲 TIMA 实验室、法国 LAAS-CNRS 实验室、瑞典 Chalmers 大学、意大利都灵理工大学，以及国内的清华大学、国防科大、西北大学、哈尔滨工业大学、航天五院和中科院等研究机构均已开展了一系列研究，并已取得了一些成果。单从加固手段上来分，抗单粒子效应的错误检测方法包括：使用硬件的检测及加固方法和使用软件的检测及加固方法两类。

使用硬件的检测及加固方法是指通过硬件的手段来实现故障检测的技术。单粒子效应是高能粒子撞击电子元件时所产生的一系列故障。它本质上是一种硬件故障，所以在研究早期，针对单粒子效应的故障检测大多是在硬件基础上开展的。最早针对单粒子效应的防护措施是在脆弱的元件外部增加一层“包装”[14]。这种方法成本适中，手段简单，也起到了一定的防护效果。但“包装法”对可靠性要求较高的航天领域而言，还算不上有效的解决手段。首先第一点，目前常用的“包装”材料本身会散发α粒子，导致防辐射的外壳随时可能变为损害元件的入射源。第二点，太空环境中宇宙射线发出的中子粒子很容易就可以穿透“包装”材料，继续引发单粒子效应[15]。除去包装法，硬件检测技术还常用冗余法来实现故障检测[16][17][18][19][20][21][22]。冗余方法是指将设备中较易受到单粒子效应影响的脆弱部分进行备份，准备多个相同的器件。当高能粒子引发单粒子效应时造成了硬件损坏，备份器件就可以接替原始器件继续工作，维护整个系统的稳定。若未造成硬件损坏仅仅引发了软错误，也可以通过比较原始器件和备份器件的工作结果进行错误检测。目前冗余技术中，大多使用双模冗余[23]和三模冗余[24]技术。如陈玉坤等人给出的由中国运载火箭技术研究院设计的具备重构能力的三模冗余器载计算机，该计算机就是使用三机模式实现故障检测与容错[25]。另外刘蕾等人提供的由中科院上海微系统所和中科院上海微小卫星中心联合设计的星载计算机，内部采用的也是双模冗余机制[26]。美国密苏里州大学将形式化模型应用到三模冗余中，并设计了针对单粒子翻转的检错和纠错机制，同时为三模冗余系统提供了分区方案。这种方法可以提高系统的可靠度，但是有功率消耗大的缺点[27]。米兰理工大学提出使用三模冗余和部分动态可重构相结合的方法来恢复发生在 FPGA（Field－Programmable Gate Array，现场可编程门阵列）中的软错误，文中提供不同的加固细粒度，设计时需要根据实际的需求和条件限制选择不同的粒度[28]。

综合来看，使用硬件的检测及加固方法实施简单且卓有成效，但其本质是通过高昂的硬件成本来实现的。而且在实现过程中往往适用范围较小，无法进行普适性应用。更为重要的是，从卫星不断集成化小型化的发展趋势来看，这些技术大多属于“逆势而行”，不利于未来的发展。

使用软件的检测及加固方法，大部分是通过添加指令冗余和比较以检测软错误而保护计算机系统的方法。这种方法由于有指令冗余的部分存在，也可以起到一定的错误纠正作用，故也

被部分文献称为以软件实现的硬件容错技术(Software-Implemented Hardware Fault Tolerance，SIHFT)[29]。最开始 SIHFT 单指一种固定的技术，后来随着基于软件的硬件检错技术逐渐发展，SIHFT 开始泛指所有以纯软件手段实现的硬件检测技术[30]。基于软件的检错技术无需修改底层硬件，无需限制在特定的处理器或者平台上使用，具有高度的灵活性。同时软件技术中指令冗余使程序执行了额外的指令，也同样增加了时间与空间开销。

总结对比使用硬件的检测及加固方法和使用软件的检测及加固方法，我们发现使用软件的检测及加固方法具有如下优势：

(1) 较大的检测范围。使用硬件的检测及加固方法针对的检错位置较为固定，无法对没有进行过硬件冗余或修改的部分加固。但是使用软件的检测及加固方法却可以通过对不同指令的冗余来实现更大范围的错误检测覆盖。

(2) 较高的灵活度。使用软件的检测及加固方法无需修改底层硬件，无需限制在特定的处理器或者平台上使用，跨平台所需成本较小，具有高度的灵活性。

(3) 较好的优化潜力。使用软件的检测及加固方法由于不涉及底层硬件的修改，对于技术的改动不需要额外的成本，随时可以进行技术优化。且在发展过程中使用软件的检测及加固方法逐渐引入了类似选择优化，机器学习，深度学习等领域的算法内容，具有极高的发展潜力。

(4) 较低的性能开销。尽管使用软件的检测及加固方法使用指令冗余时会带来额外的时空开销，但是相较于硬件技术冗余时的高昂成本，仍有较大的竞争力，在航天技术领域更是如此。

综上所述，在对比了软硬件方法在软错误检测方面的优劣后，将使用软件的检测及加固方法列为本文的研究重点。

1.2.2 数据流错误检错技术

数据流错误中最难检测的就是 SDC 错误。与其他错误相比，SDC 错误的传播过程十分隐蔽。首先程序是正常执行的，不会有系统错误的提示，因而很难被检测机制所捕获。其次是 SDC 产生的数据错误具有很强的随机性，没有特别固定的模式和结果。因此，SDC 是数据错误中最难解决的问题之一。目前针对 SDC 错误传统的检测的方法中，大体可分为冗余检测与断言检测两种。

传统的冗余技术存在着成本问题，但该方法实施简单，运行稳定，且具有较高的检测率。美国普林斯顿大学的 Reis 提出了一种可以检测软错误的 SWIFT(Software Implemented Fault Tolerance)技术[31]。SWIFT 首先通过回收程序在执行期间未使用的指令级资源来对程序进行冗余，在程序运行的同时执行冗余计算副本，然后通过检查点比较原始程序的运算结果和冗余副本的计算结果来检测 SDC 错误。但是，这种方法设置的冗余副本需要复制大量的指令，在程序执行时也需要额外执行副本，产生了巨大的性能开销。

传统的断言检测技术则是通过插入在程序中插入断言来检测数据流错误。断言的内容是在正常情况下程序运行时的数值特征如数值区间、数值间的关系等。Hari 等人针对 SDC 错误提出了一种程序级的 SDC 检测器[32]，这种检测器就是通过在程序中植入断言的方法来检测数据流错误，有效降低了空间成本和性能开销。但是由于程序断言的提取主要靠人工分析，存在较大的随机性，所以这种方法灵活性、普适性较差。而且如果断言设置的不好，检测率就会下降。

为了规避传统的断言检测技术与冗余检测技术的缺点，学界又出现了一种将二者混合的冗余断言检测技术。这类技术会对程序中对 SDC 较为敏感的部分进行冗余，再通过插桩断言来对错误进行检测。如 Hari 等人就提出了 Relyzer 方法以检测 SDC 错误[33]，文中将程序对 SDC 错误的敏感程度称为 SDC 脆弱性，本文沿用了这一定义。该方法通过故障注入寻找程序中 SDC 脆弱性高的位置，并针对这些位置进行冗余。既节省了一部分成本，又提高了方法的灵活性与普适性。因此，冗余断言检测技术逐渐成为了 SDC 错误检测的主流手段。

尽管冗余检测技术有着不错的效果，但是如何确定程序中 SDC 脆弱性仍是一个挑战。例如，Relyzer 对 SDC 脆弱性的提取使用的就是故障注入的方法，整个过程繁琐且不易实现。目前，分析程序 SDC 脆弱性的方法主要有两种：基于故障注入的 SDC 脆弱性分析[33]、基于程序静态分析的 SDC 脆弱性分析[34]。故障注入的方法针对指令进行针对性的故障注入实验，以此来测试程序指令的脆弱性。例如，Xu Xin 等人建立了一个故障注入框架 CriticalFault[35]，该框架与传统的故障注入工具 SFI 相比，减少了至少三分之一的故障注入空间，节省了故障注入成本。另一方面，基于静态分析的方法主要是通过对程序本身的特征进行分析判别 SDC 脆弱性高的指令。Pattabirama 等人建立了一个名为 SymPLFIED 的程序级框架[36]，以抽象的符号及量化的错误值对程序错误进行全面检测，分析故障的传播路径以及指令受故障影响的程度，从而识别程序中的 SDC 脆弱指令。由于完全舍弃了故障注入实验，这类基于静态分析的技术节省了大量的成本，但是如果应用于大规模程序，则容易导致状态爆炸，时间成本远超预期。

近年来，随着人工智能技术的兴起，越来越多基于机器学习的方法被应用到了软错误检测及加固领域。与其他传统的通用技术相比，人工智能方法具有较好的灵活性和可扩展性，可以更深入的研究故障数据和损伤规律，以建立更加精准的判别与预测模型。YANG 等人提出了一种名 PVInsiden 的方法，该方法基于支持向量机，能够识别容易引发 SDC 错误，即 SDC 脆弱性高的指令 [37]。PVInsiden 使用故障注入生成训练集数据，再通过机器学习训练检测器。由于训练检测器时使用的训练集属于目标程序的一部分，所以检测器具有很强的针对性。但是 PVInsiden 会根据不同的目标程序训练出不同的检测器，所以也具有良好的灵活性。Liu 等人提出了一种名为 SDCPredictor 的人工智能方法来识别 SDC 脆弱性高的指令[38]。该方法精度较高，但是由于随机森林本身的限制，容易导致过拟合的问题。Wang 和 Dryden 则是使用神经网络构造了一种 SDC 错误检测器[39]。该方法可以检测在程序中发生了连续迭代后的 SDC 错误，灵敏

性较高，但性能开销和时间成本也很高。

综上所述，国内外针对 SDC 错误的人工智能技术进行了较多的研究，也取得了一些不错的成果。但人工智能技术需要进行大量的故障注入实验以收集样本数据，产生较高的性能、空间、时间成本，若不加以改进，难以广泛应用。除此之外，人工智能技术对于不同的程序或者程序的不同输入也会产生一定的性能差异。因此，如何进行改进，在保留人工智能技术优点的同时克服上述缺陷是进一步研究的热点。

1.2.3 控制流错误检测技术

程序控制流错误是软错误的一大类型，根据之前对单粒子翻转损伤机理的分析可知，高能粒子在与程序调用地址相关的位置引发单粒子翻转时，就有可能造成控制流错误[40]。对于任何计算机程序，只要给定程序输入，其内部执行指令的顺序就是固定的，但是控制流错误会改变这种顺序，从而影响程序的正确执行[41]。根据 Zhu[42]和 Ohlsson[43]的实验结果可知，在程序发生的软错误中，有 33%至 77%的可能性造成程序的控制流错误。

目前检测控制流错误的软件技术通常是维护并更新一个全局的动态变量，通过监测变量的变化间接监控程序的控制流，这一变量被称为标签，这种技术也被称为基于标签实现的控制流错误检测技术[44]。

Yau 等人在早期曾经提过一种使用数据库来记录程序控制流的方法，但是这种方法需要维护一个数据库。不同程序的控制流信息不同，最终会导致数据库不断扩充，开销太高[45]。该方法虽然本身存在着局限性，但是它提出了一种程序无循环间隔的概念，这一概念就是程序基本块划分的前身。ECCA(Enhanced Control-Flow Checking Using Assertions)也属于早期软件实现的控制流错误检测方法，该方法通过在程序无分支间隔的入口和出口处插桩断言来检错[46]。此文中的无分支间隔和断言与后来的基本块和标签已经非常接近了，但是由于更新操作采用乘除法，并且插桩的断言本身较为复杂，方法实施困难且性能较差。CFCSS(Control-Flow Checking by Software Signatures)是基于标签实现的控制流错误检测方法中最为经典的方法，它将程序执行的指令划分为众多集合，并以此正式提出了基本块的概念。CFCSS 将程序划分为基本块后，通过对每一基本块分配唯一的标签并添加错误检测指令来实现控制流错误检测。作为早期以纯软件方法实现的控制流错误检测技术之一，它有着重要的参考比较价值[47]。

目前主流的控制流错误检测方法大多是基于标签实现的方法，不同方法之间的主要区别在于标签设置的数量，标签更新的方式以及标签比较的位置。Vemu 等人提出的 CEDA(Control-Flow Error Detection Using Assertions)就属于标签控制流错误检测技术中表现出色的一种。CEDA 继承了部分 CFCSS 的特性，在每一基本块的入口和出口处添加了标签更新与验证体系，体现出了不错的性能[48]。Chielle 等人在 CEDA 和一种偏硬件的方法 HETA[49](Hybrid

error-detection technique using assertions)的基础上提出了 S-SETA[50]方法。S-SETA(Selective software-only error-detection technique using assertions)和 CEDA 较为相似，利用双层标签机制，通过基本块之间的关联性来计算更新运行时的标签。Zhu 等人提出了一种双标签的控制流错误检测方法 RCFC[51]。(Regularized Control Flow Checking Algorithm)该方法无论是开销还是检测性能上均不太理想，但是双标签机制却提供了一种新的思路。在检测性能要求较高但开销可以放松限制的情况下，双标签机制就是一种能有效提高错误检测率的手段。国内对基于标签的控制流错误检测方法也有了较为深入的研究。张鹏等人就提出了一种使用双指令环设计标签的 SSCFC 技术，该技术通过引入的双指令环可以有效的解决块间滞后性和配置不灵活的缺点[52]。李爱国等人则是提出了能够记录基本块关系的 RSCFC(Relationship Signatures for Control Flow Checking)技术[53]。RSCFC 和其它的标签控制流错误检测技术不同，它的基本块在设计上可以将基本块的跳转关系直接编码进来，间接的节省了标签更新语句和标签检查语句的开销。但是该方法对于基本块总数有要求，严格受到了机器字长的限制，使算法的灵活性受到了极大的影响，难以在大体量程序上使用。张倩雯提出了名为 CEDBR(Control-flow Error Detection based on Basic-block Repartition)的双标签控制流错误检测技术，该方法有着完善的规则和算法设计，有效的提高了错误检错率，是一种较为成熟的双标签算法[54]。帕尔哈提江 斯迪克[55]通过分析程序控制流，限制间接调用函数的方法来降低调用指令的数量，再加上一种二进制的检查代码，从而达到检测目的。姬秀娟[56]则是基于程序控制流程图设计了一种基于投影的模型检测静态分析算法，来提高错误检测的有效性和准确率。

综上所述，近几年来软件实现的错误检测技术已经有了长足的发展，国内外都已有了较为深入的研究。但是目前基于软件实现的控制流错误检测技术普遍存在着性能开销过大的问题。2019 年 PABHISHEK 等人提出现有的控制流方法本身插桩的指令也容易引起软错误，同时也会增加性能开销[57]。若在程序中插桩了过多的指令，软错误漏洞可能会增加而造成反效果。而航天领域对设备的性能和稳定性的要求要比地面设备高很多。如何在保证检测方法性能的同时，解决基于软件的控制流检错方法开销过大的问题是软错误检测及加固领域研究的关键。

1.3 论文研究工作

本文针对由于太空辐射中的单粒子翻转引起的软错误问题进行了损伤机理分析，并研究了单粒子多位翻转的情况。对已有的基于软件实现的控制流错误检测技术进行总结分析，设计了面向单粒子翻转的数据流错误检测和控制流错误检测方案，设计并初步实现了面向单粒子翻转的错误检测及加固软件。并使用故障注入工具对软件加固后的程序进行故障注入实验，对加固效果进行测试。本文的研究工作包括：

(1) 系统研究了现有的面向单粒子翻转引起的软错误的检测技术，分析了硬件实现的错误

检测技术和软件实现的错误检测技术的优劣。分析了国内外对数据流错误检测技术和控制流错误检测技术的研究现状，分析了现有的研究成果的优点与不足。

(2) 对单粒子翻转的损伤机理和现象开展了研究，并根据相关原理基于 GDB 进行了二次开发，分别设计了数据流故障注入工具和控制流故障注入工具。基于 GDB 进行二次开发，搭建了单粒子翻转模拟故障注入工具和实验验证平台。具体包括数据流故障注入工具，控制流故障注入工具和单粒子翻转故障注入工具。

(3) 研究了指令的 SDC 多位脆弱性分析方法，提出了一种数据流错误检测及加固方法 SDCVA-OCSVM(Method for detecting SDC error using SDC vulnerability analysis and single-class support vector machine)，对程序进行数据流错误检测及加固。该方法可以从指令传播性和固有属性两个方面提取出能够影响指令 SDC 多位脆弱性的特征，运用单类支持向量机建立高脆弱性指令识别模型，可自动识别脆弱性较高的指令；建立了一套冗余检查规则，设计了冗余加固策略。设计了数据流错误检测及加固算法，最终通过实验验证了方法的有效性。

(4) 分析了目前最为典型的两种控制流错误检测技术，讨论了目前基于标签的控制流错误检测方法中开销和漏检率的主要矛盾。提出了一种多层分段标签控制流错误检测算法 CFMSL(Control Flow detection method based on Muti-layer Segmented Labels)，设计了科学的标签结构和更新与检查规则。该方法先以层次对基本块进行划分，又对每一层次的基本块重新设计标签，通过定理验证了方法的完备性和正确性。设计了控制流错误检测及加固算法，最终通过实验验证了方法的有效性。

(5) 综合所有研究工作，本文设计并实现了面向单粒子翻转的错误检测及加固软件。设计了相关数据结构，主要功能的核心代码。给出了数据流错误和控制流错误检测及加固的核心功能模块的设计与实现技术，给出了主要功能的核心代码。并对软件进行了故障注入实验，对软件测试结果进行了评估。

1.4 论文组织结构

为了对抗单粒子翻转对电子设备的影响，本文设计并实现了面向单粒子翻转的错误检测及加固软件，论文整体结构如图 1.3 所示。全文共包括六个章节，每一章节内容如下。

第一章为绪论，主要讨论单粒子翻转相关研究以及目前抗单粒子翻转引起的软错误的国内外研究现状。

第二章为面向单粒子翻转的错误检测及加固软件的总体设计，包括总体框架和总体流程设计。

第三章介绍了基于 SDC 多位脆弱性分析的数据流错误检测及加固方法，包括对指令传播性

特征和固有性特征的提取方法，指令识别模型的建立以及方法的实验验证。

第四章介绍了基于多层分段标签的控制流错误检测方法，包括基本块按层划分规则，基本块标签分配规则，标签更新及检查规则以及方法的实验验证。

第五章介绍了面向单粒子翻转的错误检测及加固软件的实现，包括实现的具体功能，相关的数据结构，主要功能实现时所需要的核心代码。最后介绍了能进行数据流错误检测和控制流错误检测的综合加固方法，并通过实验进行了效果验证。

第六章则介绍了本文工作的总结以及展望，讨论了研究工作继续的发展与研究方向。

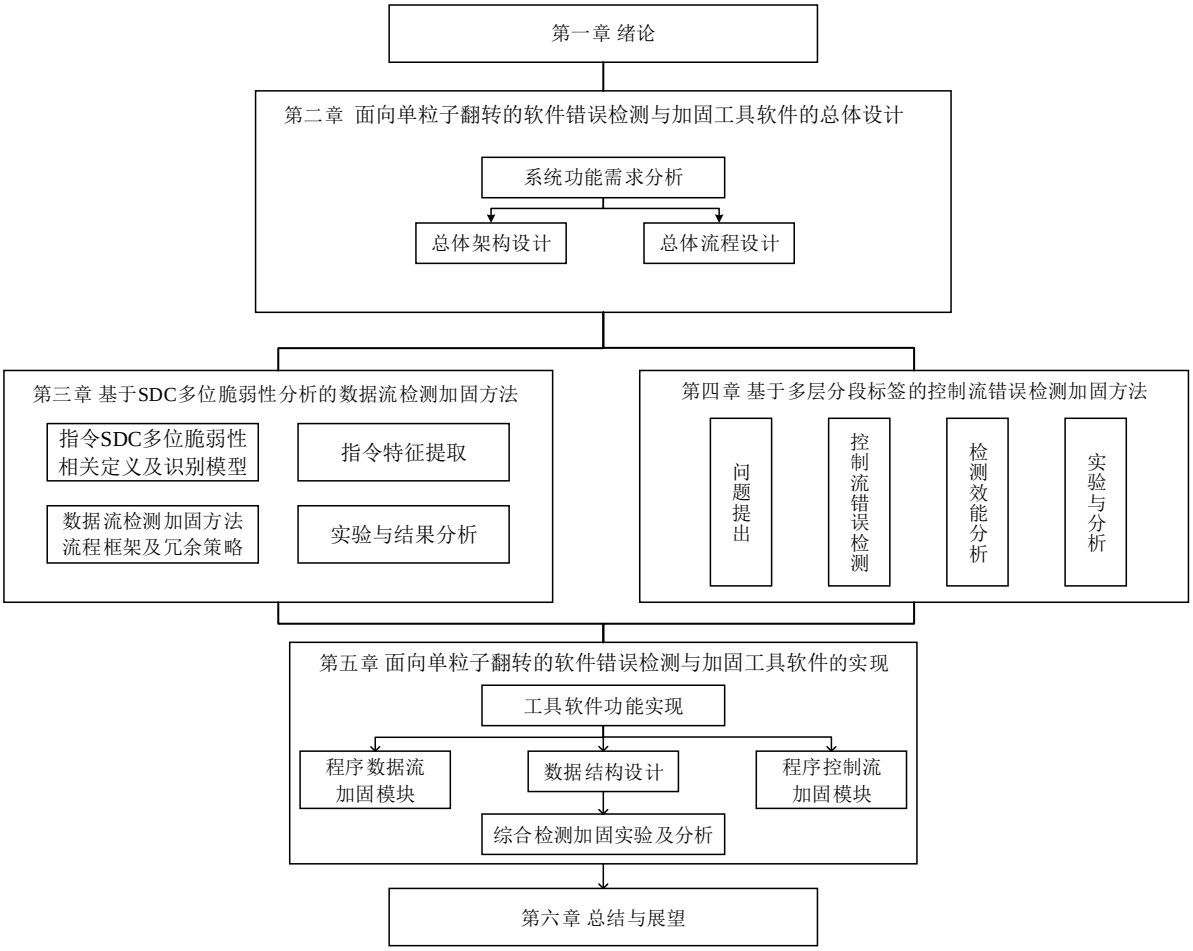


图 1.3 论文组织结构图

第二章 面向单粒子翻转的错误检测及加固软件的总体设计

针对面向单粒子翻转引起的控制流错误与数据流错误问题，本文设计并实现了面向单粒子翻转的错误检测及加固软件(Soft error detection and hardening software for single event upset，SEDHS-SEU)。尽管控制流与数据流错误均由单粒子翻转引起，却有不同的表现形式和检测方式，故 SEDHS-SEU 软件需要同时具备两种检测及加固方式。本章对 SEDHS-SEU 软件的需求进行分析，给出所需要的功能要求。进一步的对 SEDHS-SEU 软件架构进行总体设计，给出其总体流程。

2.1 SEDHS-SEU 软件的功能需求分析

SEDHS-SEU 软件主要对抗空间环境中的单粒子翻转所造成的软错误。对目标程序进行检错加固后，会让程序具有对抗单位乃至单粒子多位翻转效应的能力。对于单粒子翻转引起的数据流错误，可以采用双模检错冗余进行有效检测。对于单粒子翻转引起的控制流错误，软件可以通过对多层分段标签的更新与检查进行检测。因此，本文设计的 SEDHS-SEU 软件需要提供以下功能：

(1) 数据流错误检测及加固功能

从单粒子翻转引起的数据流错误所造成的后果来看，错误较为隐蔽且传播性较强，因此能够及早地发现错误是数据流检测及加固算法要解决的关键问题之一。同时，由于对程序加固会产生额外的时间与空间开销，从而影响设备的性能。所以实现一种开销低，检错能力强的数据流错误检测及加固方法是 SEDHS-SEU 软件的功能需求之一。

(2) 控制流错误检测及加固功能

近年来因为对错误检测率的提升，双标签控制流错误检测技术逐渐得到了发展。但是与数据流错误检测及加固方法类似，基于标签的控制流错误检测技术在插桩标签更新与检查语句时，同样会带来额外的开销。双标签控制流错误检测技术由于比传统标签技术多了一个标签，开销也相应的升高。目前而言，大部分双标签技术空间开销高达 80%左右[51][54]。故 SEDHS-SEU 软件需要设计一种能够保存双标签机制优点，又可以控制加固成本的方法。

(3) 综合加固功能

由于单粒子翻转可以随机的引发控制流和数据流两种错误，因此想要实现可以对抗单粒子翻转的 SEDHS-SEU 软件，必须对目标程序进行控制流和数据流两种错误的综合加固。使加固后的目标程序具有自动检测控制流和数据流错误的功能。

(4) 自动化处理功能

由于软件程序种类多，数量大，检错加固方法若想实现抗单粒子翻转的全面加固，需要

SEDHS-SEU 软件能够自动化，批量化的处理程序加固步骤。这是保证 SEDHS-SEU 软件可以大规模应用的根本。因此，SEDHS-SEU 软件必须具有可自动实现软错误检测及加固的能力。

(5) 运行评估显控功能

运行评估显控功能主要是控制 SEDHS-SEU 软件对目标程序进行错误检测及加固，同时对加固后的目标程序进行性能评估，并将所得到的信息进行统计与分析。运行评估显控功能需要确保对程序检测及加固过程的控制以及对加固后的程序进行评估。这是确认检测及加固过程是否正常执行并且分析检测及加固方法有效性的重要步骤。因此，SEDHS-SEU 软件需要具备高效、可靠且及时的控制功能和显示功能。

2. 2 SEDHS-SEU 软件的总体架构设计

本节给出 SEDHS-SEU 软件的总体架构设计，具体设计的架构如图 2. 1 所示，共分为系统应用层，管理层、支撑层和数据层四层。

(1) 应用层

应用层由核心加固层和实验评估层构成。

核心加固层是整个 SEDHS-SEU 软件最为核心重要的部分，负责对目标程序的加固处理。核心加固层由数据流加固子系统，控制流加固子系统和综合加固子系统构成。数据流加固子系统主要根据目标程序特点并分析程序指令信息，识别出较易受单粒子翻转的指令，再进行指令冗余加固。包含指令特征提取模块，指令识别模块，冗余加固模块。控制流加固子系统主要用于控制流错误检错加固，会分析程序控制流信息，实现程序基本块按层划分，按规则插桩标签更新检查指令。包括程序控制流分析模块，程序按层划分模块，标签更新指令插桩模块，标签检查指令插桩模块。综合加固子系统综合数据流加固和控制流加固两种方法，实现程序控制流与数据流的综合加固，包括数据流加固模块和控制流加固模块。

实验评估层主要用于评估分析系统所提供的加固方法，具体以方法的检测率和性能开销等为指标，包含控制流故障注入模块，数据流故障注入模块，综合故障注入模块。同时三种故障注入模块也分别对应控制流实验结果统计模块，数据流实验结果统计模块，综合实验结果统计模块。

(2) 管理层

管理层主要对软件运行提供重要的支持，主要功能包括：配置管理、运行监控、数据解析、数据存储、日志管理以及用户管理。配置管理主要实现对核心加固层以及实验评估层各个子系统和模块的管理，包括子系统和模块的集成注册与卸载、模块的启动与停止等；运行监控用来实时获取各个子系统和模块的状态信息，及时发现异常并进行相应的处理；数据解析模块负责对接口定义的相关协议进行解析，能够正确解析模块之间交换的数据内容；数据存储模块负责

对数据库以及文件的操作进行管理；日志管理模块负责对用户的所有请求以及系统的重要事件（如故障、状态和结果等）进行记录，并提供对日志的查看功能；用户管理模块则负责对使用者和用户进程进行管理与控制，合理分配权限。

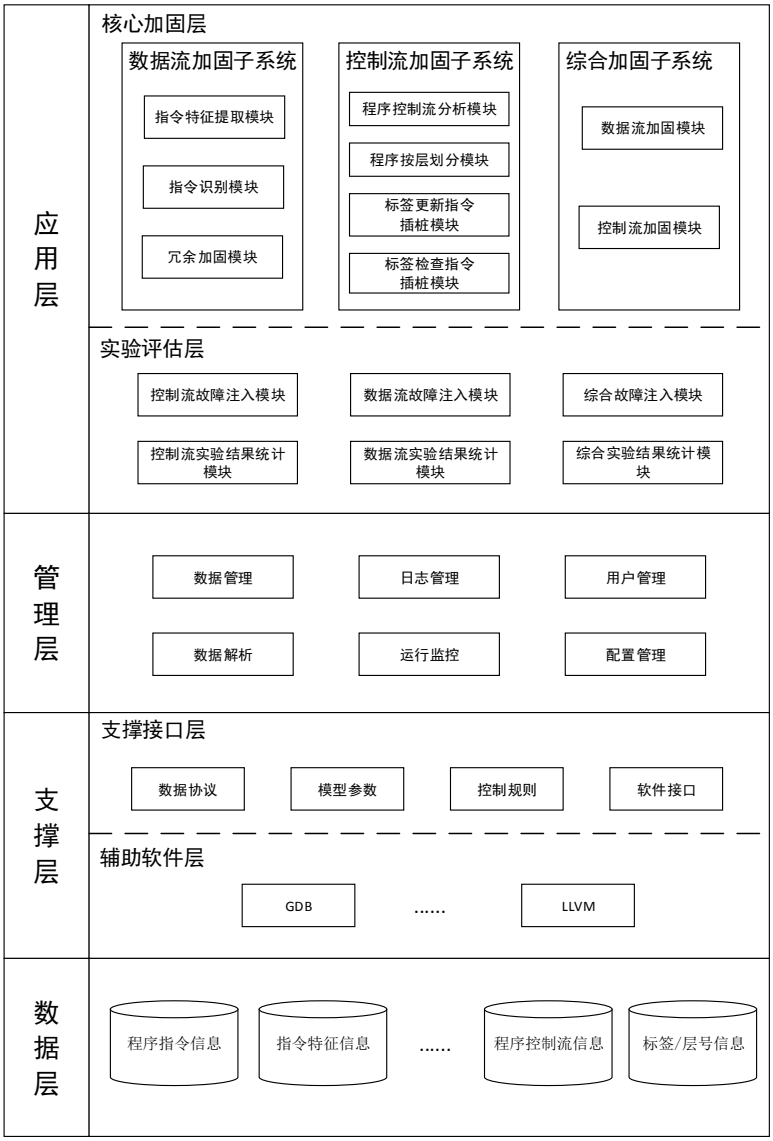


图 2. 1 SEDHS-SEU 总体架构图

(3) 支撑层

支撑层是为加固软件提供的集成框架，支撑层中辅助软件层负责系统与其他专业设计与分析软件工具(如 GDB, LLVM 等)之间的模型参数与数据之间的交互。本层将采用统一形式的模型参数或数据格式进行无缝集成，为上层的应用层和应用管理层提供支撑服务。

接口层主要采用总体定义的、统一的数据接口形式和本系统涉及到的专用接口，包括数据协议、模型参数、控制规则、GDB 调用接口以及 LLVM 数据接口等。接口层是实现数据在本

地数据库和综合数据库之间数据交换的桥梁。

(4) 数据层

数据层主要包含支撑系统运行相关数据库、硬/软件设备以及所需接口，其中数据库包含在分析目标程序过程中存储的程序指令信息，数据流方法需要提取出来的指令特征信息，控制流方法为程序分配的标签信息，层号信息等等。

2.3 SEDHS-SEU 软件的总体流程设计

SEDHS-SEU 软件主要用于对程序进行抗单粒子翻转加固，这一过程共分为三个阶段：程序分析编译阶段、程序错误检测及加固阶段和故障注入评估阶段。具体流程如图 2.2 所示。

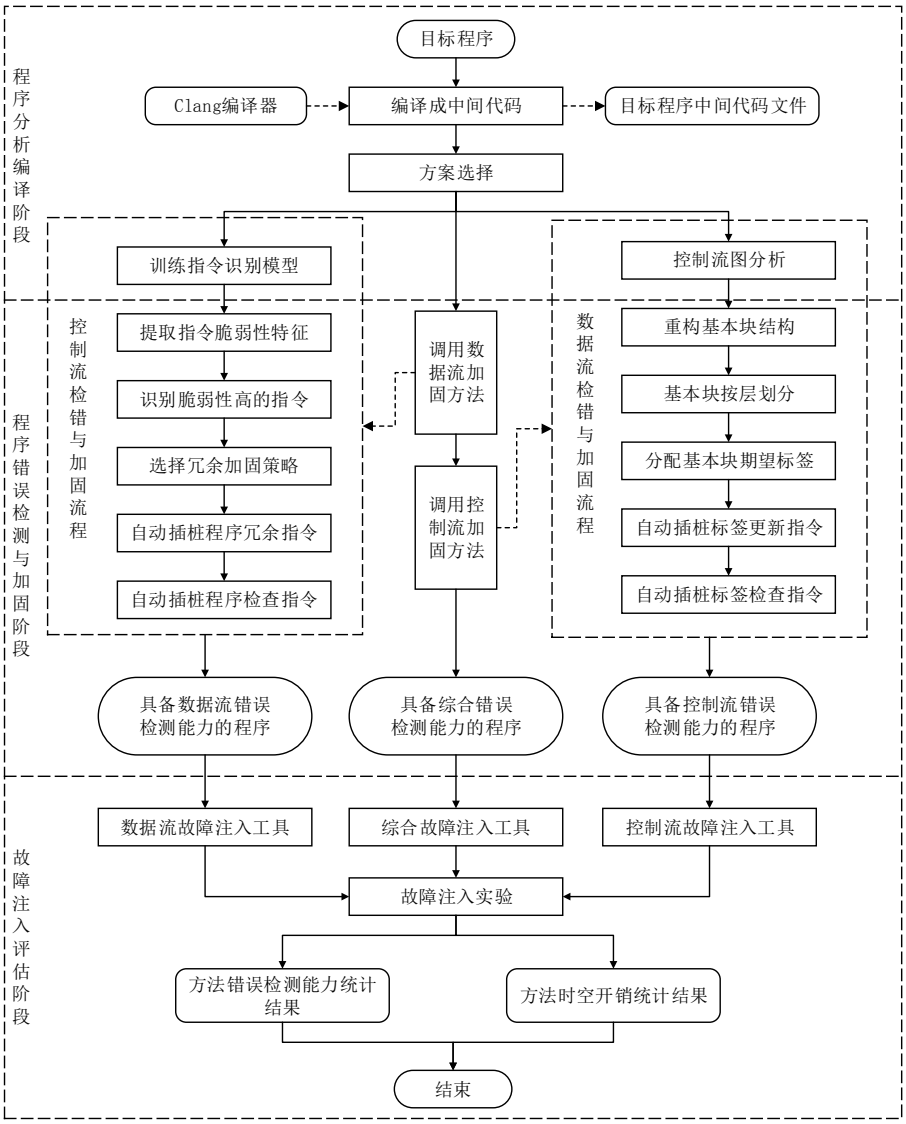


图 2.2 SEDHS-SEU 的总体流程

(1) 程序分析编译阶段

在程序分析编译阶段，软件首先需要对程序进行编译，使用 Clang 编译器[67]将其编译为中间代码文件。根据中间代码文件对指令进行分析，为后续加固方法提供足够的信息。数据流加固方法需要对目标程序的指令特征进行提取，计算包括屏蔽概率参数在内的指令特征值。控制流方法需要分析程序控制流相关信息，为后续划分层次以及分配标签做基础。

(2) 程序错误检测及加固阶段

在程序错误检测及加固阶段，则会根据选择的方法来对目标程序进行加固。数据流错误检测及加固方法需要提取训练样本指令的传播性特征和固有特征，并通过故障注入实验获取并计算训练样本的脆弱性，建立训练集。通过训练集和机器学习的方法训练指令识别模型。使用指令识别模型可识别目标程序中脆弱性较高的指令，再利用冗余加固规则对程序进行加固。控制流错误检测及加固方法则需要按层次划分程序基本块，规则相应规则分配基本块标签，插桩标签更新与检查指令。综合加固方法则是综合数据流加固和控制流加固方法，对目标程序进行双重加固，以此达到抗单粒子翻转的效果。

(3) 故障注入评估阶段

无论最终选择的是哪一种加固方法，均有对应的故障注入工具。使用相应的故障注入工具可以对加固后的目标程序进行故障注入实验，通过实验结果最终可以评估方法的有效性和对程序的适应性。

2.4 关键技术分析

为了实现 SEDHS-SEU 软件，本文突破了以下几项关键技术。

(1) 面向单粒子翻转的多位 SDC 脆弱性计算方法

在抗单粒子翻转的数据流错误检测及加固方法中，关键技术之一是针对指令的多位 SDC 脆弱性计算方法。当太空辐射环境中的计算机系统发生单粒子翻转时，有不低的概率引发存储器或者寄存器中的数据单元发生多位翻转。因此，需要针对单粒子多位翻转进行研究。这个问题目前尚未很好的解决，是一项有挑战性的关键技术。

(2) 指令传播性特征和固有性特征提取与计算方法

在抗单粒子翻转的数据流错误检测及加固方法中，关键技术之二是指令传播性特征和固有性特征的计算方法。在已有的研究工作中，对于指令特征的提取大多为较简单的特征罗列，并未深入挖掘深层次的传播性特征。解决这一问题需要结合程序切片技术，分析提取出对数据错误传播影响较大的指令特征。建立指令传播性特征和固有性特征提取与计算方法是整个数据流错误检测及加固方法的基础。

(3) 基本块按层次划分方法

在抗单粒子翻转的控制流错误检测及加固方法中，关键技术之一是基本块按层次划分方法。由于目标程序存在多样性，想要以统一的规则适应所有程序就需要研究一种对程序结构进行重

划分的方法。基本块按层次划分的目的就是将不同结构的目标程序进行统一处理，使其具备相似的程序结构，为标签的设置与分配建立基础。

(4) 多层分段标签方法

在抗单粒子翻转的控制流错误检测及加固方法中，关键技术之二是多层分段标签方法。控制流错误检测方法中双标签技术能够提高错误检测率，但同时有着较高的时空开销。为了节省成本，当前的控制流错误检测方法大多还是会选择单标签技术。本文提出的多层分段标签方法通过多层次，分数段的形式，使用单标签实现了双标签的检测效果，既节省了时空成本，又提高了错误检测率。

2.5 本章小结

本章主要分析了本文所实现的 SEDHS-SEU 软件的总体设计。对 SEDHS-SEU 软件进行需求分析，提出了五项需要实现的功能。提出 SEDHS-SEU 软件的总体设计和总体流程设计，详细分析整个软件的组织架构和整体逻辑设计，并给出了对目标程序进行加固的流程阶段和所有阶段的详细描述。

第三章 基于 SDC 多位脆弱性分析的数据流错误检测及加固方法

由单粒子翻转引起的数据流错误中最难检测的是一种无记载数据损坏(Silent Data Corruption, SDC)的错误。本文针对 SDC 问题，提出了一种检测及加固方法 SDCVA-OCSVM (Method for detecting SDC error using SDC vulnerability analysis and one-class support vector machine)旨在保护程序使用的存储器或寄存器中的数据。本章将对这一方法进行详细论述，并通过实验验证该方法的有效性。

3.1 数据流错误检测理论

3.1.1 单粒子多位翻转

在现有的抗单粒子效应加固方法中，对于单粒子翻转的考虑大多为单位翻转错误(single Bit Upsets, SBUs)[61]，即错误翻转的位数只有 1 位。然而，随着现代卫星集成化的提高，芯片已经进入了纳米级的时代，高能粒子进入纳米芯片后，极可能会在元件相邻的数据位上产生多个翻转错误。这种现象也被称为多位翻转错误(Multiple Bit Upsets, MBUs)[62]。参考文献[63]的相关实验及数据表明在某些航天器芯片中，多位翻转错误至少有 10%的发生率，这是一个无法忽视的概率。

S. Yoshimoto 等人选用 IDT7164 型号的 SRAM 作为实验元件，以 Kr 离子为辐射源进行了单粒子效应辐射实验[64][65]。实验中 Kr 离子的线性能量转移值$\mathrm{LET} = 41\mathrm{Mev}/(\mathrm{mg}/\mathrm{cm}^2)$，入射角度为 30°。最终测得 Kr 离子所造成的单粒子翻转现象中，单位翻转发生的概率为 83.3%，多位翻转概率高达 16.7%。在多位翻转事件中，两位翻转的概率为 82.63%，三位翻转概率为 15.97%，四位翻转概率为 0.73%，具体的发生形式如图 3.1 所示。

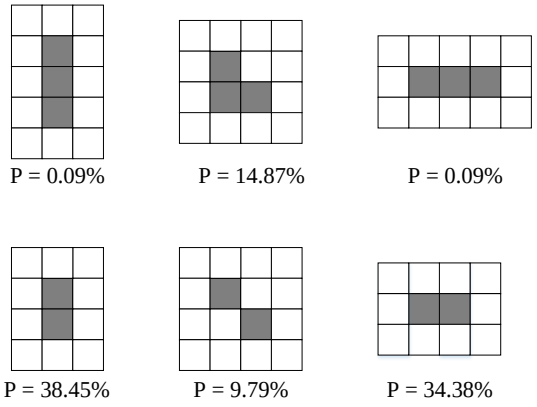


图 3.1 单粒子多位翻转发生形式及发生概率

从图 3.1 中我们可以看出，单粒子多位翻转存在着多种发生形式。多位翻转中上下相邻的

两个单位元同时发生翻转的概率最高，为 38.45%。左右相邻的两个单位元同时发生翻转的概率次之，为 34.38%。实验中这两种形式的翻转的发生概率之和超过了 70%，即有一大半的多位翻转均以这两种形式存在。两位翻转中还有一种交叉相邻的形式，发生概率为 9.79%。实验中连续三个横向相连或者竖向相连的单位元，即横向三位翻转和竖向三位翻转同时发生的概率反而不高，均不到 1%。反倒是“L”型翻转的概率是三位翻转占比最高的，为 14.87%。综合此实验中的所有结果，可以发现单粒子翻转中的所有多位翻转只会发生在相邻的单位元中，而且会以某种特定的形式存在，这为数据流错误检测方法提供了重要参考。

3.1.2 典型单粒子单位翻转检测方案分析

现有的数据流错误检测技术中，大多考虑的是单粒子单位翻转。典型工作如：Liu 等人于 2019 年提出的 SDCPredictor 方法就是一种只考虑单粒子单位翻转的方法[38]。下面本文将对其进行详细介绍。

SDCPredictor 属于一般典型的数据流错误检测技术，它旨在建立一个预测模型，使得该模型能够较为准确的预测指令 SDC 脆弱性，尽量减小误差。为此，SDCPredictor 提取了一些指令的动态以及静态特征，并借助故障注入实验创建了训练数据。基于训练数据集，SDCPredictor 使用随机回归森林构建预测模型。最终通过预测模型识别目标程序中 SDC 脆弱性高的指令。

为了得到训练样本，SDCPredictor 借助了故障注入工具 PINFI 进行故障注入实验。在实验过程中，将故障注入进选定的指令中。为了模拟单粒子翻转，SDCPredictor 对指令的数据单元的某一位进行翻转且只翻转一次。在每次运行中，将故障（即一位翻转）注入动态指令实例的数据单元中。通过使用相同的输入执行原始可执行程序获得无故障的结果，将故障结果与无故障结果进行比较以完成故障注入实验。从上述过程可以看出，SDCPredictor 在模拟单粒子翻转时，并未考虑到多位翻转的情况。

SDCPredictor 中提出了一种名为 SDC 倾向性的定义，表示指令错误导致 SDC 的可能性。计算方法如式(3-1) [38]所示[38]：

$$P(SDC)=\frac{N_{SDC}}{N_{fault}}\times D(I) \tag{3-1}$$

其中，N_{SDC} 表示通过故障注入实验得到的所有故障中 SDC 的次数，N_{fault} 为初始故障总数。$D(I)$ 是指令 I 的动态计数比，指的是执行的动态指令实例数与程序中动态指令总数之比。这是 SDCPredictor 的核心定义，而从式(3-1)中可以看出，SDC 倾向性的计算需要大量使用故障注入工具。由于 SDCPredictor 的故障注入工具并不具备模拟单粒子多位翻转的功能，故 SDCPredictor 无法检测单粒子多位翻转。

3.1.3 单粒子多位翻转检测方案分析

现有技术中对单粒子多位翻转的研究与讨论主要集中在基于硬件的错误检测及加固领域。Santos 等人于 2017 年提出在硬件加速器中应用三模冗余技术，以减轻 SRAM 受到单粒子多位翻转的影响[85]。同年，Appathurai 等人提出了一种 CMC 方法，该方法能够检测并校正内存受单粒子多位翻转影响而引起的错误[86]。从上述研究可以看出，硬件加固领域已将单粒子多位翻转作为常态问题加以考虑，但目前主流的数据流冗余断言检测技术大多未考虑到多位翻转问题。如上一节提到的 SDCPredictor 方法就未考虑单粒子多位翻转发生时的情况。除此之外，wang 等人于 2019 年提出的 PVInsiden 方法[75]也存在对单粒子多位翻转现象考虑不足的问题。从操作层面上看，研究单粒子多位翻转问题也要比只考虑单粒子单位翻转问题更难一些。首先单粒子多位翻转现象比单粒子单位翻转现象更为随机，其发生形式相比单位翻转更加多样。其次，对于多位翻转造成的软错误在程序中传播的研究也更具难度。最后，目前主流的单粒子翻转故障注入仿真工具鲜有考虑到多位翻转的情况，缺乏直接可用的工具。

综合考虑单位与单粒子多位翻转问题是实现单粒子翻转检测的基础，本文在设计单粒子多位翻转检检测方案时着重解决了以下几个问题：

(1) 单粒子多位翻转 SDC 脆弱性的计算方法

单粒子多位翻转 SDC 脆弱性的计算方法是进行抗单粒子翻转数据流错误检测及加固的基础，在已有的研究工作中，对 SDC 脆弱性的计算仍停留在单位翻转的层面，而对多位翻转鲜有成果。为解决这一问题，本文参考了大量文献，分析并总结了单粒子多位翻转的发生情况。提出了面向单粒子多位翻转的 SDC 脆弱性计算方法。

(2) 单粒子多位翻转对数据错误传播的影响

由于数据错误在程序中具有传播性，单位翻转和多位翻转对错误在程序中的传播有不同的影响，因此本文分析了多位翻转对错误传播的影响，提出了指令传播性特征的提取与计算方法。数据错误在程序中传播有时会导致整个程序崩溃，有时却不影响程序正常执行，这是因为数据错误在程序传播过程中有一定概率会被屏蔽。根据这一特点以及单粒子多位翻转损伤机理，本文提出了屏蔽特征及相应的特征提取计算方法，进一步分析了数据错误在程序中的传播性，提高了数据流错误检测及加固方法的有效性。

(3) 故障注入实验平台

在面向单粒子翻转引发的数据流错误检测方法中，故障注入工具是必要的。在进行方法性能评估和特征提取的过程中，均需要使用故障注入工具。但当前主流的模拟计算机发生单粒子翻转导致数据流错误的开源仿真故障注入工具中，并不支持用户对数据单元进行一次性多位翻转，因此需要根据单粒子多位翻转的特性进行定制化开发。本文基于 GDB 进行二次开发，搭建了能够实现单粒子多位翻转故障注入的实验平台。

(4) 基于 SDC 多位脆弱性分析的数据流错误检测及加固方法

本文基于对 SDC 脆弱性的分析和对单粒子翻转问题的研究，提出了数据流错误检测及加固方法。通过对故障注入实验结果以及程序结构的相关分析，提取出了指令的传播性和固有性特征，建立了指令识别模型和冗余加固策略，最终实现了面向单粒子翻转的数据流错误检测及加固。

3.2 高 SDC 多位脆弱性指令识别

3.2.1 相关定义

本文中主要针对的是空间辐射环境中的单粒子翻转问题，而由这一现象所引发的 SDC 错误也与单粒子翻转损伤机理有一定的关系。在本文中所提出的指令 SDC 多位脆弱性正是结合了 3.1.1 节中单粒子翻转现象研究而提出的。指令 SDC 多位脆弱性是指当单粒子翻转发生在指令所使用或者存储的数据中时，程序发生 SDC 错误的概率。具体定义如下所示。

程序指令集合(I_{sset})：程序指令集合是指目标程序经过 Clang 编译器编译后生成的中间代码文件中所包含的全体指令，如式(3-2)所示。

$$I_{sset}=\{I_1,...,I_i,...,I_{size(I_{sset})}\} \tag{3-2}$$

其中 I_i 表示在目标程序中的第 i 条指令。$size()$ 表示括号内集合的元素个数，如 $size(I_{sset})$ 表示的就是程序内指令数目。

程序动态指令集合(I_{dset})：指令在程序运行时往往会执行多次，而每一次执行时受到单粒子翻转的影响也不一样。故定义程序动态指令集合 I_{dset}，以程序指令执行次数为基准，令 n_i 代表第 i 条指令的执行次数，则动态指令集合可表示为式(3-3)。

$$I_{dset}=\{I_1^1,I_1^2,...,I_1^{n_1},...,I_i^1,I_i^2,...,I_i^{n_i},...,I_{size(I_{sset})}^1,I_{size(I_{sset})}^2,...,I_{size(I_{sset})}^{n_{size(I_{sset})}}\} \tag{3-3}$$

数据流错误翻转概率：单粒子翻转现象引起单位数据流错误翻转的概率，记为 P_{SBU}。单粒子翻转现象引起横向二位数据流错误翻转的概率，记为 P_{H2BU}。单粒子翻转现象引起竖向二位数据流错误翻转的概率，记为 P_{V2BU}。单粒子翻转现象引起交叉二位数据流错误翻转的概率，记为 P_{C2BU}。单粒子翻转现象引起横向三位数据流错误翻转的概率，记为 P_{H3BU}。单粒子翻转现象引起竖向三位数据流错误翻转的概率，记为 P_{V3BU}。单粒子翻转现象引起 L 型三位数据流错误翻转的概率，记为 P_{L3BU}。而参考单粒子翻转辐射实验结果，发生四位以上翻转的概率不足 1%，故本文未将其纳入讨论范围。

指令多位 SDC 脆弱性(P_{ecSDC})：指令多位 SDC 脆弱性是指当前指令所使用的数据受到单粒子翻转的影响时，程序发生 SDC 错误的概率，以 $P_{ecSDC}(I_i)$ 表示。脆弱性需要依靠数据流故障注入实验获得。n_i 代表程序中静态指令 I_i 的动态执行次数。设 R 代表向指令中注入故障的总次数，则 $R_{SBU}(I_i^k)$ 代表的就是向动态指令 I_i^k 注入单粒子单位翻转故障的总次数。依此类推，

$R_{H2BU}(I_i^k)$，$R_{V2BU}(I_i^k)$，$R_{C2BU}(I_i^k)$，$R_{H3BU}(I_i^k)$，$R_{V3BU}(I_i^k)$，$R_{L3BU}(I_i^k)$分别代表向动态指令I_i^k注入横向二位翻转故障、竖向二位翻转故障、交叉二位翻转故障、横向三位翻转故障、竖向三位翻转故障、L 型三位翻转故障的总次数。T 代表向指令中注入故障后造成 SDC 错误的次数，同理，$T_{SBU}(I_i^k)$，$T_{H2BU}(I_i^k)$，$T_{V2BU}(I_i^k)$，$T_{C2BU}(I_i^k)$，$T_{H3BU}(I_i^k)$，$T_{V3BU}(I_i^k)$，$T_{L3BU}(I_i^k)$分别代表对应类型的故障注入后造成 SDC 错误的次数。P_{SBU}，P_{V2BU}，P_{H2BU}，P_{C2BU}，P_{H3BU}，P_{V3BU}，P_{L3BU}为之前定义的单粒子翻转引发每种数据流错误模型的概率，可以使用式(3-4)对指令多位 SDC 脆弱性进行计算。

$$P_{ecSDC}(I_i)=\frac{1}{n_i}\sum_{k=1}^{n_i}P_{ecSDC}(I_i^k)=\frac{1}{n_i}\sum_{k=1}^{n_i}(P_{SBU}\times\frac{T_{SBU}(I_i^k)}{R_{SBU}(I_i^k)}+P_{H2BU}\times\frac{T_{H2BU}(I_i^k)}{R_{H2BU}(I_i^k)}+P_{V2BU}\times\frac{T_{V2BU}(I_i^k)}{R_{V2BU}(I_i^k)}$$

$$+P_{C2BU}\times\frac{T_{C2BU}(I_i^k)}{R_{C2BU}(I_i^k)}+P_{H3BU}\times\frac{T_{H3BU}(I_i^k)}{R_{H3BU}(I_i^k)}+P_{V3BU}\times\frac{T_{V3BU}(I_i^k)}{R_{V3BU}(I_i^k)}+P_{L3BU}\times\frac{T_{L3BU}(I_i^k)}{R_{L3BU}(I_i^k)}) \quad (3\text{-}4)$$

易引发 SDC 错误的指令(I_{ecSDC})： I_{ecSDC} 是程序中某些指令一旦受到了单粒子翻转的影响，就容易引发程序发生 SDC 错误。在本文中，当某一指令的多位 SDC 脆弱性 P_{ecSDC} 超过阈值时，该指令就可称为 I_{ecSDC} 指令。

3.2.2 高多位 SDC 脆弱性指令识别模型

高多位 SDC 脆弱性指令识别模型的主要作用是识别出程序中 I_{ecSDC} 指令，将程序指令集中 I_{ecSDC} 指令和正常指令进行分类，故需要训练一个能够区分二者的分类器。从样本数据的特征来看，导致 SDC 错误的指令与正常指令较难区分，有的甚至会被分类器混淆成一类。所以一个在二分类问题上性能更为优秀的检测模型是必要的。

单类支持向量机（One-Class Support Vector Machine，OCSVM）[68]算法是在传统 SVM 算法上进行扩展的一种无监督学习算法，为了更好适应本文中样本数据的特点并满足指令检测的性能需求，我们选择了单类支持向量机 OCSVM，主要有以下四点原因：

(1)指令特征数据主要由 bool 类型和 int 类型组成，SVM 可以应用于这种混合特征数据类型。而将 int 类型的数据归一化处理后，单类支持向量机也可以完全适应这种混合数据。特征数据经过标准化处理后，原始特征转化为无量纲化指标测评值，各指标值均处于同一数量级别，方便 OCSVM 进行综合测评。

(2)单类支持向量机的特征参数经过归一化后可直接使用，不需要过多的调整。建立模型过程也较为简单，无需太多繁琐的操作。

(3)单类支持向量机本身对于二分类问题具有较好的性能，被广泛用于各种分类预测问题，在软错误领域也是一种杰出的分类模型[54][75]。

(4)单类支持向量机和传统的支持向量机相比对训练数据集的要求较低，同时也具备了支持向量机算法速度快、泛化能力强等优点，有利于实际应用。

指令样本空间由m组形如$(\vec{F}_1, L_1)$,$(\vec{F}_2, L_2)$,…, $(\vec{F}_m, L_m)$的数据组成。这其中$\vec{F}_j$是一个多维实数向量，涵盖了提取出的指令特征。L_j为样本类型标签，代表了该条数据所对应指令的类型。设指令 SDC 多位脆弱性阈值为S，若多位脆弱性超过了阈值，则将该样本所对应的指令判别为I_{ecSDC}指令，同时将该指令的标签变为+1；反之，则判别为普通指令，标签值变为-1，如式(3-5)所示。

$$L=\begin{cases}+1,\ P_{ecSDC}(I_i)\geq S\\ -1,\ P_{ecSDC}(I_i)<S\end{cases} \tag{3-5}$$

根据 OCSVM 理论，建立指令分类模型，如式(3-6)[69]所示：

$$f(F)=\mathrm{sgn}(\omega\cdot\varphi(F)-\rho) \tag{3-6}$$

$f(F)$为单类支持向量机划分指令的分类函数，其中F为多维实数向量，ω为划分超平面的法向方向，ρ代表坐标原点相对于平面的偏移值，因此$\rho\|\omega\|$代表坐标原点到超平面的距离。将其变为函数求解的最优化问题如式(3-7)[70]：

$$\begin{aligned}&\min_{\omega\in F,\rho\in R,\xi\in R^l}\frac{1}{2}\|\omega\|^2+\frac{1}{vn}\sum_i\xi_i-\rho\\&s.t.(\omega\cdot\phi(F))\geq\rho-\xi_i,\xi_i\geq 0\end{aligned} \tag{3-7}$$

(3-6)式为单类支持向量机的基本型，其本身是一个凸二次规划。其中ξ_i表示“松弛变量”，一般是指损失函数。其中n为样本数量，$v\in(0,1)$为惩罚系数。为了有效求解该问题，需要引入拉格朗日乘子法，拉格朗日函数如式(3-8)[70]所示：

$$Lagrange(\omega,\rho,\alpha,\xi,\mu)=\frac{1}{2}\|\omega\|^2+\frac{1}{vn}\sum_i\xi_i-\rho-\sum_i a_i\left((\omega\cdot\Phi(F_i))-\rho+\xi_i\right)-\sum_i\mu_i\xi_i \tag{3-8}$$

其中$a_i\geq 0,\mu_i\geq 0$为拉格朗日乘子。将拉格朗日函数分别对 ω , ρ 和 ξ 求偏导，并令导数为 0，求出结果带入拉格朗日函数，再考虑相关约束，最终得到式(3-9)所示的对偶问题。

$$\begin{aligned}&\min_\alpha\frac{1}{2}\sum_{ij}\alpha_i\alpha_j\Phi(F_i)\Phi(F_j)=\min_\alpha\frac{1}{2}\sum_{ij}\alpha_i\alpha_j k(F_i,F_j)\\&s.t.\ \sum_i a_i=1,\ 0\leq\alpha_i\leq\frac{1}{vn}\end{aligned} \tag{3-9}$$

其中$k(F_i,F_j)$表示关于F的核函数，本文选择高斯核函数：$K\left(F_i,F_j\right)=exp\left(-\frac{\|F_i-F_j\|^2}{2\sigma^2}\right)$。解出问题(3-9)可以得到拉格朗日乘子$\alpha$，最终可得到超平面模型，如式(3-10)所示：

$$f(F)=\mathrm{sgn}(\sum_i\alpha_i k(F_i,F_j)-\rho) \tag{3-10}$$

上述多位 SDC 脆弱性指令识别模型可以对程序中的指令进行分类，区分正常指令和I_{ecSDC}指令，同时需要考虑的主要参数有两个，惩罚因子v和核函数中的σ。为了有效的对指令进行分类，需要上述两参数进行优化，下一节对参数优化方法展开讨论。

3.2.3 参数优化

为了提高指令识别模型的性能，对模型的参数进行优化以提高指令识别模型的判别能力。在本文提出的基于 OCSVM 的指令识别模型中，需要提前设置的参数有式(3-7)中的惩罚因子 v，核函数中的 σ。正则化参数 v 为检测模型对异常点的容忍度，高斯核函数带宽 σ 控制函数的径向作用范围。由于这两个参数的组合 $c=(v,\sigma)$ 的取值将会限制指令识别模型的性能，因此本文将这个参数选择问题当作最优化问题处理。由于灰狼优化算法[71][72][73]具有较强的收敛性能、参数少、易实现等优点，因此本文使用灰狼优化算法来解决指令识别模型参数寻优问题。

根据灰狼算法优化 OCSVM 参数主要分为以下几个步骤：

1) 选择几个初始化参数组合，$\{c_1,c_2,...c_i,...,c_n|c_i=(v_i,\sigma_i)\}$，初始化系数向量 $\vec{A}$ 和 $\vec{B}$。

2) 依据训练样本及相关参数训练 OCSVM 模型，同时计算适应度 fit，适应值为参数适应判断标准，计算方法如式（3-11）所示：

$$fit=1-\frac{num_f}{num_t+num_f} \tag{3-11}$$

其中，num_t 为模型分类正确的指令数目，num_f 为模型分类错误的指令数目，fit 值越高，越接近最优值。根据所有参数位置的适用度选出 α,β,δ 三项较优先的参数组合，剩下的均为 ω 参数组合。

3) 根据所有位置更新公式(3-12)至(3-16)，更新参数组合的位置。

$$\vec{D}=\left|\vec{B}\cdot\vec{c}_p(t)-\vec{c}(t)\right| \tag{3-12}$$

$$\vec{c}(t+1)=\vec{c}_p(t)-\vec{A}\cdot\vec{D} \tag{3-13}$$

$$\vec{D}_\alpha=\left|\vec{B}_1\cdot\vec{c}_\alpha(t)-\vec{c}(t)\right|,\vec{D}_\beta=\left|\vec{B}_2\cdot\vec{c}_\beta(t)-\vec{c}(t)\right|,\vec{D}_\delta=\left|\vec{B}_3\cdot\vec{c}_\delta(t)-\vec{c}(t)\right| \tag{3-14}$$

$$\vec{c}_1(t)=\left|\vec{c}_\alpha(t)-\vec{A}_1\cdot\vec{D}_\alpha\right|,\vec{c}_2(t)=\left|\vec{c}_\beta(t)-\vec{A}_2\cdot\vec{D}_\beta\right|,\vec{c}_3(t)=\left|\vec{c}_\delta(t)-\vec{A}_3\cdot\vec{D}_\delta\right| \tag{3-15}$$

$$\vec{c}(t+1)=\frac{\vec{c}_1(t)+\vec{c}_2(t)+\vec{c}_3(t)}{3} \tag{3-16}$$

其中，$\vec{c}=(v,\sigma)$ 为待优化的参数位置。$\vec{c}_p$ 代表参数组合的最优解。t 为当前迭代次数，最大迭代次数为 $t_{\max}$。$\vec{D}$ 代表迭代过程中，前一迭代轮次的参数组合最优解和当前候选参数组合之间的差值。$\vec{D}_\alpha$，$\vec{D}_\beta$，$\vec{D}_\delta$ 分别表示当前候选参数组合和前一轮次最优的三项参数组合之间的差值。$\vec{A}$ 和 $\vec{B}$ 为系数向量用以更新参数组合的值。系数向量的计算式如(3-17)至(3-19)所示：

$$\vec{A}=2\vec{a}\cdot\vec{r}_1-\vec{a} \tag{3-17}$$

$$\vec{B}=2\cdot\vec{r}_2 \tag{3-18}$$

其中，$\vec{r}_1$ 和 $\vec{r}_2$ 是[0,1]的随机向量。$\vec{a}$ 是从 2 线性下降至 0 的分量，如式(3-19)所示：

$$\vec{a}=2-2t/t_{\max} \tag{3-19}$$

4) 计算更新后所有参数组合的适应值，比较这一轮运动中参数组合适应值是否增加，若增大则更新参数组合的位置，否则参数组合保持原位置不动。位置优化后再次重新分配阶级，适应值前三名升级为新的α,β,δ优先组合参数。在迭代过程中要始终保持其处于最优位置。

5) 确定迭代次数是否达到设置的阈值或当前全局最佳位置是否满足最小限制。如果满足条件，则终止循环，并返回最佳参数组合(v_{best},σ_{best})。否则，处理返回到步骤 4)。

6) 通过最佳的参数组合(v_{best},σ_{best})，建立 OVSVM 模型。

3.2.4 模型训练样本选择

为了获得高质量的训练样本，还需确定训练数据集的来源。本文使用了一种部分故障注入方式，从目标程序中抽取 30%的指令用作训练样本，从而训练机器学习模型。为了确认部分故障注入方式的效果，我们进行了实验验证。这项实验主要目的是探究部分故障注入的指令抽取比例与检测器性能之间的关系。实验中选择精确率(precision)、召回率(recall)以及 f-score 三项指标作为衡量标准，这三项指标被广泛应用于分类器的性能评估，有着较好的评估效果。从目标程序中抽取一定比例的指令，再针对抽出的指令提取特征并进行故障注入实验进行分类以建立训练数据样本，训练检测器，再使用检测器对目标程序剩余的指令进行判别。这里我们使用快速排序 qsort.c 作为实验对象，得到的实验结果如图 3.2 所示。

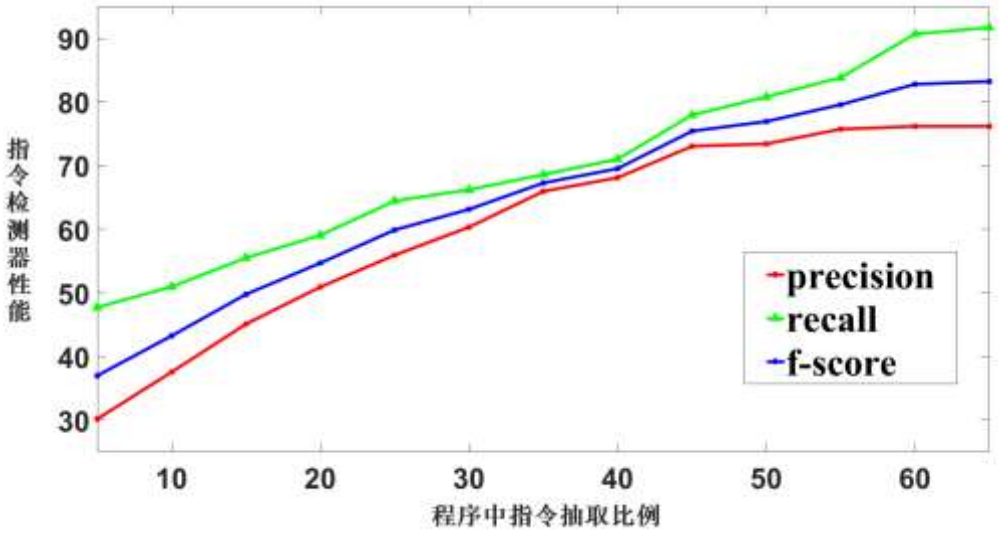


图 3.2 程序中指令抽取比例与指令检测器性能之间的关系

从图 3.2 中，我们发现准确率并未按照理想状态下在 30%比例时有特别明显的变化趋势，而是在 35%左右开始逐渐平稳。召回率则一直呈上升趋势，将二者综合的 f-score 也大体上呈现出了上升趋势。另一方面，训练后的指令检测器性能也不理想，即使抽取了 65%指令对检测器进行训练，检测准确度也仅为 76.2%。经过对实验的深入分析，我们认为造成此结果的原因主

要有以下几点：(1)训练样本不足。实验中的目标程序 qsort.c 中经过 clang 编译成 LLVM 中间文件后只有 157 条指令，再按比例抽取后也仅能形成不到一百个样本。(2)特征应用不全。qsort.c 程序规模较小，很多指令特征并没有在这次实验中得到很好的体现，这说明样本类型也不够丰富。(3)程序具有灵活性。不同的程序的指令对检测器有不同的训练效果，若单纯以 30%随机抽取为基准不能适用于所有程序。因此，我们需要构建一种数量足够，类型丰富并且具有一定普适性的训练样本。故在训练集中除了抽取目标程序的部分指令外，还需添加其它程序作为训练样本的补全。

为了获得理想的训练样本，我们从嵌入式基准测试集 mibench[83]中选择了几个具有代表性的程序作为基本训练样本的指令来源。又根据训练样本指令特征涵盖情况编写了 PolNom.c 程序作为补充，具体训练程序如表 3. 1 所示。选定好训练样本，完整检测器训练过程如下：先将基本训练程序转化为训练样本，再根据目标程序大小抽取一定比例的指令，然后通过故障注入实验同样转化为样本数据，最后将二者结合组成训练样本空间，以此来训练指令检测器。

表 3. 1 基本训练程序

名称	描述
basicmath	该程序中包含各种基础数学运算操作，可以大量使用指令运算相关特征。
bitcnts	mibench 中用于测试 CPU 位计算能力的测试集
jpeg	JPEG 图像的编解码程序
ispell	用于快速拼写检查的程序
stringsearch	用于查找字符串的程序
sha	SHA 散列算法
MM	执行矩阵乘法运算的程序
CRC32	CRC32 的计算程序
PolNom	一种特定多项式求解及打印程序，主要涉及内存及堆栈的使用

针对每一个目标程序，数据流错误检测与加固方法中的指令识别模型将会在固定训练样本的基础上，不仅使用表 3. 1 的程序集，还会使用部分故障注入的形式，提取目标程序中的部分指令作为训练样本对模型进行动态训练。通过这样不断更新训练样本，重新训练模型的形式，尽可能地减少概念漂移对方法产生的不良影响。

3. 3 指令特征提取

SDC 是一种较为特殊的错误，它虽然会改变程序的数据流，但不会对程序的执行造成严重损害或者引发系统异常。本文在进行指令多位 SDC 脆弱性研究时，考虑了 SDC 的两个基本特

点。第一点，程序产生 SDC 错误一定是因为错误发生在了一些可以左右程序输出结果的指令上。错误有可能是由单粒子翻转直接引起，也有可能是经由数据流传播导致的。第二点，故障注入时导致的 SDC 结果具有互斥性。当程序发生系统错误或者崩溃时，程序就不会再引发 SDC 错误，同理，SDC 错误也不会进一步的导致系统错误或者悬挂崩溃。这两个特点，前者有助于提取 SDC 的指令相关特征，后者帮助我们计算指令 SDC 多位脆弱性。

3.3.1 提取指令传播性特征

SDC 错误的产生有可能是错误通过指令间的数据交互，最终作用到了可以左右输出程序结果的指令上。由此可看出，指令的多位 SDC 脆弱性会在一定程度上受到数据在程序内传播关系的影响。单粒子翻转发生时产生的错误数据有时并不会立刻影响到程序执行，往往要经过传播后才会导致程序发生 SDC 错误，这种现象本文称其为传播性 SDC 错误。为了研究传播性 SDC 错误，需要对数据在程序中的传播进行分析，并且可以出提取出一些和数据在指令中传播相关的特征。对这一类特征，本文称其为指令的传播性特征。本文从数据传播和错误屏蔽两个方面对指令传播性特征进行分析。

3.3.1.1 数据传播特征分析

数据传播特征与时间、数据位数、数据类型等要素有关。对于程序中所使用的数据变量而言，生命周期是其在传播过程中非常重要的特征之一。由于空间环境中发生单粒子翻转现象具备随机性，变量生命周期越长就越容易受到单粒子翻转影响。计算机中时间统计的精度存在限制，指令的执行时间太短，用有限的时钟周期有时很难精确统计出指令执行时间。因此本文使用指令执行次数作为执行时间的模拟替代。我们将数据变量在程序中的生命周期定义为变量在内存中分配的执行时间段，具体为从第一次调用该变量开始，到最后一次调用该变量结束。数据变量生命周期具体计算如式(3-20)所示。

$$life(num) = life_{end}(num) - life_{begin}(num) \tag{3-20}$$

其中，$life_{end}(num)$ 和 $life_{begin}(num)$ 为程序最后一次调用变量和第一次调用该变量时程序已执行的动态指令条数。

包括生命周期在内，本文将数据变量中和 SDC 错误传播有关的特征记为 F_{data}，并统计出了一个四元组，如式(3-21)所示。

$$F_{data} = \langle life(num), width(num), is\_\text{int}, is\_float \rangle \tag{3-21}$$

其中，$life(num)$ 为变量生命周期，$width(num)$ 为数据变量的二进制位数，is_int 表示判断当前数据变量类型是否为整型，is_float 为判断数据变量是否为浮点类型。数据变量的位数影响着单粒子翻转时的影响范围，不同位数的数据受到错误翻转的影响是不同的。同理，错误翻转对不同类型的数据影响也是不同的，故将整型和浮点型的区别也纳入了指令特征的考量。

除去上述关于数据变量本身的特征之外，一些和指令操作相关的特征也会影响到数据错误在程序中的传播。程序中一般存在着进行运算操作的指令，而和指令运算相关的特征是影响数据错误传播的重要因素。在程序中这类指令特征有时可以屏蔽错误，故此类特征搭配下一节中的屏蔽特征可以对数据错误传播有更好的描述。针对这一类特征，本文将其记为 F_{OP}，具体表达如(3-22)所示。

$$F_{OP}=\langle is\_and\_op, is\_or\_op\rangle \tag{3-22}$$

式(3-22)中，is_and_op 代表对运算类型的判断，确认其是否是与运算，若是则相应字段标记为 1，否则标记为 0。is_or_op 会判断运算指令是否为或运算，同样用 0 和 1 进行区分。

除此之外，比较指令相关特征也是影响数据错误传播的一类重要特征。比较指令在程序执行过程中决定了程序的流程方向，若出错容易导致程序错误跳转，发生故障。另一方面，比较运算的结果是否正确有着较为宽松的判定。例如在判定 x>y 的过程中，若受到了单粒子翻转的影响，结果并非一定产生变化，故此类特征可同样配合屏蔽特征数据错误传播。针对这一类特征，本文将其记为 F_{CMP}，具体表达如(3-23)所示。

$$F_{CMP}=\langle is\_cmp, is\_icmp, is\_fcmp\rangle \tag{3-23}$$

式(3-22)中，is_cmp 代表对指令类型的判断，确认其是否是比较指令，若是则相应字段标记为 1，否则标记为 0。is_icmp 会判断比较指令中的两个操作数是否均为整型，同样用 0 和 1 进行区分。is_fcmp 会判断比较指令中的操作数是否含有浮点型，也使用 0 和 1 区分。

3.3.1.2 屏蔽特征分析

从指令的角度来说，单粒子翻转造成的数据错误有可能在传播过程中被屏蔽了[74]。这是影响数据错误最终是否能造成 SDC 错误的关键因素之一。为此本文设计了一种专门评估在程序中会屏蔽错误的屏蔽概率参数 P_{mask}，其中主要涉及到逻辑运算和比较运算。在具体分析过程中，我们分析单粒子翻转的效果，并确定错误屏蔽的条件，推导出屏蔽概率参数 P_{mask}。使用程序切片分析技术分析，基于数据变量建立程序数据流指令集合，根据数据流指令集合和 P_{mask} 进一步计算出每一条指令的屏蔽特征 f_m。

程序在运行过程中，当数据发生错误时，运算操作指令有一定概率会掩盖数据错误，本文将这一概率称为运算指令的屏蔽概率参数 P_{mask}，对于不具备屏蔽数据错误能力的指令，其屏蔽概率参数的值为 0。另一方面，虽然并非所有的指令都有掩盖错误的能力，但由于数据在程序中具有传播性，若某指令的切片中包含具有概率屏蔽参数的指令，也有概率将数据错误屏蔽，本文将这一概率称为指令的屏蔽特征 f_m。

在计算 P_{mask} 之前，仍需定义一些函数及相关变量。设函数 $f(x)$ 表示二进制数 x 的总位数，例如若 x 为 32 位数，则 $f(x)=32$。设 w_1^x 和 w_0^x 分别代表二进制数 x 中 1 和 0 的位数，两函数

具体的计算方法如(3-24)和(3-25)所示。

$$w_1^x(b_1,b_2)=\sum_{k=b_1}^{b_2}x_k \tag{3-24}$$

$$w_0^x(b_1,b_2)=\sum_{k=b_1}^{b_2}(1-x_k) \tag{3-25}$$

式中，函数 $w_1^x(b_1,b_2)$ 代表二进制数 x 中，第 b_1 位至第 b_2 位中 1 的总位数，$w_0^x(b_1,b_2)$ 代表 x 中第 b_1 位至第 b_2 位中 0 的总位数，其中 $1\le b_1<b_2\le f(x)$，k 代表二进制数 x 中的第 k 位。

设 Z 为二进制数中连续相同数位数的集合。$Z_2^0(x)$ 代表 x 中连续 2 位以上均为 0 的位数集合，集合内元素从右至左统计。$Z_3^0(x)$ 代表 x 中连续 3 位以上均为 0 的位数集合，$Z_2^1(x)$ 代表 x 中连续 2 位以上均为 1 的位数集合，$Z_3^1(x)$ 代表 x 中连续 3 位以上均为 1 的位数集合。若设 x 为 32 位二进制数 00100111011110001000111111110111，则 $Z_2^0(x)$= {3，3，：，$Z_3^0(x)=\{3,3\}$，$Z_2^1(x)=\{3,8,4,3\}$，$Z_3^1(x)=\{3,8,4,3\}$。

首先讨论逻辑运算相关指令中屏蔽概率参数 P_{mask} 的计算。与运算和或运算是较为常见的逻辑运算，指令中包含的 AND x,y (或者 OR x,y)就是指数据 x 按位与(或者按位或) y 的逻辑计算。设 x 为受到单粒子翻转影响的数据，则在与运算中 y 对应的位为 0 时，就可以将 x 中受到的错误屏蔽下来，而或运算则是 y 对应的位为 1 时，就可以屏蔽错误。要计算此时指令的 P_{mask}，需要对 AND 和 OR 运算的双方数据进行分析，同时也要考虑到多位翻转的情况，下面分情况进行讨论：

(1) 按位与运算

若指令中存在两数 x 与 y 进行按位与运算如 AND x,y。设单粒子翻转现象发生在 x 或者发生在 y 上的概率是随机且均匀的。为计算屏蔽概率参数 P_{mask}，需要计算单位屏蔽概率参数 p_{mask}^1，双位屏蔽概率参数 p_{mask}^2 以及三位屏蔽概率参数 p_{mask}^3。

p_{mask}^1 为单粒子翻转现象错误翻转了 1 位数据时的屏蔽概率，为了计算该值，需要考虑二进制数 x 与 y 所有的位数中，翻转后错误会被屏蔽的位数。与运算中，操作数的 0 位会屏蔽另一操作数对应位的翻转错误。故只需统计操作数中 0 位的数量占总位数的比率，就可算出 1 位屏蔽概率参数，计算式如(3-26)所示。

$$p_{mask}^1=\frac{w_0^x(1,f(x))+w_0^y(1,f(y))}{f(x)+f(y)} \tag{3-26}$$

其中 $w_0^x(1,f(x))$ 代表数 x 中 0 位的个数，$w_0^y(1,f(y))$ 代表数 y 中 0 位的个数，$f(x)$ 代表数 x 的总位数，$f(y)$ 代表数 y 的总位数，在与运算中只统计操作数中 0 位便可计算屏蔽概率参数，故只用了 0 位相关函数。与运算中的 2 位屏蔽概率参数与 3 位屏蔽概率计算参数的计算方法类似，只需统计操作数中连续两位均为 0 位的个数和连续三位均为 0 位的个数即可，计算式如(3-27)和(3-28)所示。

$$p_{mask}^{2}=\frac{\sum_{l=1}^{size(Z_2^0(x))}(Z_2^0(x)(l)-1)+\sum_{l=1}^{size(Z_2^0(y))}(Z_2^0(y)(l)-1)}{f(x)+f(y)-2} \tag{3-27}$$

$$p_{mask}^{3}=\frac{\sum_{l=1}^{size(Z_3^0(x))}(Z_3^0(x)(l)-2)+\sum_{l=1}^{size(Z_3^0(y))}(Z_3^0(y)(l)-2)}{f(x)+f(y)-4} \tag{3-28}$$

其中，$Z_2^0(x)$ 和 $Z_3^0(x)$ 分别代表 x 中连续 2 位和连续 3 位为 0 的位数集合，$Z_2^0(y)$ 和 $Z_3^0(y)$ 分别代表 y 中连续 2 位和连续 3 位为 0 的位数集合，l 代表集合中第 l 项元素，$size()$ 表示括号内集合的元素个数。通过 p_{mask}^1，p_{mask}^2，p_{mask}^3 可以计算屏蔽概率参数 P_{mask}，计算公式如(3-29)所示。

$$P_{mask}=(P_{SBU}+P_{V2BU}+P_{V3BU}+P_{C2BU})\cdot p_{mask}^1+(P_{H2BU}+P_{L3BU})\cdot p_{mask}^2+P_{H3BU}\cdot p_{mask}^3 \tag{3-29}$$

P_{SBU} 为单粒子翻转现象引起单位数据流错误翻转的概率，P_{H2BU} 为横向二位数据流错误翻转的概率，P_{V2BU} 为竖向二位数据流错误翻转的概率，P_{C2BU} 为交叉二位数据流错误翻转的概率，P_{H3BU} 为横向三位数据流错误翻转的概率，P_{V3BU} 为竖向三位数据流错误翻转的概率，P_{L3BU} 为引起 L 型三位数据流错误翻转的概率。而参考单粒子翻转辐射实验结果，发生四位以上翻转的概率不足 1%，故本文未将其纳入讨论范围，(3-29)式还会出现多次，下文不再对其做单独说明。

(2) 按位或运算

指令中 x 与 y 进行按位或运算，设单粒子翻转现象发生在 x 或者发生在 y 上的概率是随机且均匀的。或运算中，操作数的 1 位会屏蔽另一操作数对应位的翻转错误。故只需统计操作数中 1 位的数量占总位数的比率，就可算出 1 位屏蔽概率参数，计算式如(3-30)所示。

$$p_{mask}^{1}=\frac{w_1^x(1,f(x))+w_1^y(1,f(y))}{f(x)+f(y)} \tag{3-30}$$

其中 $w_1^x(1,f(x))$ 代表数 x 中 1 位的个数，$w_1^y(1,f(y))$ 代表数 y 中 1 位的个数，$f(x)$ 代表数 x 的总位数，$f(y)$ 代表数 y 的总位数。或运算中的 2 位屏蔽概率参数与 3 位屏蔽概率计算参数的计算方法类似，只需统计操作数中连续两位均为 1 位的个数和连续三位均为 1 位的个数即可，计算式如(3-31)和(3-32)所示。

$$p_{mask}^{2}=\frac{\sum_{l=1}^{size(Z_2^1(x))}(Z_2^1(x)(l)-1)+\sum_{l=1}^{size(Z_2^1(y))}(Z_2^1(y)(l)-1)}{f(x)+f(y)-2} \tag{3-31}$$

$$p_{mask}^{3}=\frac{\sum_{l=1}^{size(Z_3^1(x))}(Z_3^1(x)(l)-2)+\sum_{l=1}^{size(Z_3^1(y))}(Z_3^1(y)(l)-2)}{f(x)+f(y)-4} \tag{3-32}$$

其中，$Z_2^1(x)$ 和 $Z_3^1(x)$ 分别代表 x 中连续 2 位和连续 3 位为 1 的位数集合，$Z_2^1(y)$ 和 $Z_3^1(y)$ 分别代表 y 中连续 2 位和连续 3 位为 1 的位数集合，l 代表集合中第 l 项元素，$size()$ 表示括号内集合的元素个数。通过 p_{mask}^1，p_{mask}^2，p_{mask}^3 可以计算屏蔽概率参数 P_{mask}，计算公式如(3-33)所示。

$$P_{mask}=(P_{SBU}+P_{V2BU}+P_{V3BU}+P_{C2BU})\cdot p_{mask}^1+(P_{H2BU}+P_{L3BU})\cdot p_{mask}^2+P_{H3BU}\cdot p_{mask}^3 \tag{3-33}$$

(3) 比较运算

在程序中比较指令之后通常是跳转指令，根据比较的结果跳转指令会选择一个分支。若单粒子翻转引起的数据错误未改变跳转指令的跳转分支，则此错误就会被屏蔽。继续使用 x,y 作

为示例对屏蔽概率参数 P_{mask} 进行分析。若比较指令中 $x>y$ 程序跳转至正确目标指令，即按正常执行时，x 要比 y 大。设单粒子翻转会随机发生在数据 x 和 y 上，且分布均匀。此时可分为下列几种情况进行讨论。

1) $(\lfloor \log_2(x) \rfloor > \lfloor \log_2(y) \rfloor)$ 且 $(y < x - 2^{\lfloor \log_2(x) \rfloor})$

$\lfloor \log_2(x) \rfloor$ 与 $\lfloor \log_2(y) \rfloor$ 不相等说明 x 与 y 的最高 1 位并不在同一位上，$y < x - 2^{\lfloor \log_2(x) \rfloor}$ 则是指 x 去掉最高位仍然要比 y 大。此时翻转 x 中的任意一位，x 仍比 y 大，错误会被屏蔽。若翻转 y 中末位至 x 最高 1 位对应位，错误也会被屏蔽，故此时的单位屏蔽概率参数 p^1_{mask} 的计算如式(3-34)所示。

$$p^1_{mask} = \frac{f(x) + \lfloor \log_2(x) \rfloor + 1}{f(x) + f(y)} \tag{3-34}$$

2) $(\lfloor \log_2(x) \rfloor > \lfloor \log_2(y) \rfloor)$ 且 $(y \geq x - 2^{\lfloor \log_2(x) \rfloor})$

与上一种情况相比，当 x 最高位发生翻转以及 y 中 x 最高位对应的位发生翻转时，错误不会被屏蔽。故此时参数 p^1_{mask} 的计算如式(3-35)所示

$$p^1_{mask} = \frac{f(x) + \lfloor \log_2(x) \rfloor - 1}{f(x) + f(y)} \tag{3-35}$$

3) $\lfloor \log_2(x) \rfloor = \lfloor \log_2(y) \rfloor$

$\lfloor \log_2(x) \rfloor$ 与 $\lfloor \log_2(y) \rfloor$ 相等说明 x 与 y 的最高 1 位相同，此时问题的关键点并不在此处，而在于 x 与 y 不同的最高位 $\lfloor \log_2(x \wedge y) \rfloor$。此时需要使用一辅助分类的二进制数 u 进行分析。u 为一种满足特定数值的二进制数，其前 $(f(u) - \lfloor \log_2(x \wedge y) \rfloor)$ 位均为 0，后 $\lfloor \log_2(x \wedge y) \rfloor$ 位均为 1。即 $f_0^u(1, \lfloor \log_2(x \wedge y) \rfloor) = 0$，$f_1^u(\lfloor \log_2(x \wedge y) \rfloor + 1, f(u)) = 0$。此时单位屏蔽概率参数 p^1_{mask} 的计算可使用 u 进行分类讨论。

3.1) $(x \& u) > (y \& u)$

x 中 0 位翻转成 1 位会使 x 增大，y 中的 1 位变为 0 位会使得 y 变小，二者均不会导致结果发生改变，错误会被屏蔽。x 从末位至 $\lfloor \log_2(x \wedge y) \rfloor$ 位中的 1 位翻转尽管会使 x 减小，但仍保证 x 比 y 大。y 从末位至 $\lfloor \log_2(x \wedge y) \rfloor$ 位中的 0 位翻转会使 y 增大，但仍比 x 小。故屏蔽概率参数 p^1_{mask} 如式(3-36)所示。

$$p^1_{mask} = \frac{w_0^x(1, f(x)) + w_1^y(1, f(y)) + w_1^x(1, \log_2(x \wedge y)) + w_0^y(1, \log_2(x \wedge y)) + 2}{f(x) + f(y)} \tag{3-36}$$

3.2) $(x \& u) \leq (y \& u)$

与上一种情况相比，此时翻转 x 和 y 的 $\log_2(x \wedge y)$ 位会导致结果改变，故屏蔽概率参数 p^1_{mask} 如式(3-37)所示。

$$p^1_{mask} = \frac{w_0^x(1, f(x)) + w_1^y(1, f(y)) + w_1^x(1, \log_2(x \wedge y)) + w_0^y(1, \log_2(x \wedge y))}{f(x) + f(y)} \tag{3-37}$$

上述结果只给出了单位屏蔽概率参数 p^1_{mask}，多位屏蔽概率参数由于情况复杂，难以使用计

算公式算出，故本文使用枚举法进行计算。对于二进制数 x 与 y，遍历翻转 x 中所有的连续两数据位，再比较 x 与 y。若 $x > y$，则此时未发生屏蔽。若 $x \le y$，证明此时发生屏蔽。记 $t^2_{mask}(x)$ 为枚举翻转 x 的过程中发生屏蔽的次数。同理可进行三位连续翻转，并记 $t^3_{mask}(x)$ 为枚举翻转 x 的过程中发生屏蔽的次数。则比较指令的 2 位屏蔽概率参数和 3 位屏蔽概率参数的计算如式(3-38)和(3-39)所示。

$$p^2_{mask} = \frac{t^2_{mask}(x) + t^2_{mask}(y)}{f(x) + f(y)} \tag{3-38}$$

$$p^3_{mask} = \frac{t^3_{mask}(x) + t^3_{mask}(y)}{f(x) + f(y)} \tag{3-39}$$

通过 p^1_{mask}， p^2_{mask}， p^3_{mask} 可以计算屏蔽概率参数 P_{mask}，计算公式如(3-40)所示。

$$P_{mask} = (P_{SBU} + P_{V2BU} + P_{V3BU} + P_{C2BU}) \cdot p^1_{mask} + (P_{H2BU} + P_{L3BU}) \cdot p^2_{mask} + P_{H3BU} \cdot p^3_{mask} \tag{3-40}$$

本文借助 LLVM 工具，建立程序有向依赖图 PDG（Program Dependence Graph）[79]对程序切片，进而实现对程序的理解与分析。根据 PDG，我们将程序中的指令表示为式(3-41)所示的四元组。其中，O_s 为指令 I 中的源操作数，O_d 为指令 I 中的目的操作数，S_B 为指令 I 的后向切片，S_F 为指令 I 的前向切片。

$$I = \langle O_s, O_d, S_B, S_F \rangle \tag{3-41}$$

使用程序切片技术分析数据变量在程序中的传输过程，可生成基于数据变量的与程序数据流相关的指令集合。该指令集合是由程序中所有包含该数据变量指令组成的有序集合，具体表示如式(3-42)所示。

$$S_{df}(x) = \{I_{in}, ..., I_i, ..., I_{end}\} \tag{3-42}$$

x 为程序中的数据变量，I_{in} 一般为数据输入指令，指程序中调用该数据的第一条指令。I_{end} 为数据末端指令，是程序最后包含该数据的指令。I_i 为程序中包含该数据的第 i 条指令，由于 $S_{df}(x)$ 为有序集合，故 i 也代表指令在程序中的位置。

通过屏蔽概率参数 P_{mask} 和数据流指令集合 $S_{df}(x)$ 可以计算出指令 I_i 关于数据变量 x 的屏蔽概率特征 $f_m(I_i, x)$，计算方法如式(3-43)所示。

$$f_m(I_i, x) = \begin{cases} 0 & ,if \quad I_i \notin S_{df}(x) \\ 1 - \prod_{j=i}^{end} (1 - P_{mask}(I_j)), if \quad I_i \in S_{df}(x) \end{cases} \tag{3-43}$$

将屏蔽概率参数 P_{mask} 和屏蔽概率特征 $f_m(I_i, x)$ 组合，可生成屏蔽特征，记为 F_{mask}，表达如式(3-44)所示。

$$F_{mask} = \langle P_{mask}, f_m(I_i, x) \rangle \tag{3-44}$$

综合上述所有传播性特征分析，本文提取了影响数据错误在程序中传播的指令传播性特征 F_{spread}，如式(3-45)所示。

$$F_{spread}=\langle F_{data},F_{OP},F_{CMP},F_{mask}\rangle \tag{3-45}$$

其中，$F_{data}=\langle life(num),width(num),is_\text{int},is_float\rangle$为数据变量相关特征，$F_{OP}=\langle is_and_op,is_or_op\rangle$为运算指令相关特征，$F_{CMP}=\langle is_cmp,is_icmp,is_fcmp\rangle$为比较指令相关特征，$F_{mask}=\langle P_{mask},f_m(I_i,x)\rangle$为屏蔽特征。

3.3.2 提取指令固有性特征

除去指令传播型特征外，指令本身的类型属性也是一种重要特征，本文将这些特征按照属性特点分成了 5 个类型，分别为：(1)执行时间相关特征。指令的执行时间也是一项需要考虑的重要特征。在程序中以较高的频率执行的指令最有可能频繁的和数据产生交互，这对指令的多位 SDC 脆弱性有很大的影响。(2)连接指令相关特征。在程序中一些指令起到了一种连接作用，可以是函数调用，或者参数传入，还可能是变量引用，它决定了程序数据之间的交互与返回，如果发生数据损坏也最容易引发 SDC 错误。上述两类指令特征一般被认为是较为重要的特征，也被称为关键指令[75]。(3)程序结构相关的特征。指令出错是否容易导致 SDC 错误，与其本身所处的程序代码结构高度相关。例如，顺序结构中的指令与循环结构中的指令出错引发 SDC 错误的概率不相同。(4) 算数运算相关的指令特征。程序中的算术运算相关的指令代表指令中包含有“加、减、乘、除”四则运算。算术运算和逻辑运算不同，算数运算是一种结果恒定的运算。在运算过程中某一操作数受到单粒子翻转影响后，运算结果必然会发生改变，不存在错误屏蔽的情况。(5)与存储器和寄存器读写相关的指令特征。在程序运行过程中访问的内存地址必须位于合法的地址空间内，如果访问的内容超出了内存地址的合法范围，那么此操作将导致系统段错误。若访问地址发生了错误，但是未超出内存地址的合法范围就可能引发 SDC 错误。在实验过程中，由于统计程序中各指令执行的具体时间较困难，所以使用指令执行次数来代替指令执行时间。一般而言，在整个程序运行过程中，经常执行含有读写数据操作的指令，更容易导致 SDC 错误。指令固有性特征具体见表 3.2。

表 3.2 指令固有性特征

类别	特征名称	类型	描述
执行时间相关特征	execution_time	int	指令动态执行次数
连接指令相关特征	is_global	bool	指令中是否包含全局变量
	is_func_call	bool	指令中是否包含函数调用
	is_stack_allocation	bool	指令中是否包含堆栈分配
程序结构相关的特征	bb_size	int	指令所在基本块包含的指令数目

	num_of_suc_bb	int	指令所在基本块的后继基本块数量
	loop_depth	int	嵌套此指令的最大循环深度
算术运算指令相关的特征	is_add_op	bool	指令中是否包含加法运算
	is_ subtraction _op	bool	指令中是否包含减法运算
	is_multiplication _op	bool	指令中是否包含乘法运算
	is_ division _op	bool	指令中是否包含除法运算
与存储器和寄存器读写相关的指令特征	is_read_with_mem	bool	指令中是否包含从存储器读取数据的操作
	is_write_with_mem	bool	指令是否包含从存储器写入数据的操作
	is_read_with_reg	bool	指令是否包含从寄存器读取数据的操作
	is_write_with_reg	bool	指令是否包含从寄存器写入数据的操作

综合上述所有固有性特征分析，本文提取了影响数据错误在程序中传播的指令传播性特征 $F_{inherence}$，如式(3-46)所示。

$$F_{inherence}=\left\langle t_{execution},F_{connect},F_{structure},F_{arithmetic},F_{MemReg}\right\rangle \tag{3-46}$$

其中，$t_{execution}$为执行时间相关特征，代表程序运行一次时指令的动态执行次数。连接指令相关特征$F_{connect}=\langle is_global,is_func,is_stack\rangle$，内部元素分别代表指令中是否含有全局变量、是否包含函数调用、是否包含堆栈分配。程序结构相关的特征$F_{structure}=\left\langle bb_{size},bb_{succ},d_{loop}\right\rangle$，内部元素分别代表指令所在基本块的大小，所在基本块后继基本块的数量，嵌套指令的最大循环深度。算术运算指令相关的特征$F_{arithmetic}=\langle op_{add},op_{sub},op_{mul},op_{div}\rangle$，内部元素分别代表指令中是否包含加法运算、是否含有减法运算、是否含有乘法运算、是否含有除法运算。与存储器和寄存器读写相关的指令特征$F_{MemReg}=\left\langle r_{Mem},w_{Mem},r_{Reg},w_{Reg}\right\rangle$，内部元素分别代表指令中是否包含从存储器读取数据的操作、是否包含从存储器写入数据的操作、是否包含从寄存器读取数据的操作、是否包含从寄存器写入数据的操作。

在特征提取过程中，提取出的特征类型包含有 int 类型，对于所有 int 类型特征，需要归一化处理，公式如(3-47)所示。

$$f_i=\frac{\left|feature_i-feature_{i\min}\right|}{\left|feature_{i\min}-feature_{i\max}\right|} \tag{3-47}$$

其中， $feature_i$ 代表未经过处理的第 i 项指令特征， $feature_{i\max}$ 和 $feature_{i\min}$ 分别代表所有数据样本中该项特征的最小值与最大值， f_i 代表经过归一化处理后的第 i 项指令特征。

3.4 基于 SDC 多位脆弱性分析的数据流错误检测及加固方法

3.4.1 数据流错误检测算法

本文设计的数据流错误检测算法共包含两个部分：指令特征提取算法(IFE, Instruction feature extraction)和 I_{ecSDC} 指令识别算法(I- I_{ecSDC} , Identification of instructions that easily cause SDC errors)。

IFE 算法主要是对程序的指令特征进行提取。算法将程序中的所有指令分成两类，指令传播性特征和指令固有性特征，并根据相应计算公式及函数计算并提取特征，IFE 算法的具体步骤如下：

Step1. 将待提取指令特征的程序编译成中间代码，分提出中间代码中的指令。

Step2. 判断指令中是否含有数据变量及指令类型，提取出指令的数据传播性特征，如式(3-21)所示。

Step3. 判断指令类型，对含有逻辑运算操作的指令和比较运算操作的指令提取相关特征，如式(3-22)和(3-23)所示。对于含有逻辑运算操作的指令，使用式(3-24)至(3-33)计算其屏蔽概率参数。对含有比较运算操作的指令，则使用式(3-34)至(3-40)计算器屏蔽概率参数。

Step4. 使用程序切片技术，生成基于数据变量的与程序数据流相关的指令集合，如式(3-41)和(3-42)所示。根据指令集合和屏蔽概率参数，使用式(3-43)计算出指令的屏蔽概率特征。将上述结果组合，生成屏蔽特征如式(3-44)所示。

Step5. 将 step2 至 step4 中的所有特征进行组合，最终生成指令传播性特征，如式(3-45)所示。

Step6. 根据表 3.2 指令固有性特征表 3.2 遍历程序中所有指令，结合 LLVM 的静态分析和动态分析技术提取指令固有性特征，如式(3-46)所示。

Step7. 将得到的指令传播性特征和指令固有性特征组合，并根据式(3-47)进行归一化处理，最终生成指令的特征向量。

表 3.3 符号定义

符号	描述
Pg	待提取指令的程序
$ISet(Pg)$	程序所有指令的集合
I_i	程序中第 i 条指令
$I_i.data\_variable$	指令 I_i 的数据变量
$I_i.F_{data}$	指令 I_i 的数据传播特征
$I_i.opcode$	指令 I_i 的操作码类型

$I_i.F_{op}$	指令 I_i 的逻辑运算相关特征
$I_i.P_{mask}$	指令 I_i 的屏蔽概率参数
$I_i.F_{cmp}$	指令 I_i 的比较运算相关特征
$S_{df}(I_i.data\_variable)$	基于数据变量 $I_i.data\_variable$ 的与程序数据流相关的指令集合
$I_i.f_m$	指令 I_i 的屏蔽概率特征
$I_i.F_{mask}$	指令 I_i 的屏蔽特征
$I_i.F_{spread}$	指令 I_i 的传播性特征
$I_i.F_{inherence}$	指令 I_i 的固有性特征

为了进一步说明指令特征提取过程，文本给出了算法的伪代码形式，表 3.3 给出了算法中所用到的符号定义。代码从对程序的预处理开始，直至完成对程序中所有指令的特征提取，如算法 3.1 所示。

算法 3.1 指令特征提取算法

IFE算法 指令特征提取

输入：待提取指令特征的程序*Pg*
输出：每条指令的特征向量
1：**Clang** *Pg* to get *ISet(Pg)*
2：**for all** I_i in *ISet(Pg)* **do**
3： **if** I_i.*data_variable!=∅* ***then***
4： analyze $I_i.F_{data}$ using formula (3-20) and (3-21)
5： **end if**
6： **if** I_i.opcode==*and* ***then***
7： analyze $I_i.F_{op}$ using formula (3-22)
8： calculate $I_i.P_{mask}$ using formula from(3-24) to (3-29)
9： **else if** I_i.opcode==*or* ***then***
10： analyze $I_i.F_{op}$ using formula (3-22)
11： calculate $I_i.P_{mask}$ using formula from(3-30) to (3-33)
12： **else if** I_i.opcode==*cmp* ***then***
13： analyze $I_i.F_{cmp}$ using formula (3-23)
14： calculate $I_i.P_{mask}$ using formula from(3-34) to (3-40)
15： **end if**
16：**end for**
17：**for** all I_i in *ISet(Pg)* do
18： Identify the slice of I_i and generate $S_{df}(I_i.data\_variable)$ using formula (3-41) to (3-42)
19： calculate $I_i.f_m$ using formula (3-43)
20： generate $I_i.F_{mask}$ using formula (3-44)
21： generate $I_i.F_{spread}$ using formula (3-45)
22：**end for**
23：**for all** I_i in *ISet(Pg)* **do**
24： analyze and calculate $I_i.F_{inherence}$ in table 3.2 using formula (3-46)
25： Normalized and generate features of Instruction using formula (3-47)
26：**end for**

算法首先将程序编译成中间代码文件(Line 1)，提取指令数据传播特征(Line 2-5)。根据指令的操作类型，提取指令运算相关的特征并计算屏蔽概率参数(Line 6-16)。使用程序切片分析技术生成基于数据变量的与程序数据流相关的指令集合，并结合屏蔽概率参数计算屏蔽概率特征，最终生成指令的屏蔽特征(Line 17-20)。综合上述计算结果，生成指令的传播性特征(Line 21)。从头遍历所有指令，生成指令的固有性特征(Line 22-24)，最终经过归一化处理提取出所有指令

特征(Line 25-26)。

I-I_{ecSDC}算法是数据流错误检测算法中的指令识别算法，针对目标程序，训练指令识别模型，提取目标程序内部指令特征，识别目标程序中的脆弱性指令，最终识别出目标程序中所有I_{ecSDC}指令。I-I_{ecSDC}算法的具体步骤如下：

Step1. 将目标程序和训练程序编译成中间代码，分别提出中间代码中的指令。

Step2. 使用 IFE 算法分析并提取训练程序以及目标程序指令中的指令特征值，包括传播性特征和固有性特征。

Step3. 对基本训练的指令进行故障注入实验以判别训练样本类型，通过式(3-4)计算指令的多位 SDC 脆弱性。根据脆弱性进一步获得指令标签，和指令特征结合形成训练样本。

Step4. 建立并使用训练集训练 SDCVA-OCSVM 的指令识别器，如式(3-5)至(3-19)所示。

Step5. 通过指令识别器识别目标程序中的所有I_{ecSDC}指令。

表 3. 4 符号定义

符号	描述
Pg_{target}	待检测的目标程序
Pg_{train}	用于训练的基本训练程序
S	指令 SDC 诱发率的阈值
$ISet(Pg_{target})$	目标程序所有指令的集合
$ISet(Pg_{train})$	训练程序所有指令的集合
$size()$	所含指令的数目
I_i	程序中第i条指令
$I_i.P_{ecSDC}$	指令 SDC 诱发率
$I_i.Label$	指令的类型标签
$TSet$	用作训练样本的指令集
$ISet_{ecSDC}(Pg_{target})$	目标程序所有I_{ecSDC}指令的集合
$unISet_{ecSDC}(Pg_{target})$	目标程序所有非I_{ecSDC}指令的集合

为了进一步说明 SDCVA-OCSVM 识别I_{ecSDC}指令的过程，文本给出了 I-I_{ecSDC}算法的伪代码形式，表 3. 4 给出了算法中所用到的符号定义。代码从对目标程序的预处理开始，直至完成对目标程序中所有I_{ecSDC}的指令，如算法 3. 2 所示。

提取目标程序与基本训练程序的指令集合，并获取所有指令特征值(Line 1-4)。其次，对基本训练集中的指令进行完全故障注入实验以获得他们的指令标签，标签与指令特征组合形成训练样本(Line 5-13)。然后，使用训练集训练基于 OCSVM 的指令检测器(Line 14)。最后，使用指令检测器对剩余的指令进行分类(Line 15-22)。

算法 3.2 指令识别算法

I-I_{ecSDC}算法 指令识别算法

输入：待识别I_{ecSDC}指令的目标程序Pg_{target}和训练程序集Pg_{train}
输出：目标程序中I_{ecSDC}指令集合

```
Clang Pg_target and Pg_train to get ISet(Pg_target) and ISet(Pg_train)
for all I_i in ISet(Pg_target)∪ISet(Pg_target) do
    analyze and calculate features of Instruction using algorithm IFE.
end for
for all I_i in ISet(Pg_train) do
    calculate P_ecSDC(I_i) using formula (3-4) and fault injection tools.
    if P_ecSDC(I_i)>S then
        I_i.Label = +1
    else
        I_i.Label = -1
    end if
    generate sample of I_i , then add it to TSet.
end for
Generate and train SDCVA-OCSVM instruction identifier using TSet and
formula from (3-5) to (3-19)
for all I_i in ISet(Pg_target) do
    get I_i.Label using idetifier
    if I_i.Label = +1 then
        add I_i to ISet_ecSDC(Pg_target)
    else
        add I_i to unISet_ecSDC(Pg_target)
    end if
end for
```

3.4.2 指令加固策略设计

如今，大部分技术在找到 I_{ecSDC} 指令后，往往会忽视冗余加固的部分。但是，建立科学合理的复制与检查规则，再对脆弱部位加固，可以有效的提高方法的效率并节约成本。文献[30][77]使用了一种数据流冗余备份方法，称为 VAR (Variables technology)方法，旨在降低内存，性能及时间成本，实现低能耗加固。该技术在对 miniMIPS 处理器的故障注入实验中，表现出了良好的性能[76]。

但是 VAR 技术本身是一种全冗余加固，具有较高的成本，并且更针对硬件，而且本身对 SDC 错误不具备针对性，所以本文在原有的 VAR 技术基础上对其规则进行了改进，提出了名为 IRVAR(Intermediate Representation VAR)的冗余加固方法。在 LLVM 中，clang 编译的中间文件很特殊，它使用一种静态单赋值(Static Single Assignment, SSA)的中间语言。中间文件是介于源代码与汇编语言文件之间的形式，内部由中间指令也称为 IR 指令构成，本文中所提及的所有指令均为 IR 指令。在 IRVAR 方法中，我们规则制定的主要对象也是 IR 指令。IRVAR 方法共由三种不同的规则组成：全局规则，复制规则以及检查规则，如表 3.5 所示。

表 3. 5 IRVAR 方法中的复制与检查规则

名称	描述
	全局规则
G1	对程序中所有指令中使用的变量均设置一个副本变量
	复制规则
D1	复制所有的指令
D2	复制所有的 I_{ecSDC} 指令
D3	复制除了含有 load/store 操作的所有 I_{ecSDC} 指令
	检查规则
C1	在所有右值中有读取变量操作指令之前设置检查点 (含有 load/store 操作以及分支跳转的指令除外)
C2	在所有赋值变量操作指令之前设置检查点 (含有 load/store 操作以及分支跳转的指令除外)
C3	在含有 load 操作指令前设置检查点
C4	在含有 load 操作指令后设置检查点
C5	在含有 store 操作指令前设置检查点
C6	在含有 store 操作指令后设置检查点
C7	在分支指令及跳转指令之前设置检查点

全局规则中规定所有 I_{ecSDC} 指令中所包含的变量必须有一个冗余变量作为副本。一般的，所有针对原始变量的操作，都可以对冗余变量执行。复制规则主要考虑指令的复制方式，通过 Clang 编译出来的 LLVM 中间代码中，绝大部分指令结果都会存到一个形如%n 的局部变量中，所以复制规则也需要创建一个新的局部变量。

IRVAR 中有三种类型的复制规则 D1、D2 和 D3，通常只选择其中一到两个。LLVM 并没有明确的内存变量，也很少涉及内存地址，它提供了几乎无限的虚拟寄存器来存储一些基本类型数据，对于用户来说很难区分这些数据究竟来自于寄存器还是存储器。但是 LLVM 会通过 store/load 操作来完成寄存器与内存之间的数据交换。D1 会复制包括 store/load 操作在内的所有指令。当程序使用的存储器未采取任何保护技术时，就需要应用这一条规则。因为原始数据和副本数据可以存储在存储器中的不同位置，以达到加固目的。D2 会复制包括 store/load 操作的所有I_{ecSDC}指令。D3 会复制除了 store 操作之外的所有I_{ecSDC}指令。当存储器采取了保护技术，即在用户相信存储器绝对安全的情况下，只采取 D3 规则就足够了，这样可以减少复制代码以及访问存储器备份的开销，但为了验证方法完备性，本文暂先将存储器故障也考虑进来。

检查规则主要是规定在何时何地比较原始程序值与冗余副本值。程序运行时，设置的冗余

副本和原始程序均会正常执行，到达检查点时会对原始程序值与冗余副本值进行比较，若二者相等证明程序正常运行，若二者不等证明程序出错。检查规则可以选择多个，理论上检查规则使用的越多，加固的可靠性就越高，但开销也越大。

通过对上述规则的排列组合就可以生成冗余加固技术。本文从规则库中选择了相应规则，生成了四种冗余加固策略，每种策略应用的具体规则见表 3. 6 中。

表 3. 6 冗余加固技术规则

名称	复制规则	检查规则
IRVAR-L	D1	C4,C6,C7
IRVAR-M	D2	C4,C7
IRVAR-S	D2	C4
IRVAR-C	D2	C7

从表 3. 6 可以看出 ，这四种技术所选规则数量是从上至下依次降低的，由于所选规则不同，加固代码也有所不同，其中 IRVAR-L 是冗余代码最多的技术，开销也是四种技术中最高的。IRVAR-M 技术较为均衡，综合考虑了开销与性能的关系，这种技术与 EDDI 技术[78]很相似。IRVAR-S 技术则是开销较低的技术，只在数据存入存储器之前进行检查，以提高性能。创建该技术的主要目的是为了探究技术开销的下限，观察该技术在较低开销的情况下是否仍能提供较高的故障检测率。IRVAR-C 技术则与前三种技术不同，它完全脱离了数据在硬件间的传输过程，单纯以程序流程的观点出发，只在跳转指令前设置检查点，这也是目前应用最广的技术。

3. 4. 3 数据流错误检测与加固方案

在本文中，数据流错误检测及加固方法 SDCVA-OCSVM 旨在保护存储器或寄存器中的数据。方法的框架如 SDCVA-OCSVM 的检测流程如图 3. 4 所示。首先选取目标程序代码和训练代码，通过编译器提取代码中的指令。提取并计算上述代码的两类指令传播性特征和五类指令固有性特征。通过故障注入实验来确定训练代码中的指令是否为 I_{ecSDC} 指令，从而生成训练数据集。使用训练数据集训练检测器，训练完成后可用来识别 I_{ecSDC} 指令。将目标程序代码提取出的指令结合特征转化为数据向量，使用检测器对其进行判别，找出目标程序中所有的 I_{ecSDC} 指令。最后，建立复制与检查规则库，从中选取合适的规则形成了四种冗余加固策略：IRVAR-L，IRVAR-M，IRVAR-S，IRVAR-C。结合检测出的 I_{ecSDC} 指令以及冗余加固策略，对目标程序进行加固，得到受到保护的程序。

所示，由数据层，加固层和评估层组成。

评估层负责对 SDCVA-OCSVM 方法进行评估，包括结果显示模块和性能评估模块。性能评估模块主要对方法的 SDC 脆弱性指令识别性能和加固策略进行评估。结果显示模块会将性能评估模块的评估结果显示出来。

SDCVA-OCSVM 方法分为两个阶段，指令特征提取阶段和指令识别加固阶段。在指令特征提取阶段，通过静态分析与故障注入的方式，提取指令的相关特征。指令特征包括指令传播性特征和指令固有性特征。指令传播性特征又包括数据传播特征和屏蔽特征两类。指令固有性特征则包括存储器与寄存器相关特征、程序结构相关特征、比较指令相关特征、连接指令相关特征、指令运算相关的特征等五类。提取并计算上述所有特征并建立训练集和目标程序指令集。在指令识别加固阶段，利用训练数据集，本文建立了指令检测器来识别程序中的 I_{ecSDC} 指令。建立一系列的复制与检查规则，再根据订立的规则加固上一阶段识别出的 I_{ecSDC} 指令。

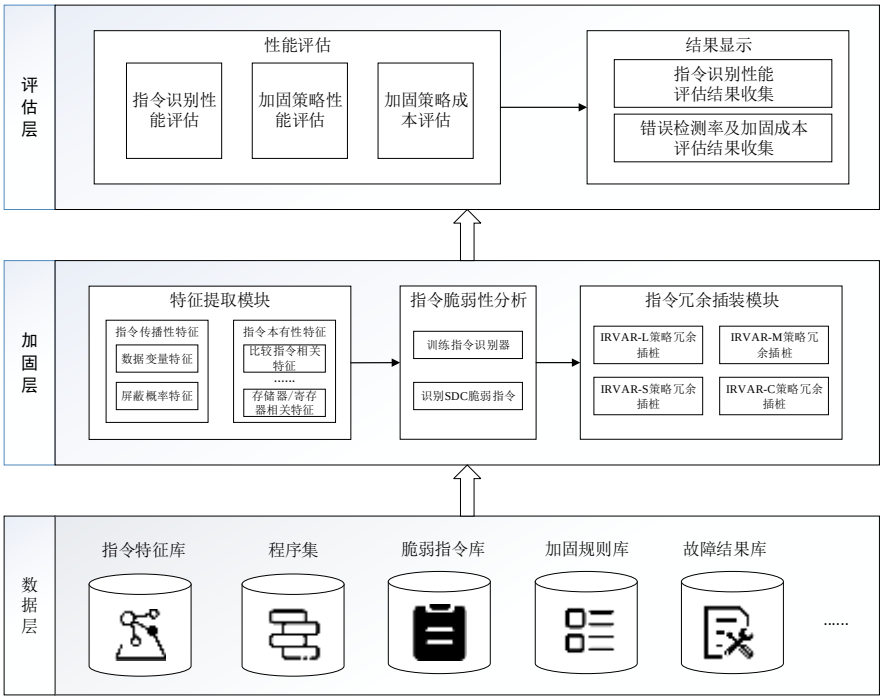


图 3. 3 SDCVA-OCSVM 检测框架

SDCVA-OCSVM 的检测流程如图 3. 4 所示。首先选取目标程序代码和训练代码，通过编译器提取代码中的指令。提取并计算上述代码的两类指令传播性特征和五类指令固有性特征。通过故障注入实验来确定训练代码中的指令是否为 I_{ecSDC} 指令，从而生成训练数据集。使用训练数据集训练检测器，训练完成后可用来识别 I_{ecSDC} 指令。将目标程序代码提取出的指令结合特征转化为数据向量，使用检测器对其进行判别，找出目标程序中所有的 I_{ecSDC} 指令。最后，建立复制与检查规则库，从中选取合适的规则形成了四种冗余加固策略：IRVAR-L，IRVAR-M，IRVAR-S，IRVAR-C。结合检测出的 I_{ecSDC} 指令以及冗余加固策略，对目标程序进行加固，得到受到保护的程序。

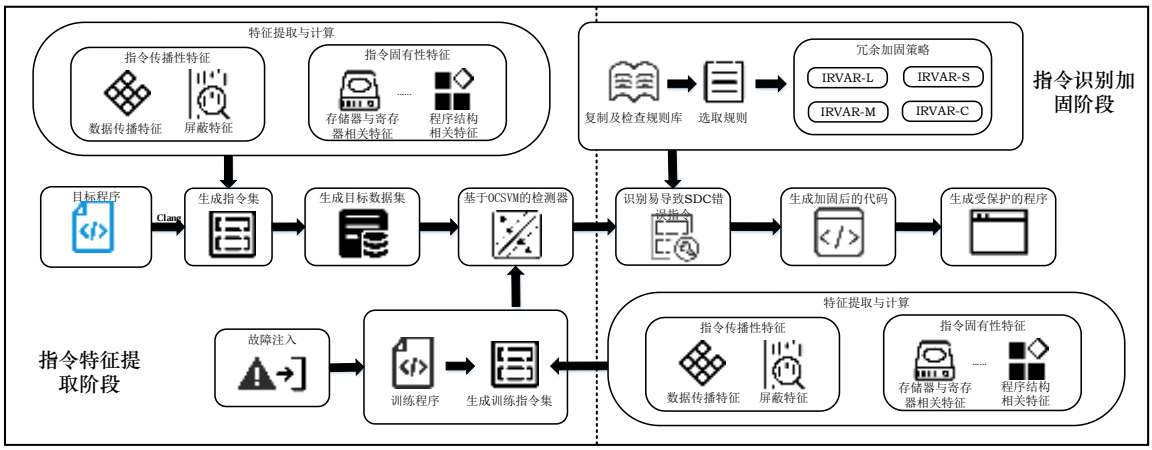


图 3. 4 SDCVA-OCSVM 检测流程

3. 4. 4 实例

为了进一步的理解 SDCVA-OCSVM 方法，本文选用快速排序算法的一部分作为应用实例，具体可见图 3. 5，图 3. 6，图 3. 7 和表 3. 7，表 3. 8。

(1) SDC 错误受传播影响示例

首先，我们选择快速排序算法作为示例来说明受到单粒子翻转影响的指令如何在程序中传播错误。快速排序算法中的一段代码及相应的中间代码的控制流程如图 3. 5 和图 3. 6 所示。在图 3. 6 中，%4 和 9%寄存器均向内存申请了 32 位空间，当发生单粒子翻转时，寄存器中存储的错误值会传播至整个程序。若%4 寄存器发生了翻转错误，根据控制流程图的后继关系，将会导致存储指令存储了错误的值到%10 寄存器中，发生数据存储错误。%11 寄存器读取了%10 寄存器中的错误值，从而导致分支跳转指令发生错误的跳转。若 9%寄存器发生故障，%42 寄存器读取了%9 寄存器中的错误值继而会影响%43 寄存器。%43 寄存器就会将错误值导入 qsort 函数中，使得函数调用指令传入了错误的参数。

```
int main(int argc, char *argv[]) {
  struct myStringStruct array[MAXARRAY];
  FILE *fp;
  int i,count=0;
  if (argc<2) {
    fprintf(stderr,"Usage: qsort_small <file>\n");
    exit(-1);
  }
  else {
    fp = fopen(argv[1],"r");
    while((fscanf(fp, "%s", &array[count].qstring) == 1) && (count < MAXARRAY)) {
            count++;
    }
  }
  printf("\nSorting %d elements.\n\n",count);
  qsort(array,count,sizeof(struct myStringStruct),compare);
  for(i=0;i<count;i++)
    printf("%s\n", array[i].qstring);
  return 0;
}
```

图 3. 5 快速排序算法部分代码

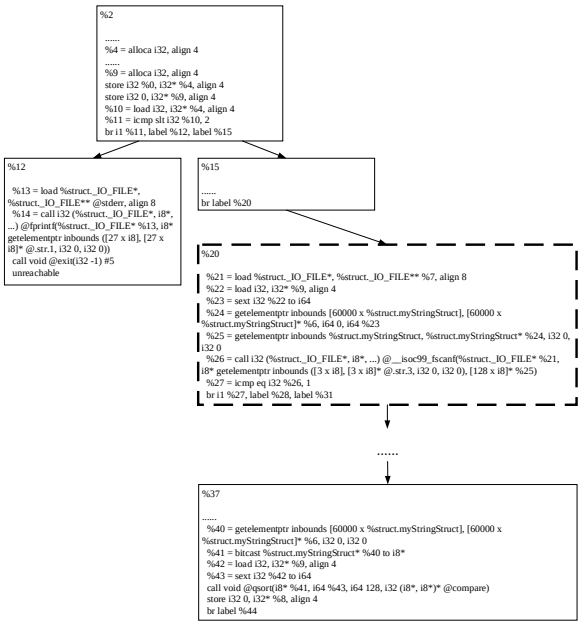


图 3. 6 代码对应的部分控制流程

(2) 指令多位 SDC 脆弱性分析实例

为了得到程序中指令的 SDC 脆弱性值并分析其是如何随指令特征不同而出现差异的，需要进行故障注入实验。我们从图 3. 5 中截取虚线框内的一行源代码作为示例表 3. 7 中列出了这行代码对应的 LLVM IR 指令及通过式(3-3)计算得到的指令所对应的 SDC 脆弱性值。从表 3. 7 中可以看出，该代码对应于图 3. 5 中虚线框内的基本块。指令的 SDC 脆弱性和数据传播性是高度相关的。比如，56 号 icmp 指令决定了下一步分支指令的跳转方向，而该指令的数据位只有 1 位，如果发生了单粒子翻转，则 icmp 比较结果必然出错，进而导致程序错误跳转，造成 SDC 错误。

表 3. 7 示例代码的指令 SDC 脆弱性

ID	指令	SDC 脆弱性
50	%21 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8	0.044894293
51	%22 = load i32, i32* %9, align 4	0.415727651
52	%23 = sext i32 %22 to i64	0.207640258
53	%24 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %23	0.2623884
54	%25 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %24, i32 0, i32 0	0.259781346

55	%26 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %25)	0.833333
56	%27 = icmp eq i32 %26, 1	1
—	br i1 %27, label %28, label %31	—

通过对上述示例的分析与故障注入实验结果的分析可知，单粒子翻转是否会引起最终的SDC 错误是高度依赖于目标程序的。因此，研究高效费比的错误检测机制需要我们识别出程序中 SDC 脆弱性较高的指令，再对这些指令冗余处理。如上述示例代码中，脆弱性较高的 55 号 call 指令、56 号 icmp 指令就需要采取加固措施；脆弱性中等的 51 号 load 指令用户就可以根据性能需求来选择是否加固；而脆弱性较低的 50 号 load 指令、52 号 sext 指令、53 号和 54 号 getelementptr 指令就不需要加固。

表 3. 8 部分故障注入实验结果

ID	OPCODE	WIDTH	SDC(P_1)	SYM/CRASH (P_1)	TRUE(P_1)	……	SDC(P_7)	SYM/CRASH (P_7)	TRUE(P_7)
50	load	64	4	60	0	……	2	61	0
50	load	64	3	61	0	……	1	62	0
50	load	64	5	59	0	……	3	60	0
50	load	64	3	61	0	……	1	62	0
50	load	64	3	61	0	……	0	63	0
50	load	64	0	59	5	……	0	61	2
.....	.....	.....	.....	.....	.....	……	.....	.....	.....
56	icmp	1	1	0	0	……	1	0	0
56	icmp	1	1	0	0	……	1	0	0
56	icmp	1	1	0	0	……	1	0	0
56	icmp	1	1	0	0	……	1	0	0
56	icmp	1	1	0	0	……	1	0	0
56	icmp	1	1	0	0	……	1	0	0

我们进行故障注入实验后得到的结果如表 3. 8 所示，ID，OPCODE，WIDTH，SDC(P_i)，SYM/CRASH(P_i)，TRUE(P_i)分别表示指令分配序号，指令操作类型，指令数据位数，对应数据流错误模型故障注入发生 SDC 错误次数，对应数据流错误模型故障注入发生系统错误/崩溃次数，对应数据流错误模型故障注入程序正确执行次数。示例代码中共有 8 条静态指令，除去最后的跳转指令，分配序号后的 7 条指令组合的集合为 $I_{sset}=\{I_{50},I_{51},I_{52},I_{53},I_{54},I_{55},I_{56}\}$。$I_{56}$的指令脆弱性虽然最高，但是从表 3. 8 数据可以看出，I_{56} 只有 1 位数据位，作为示例不具有代表性，所以本文以指令 I_{50} 为例分析脆弱性计算流程。

从表 3. 8 可以看出，I_{50} 共执行了 6 次，故 $n_{50}=6$；形成动态指令集合 $\{I_{50}^1,I_{50}^2,I_{50}^3,I_{50}^4,I_{50}^5,I_{50}^6\}$；

在计算过程中，我们对单粒子多位翻转概率参考了文献[64][65]的实验结果，即 $P_{SBU}=83.3\%$，$P_{H2BU}=5.74146\%$，$P_{V2BU}=6.42115\%$，$P_{C2BU}=1.63493\%$，$P_{H3BU}=0.01503\%$，$P_{V3BU}=0.01503\%$，$P_{L3BU}=2.48329\%$。再由动态指令的脆弱性计算公式可知：$P_{ecSDC}(I_{50}^{1})=0.060352$，$P_{ecSDC}(I_{50}^{2})=0.044709393$，$P_{ecSDC}(I_{50}^{3})=0.075994046$，$P_{ecSDC}(I_{50}^{4})=0.044709636$，$P_{ecSDC}(I_{50}^{5})=0.043600964$，$P_{ecSDC}(I_{50}^{6})=0$。最后由式(3-3)可计算出指令的脆弱性。

$$P_{ecSDC}(I_i)=\frac{1}{n_i}\sum_{k=1}^{n_i}P_{ecSDC}(I_i^k)=0.044894293$$

IRVAR-L

```
%21 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8
%22 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8     //根据规则D1进行冗余
%23 = icmp ne i32 %21,%22              //根据规则C4设置检查点
br i1 %23, error, label %24
%25 = load i32, i32* %9, align 4
%26 = load i32, i32* %9, align 4     //根据规则D1进行冗余
%27 = icmp ne i32 %25,%26              //根据规则C4设置检查点
br i1 %27, error, label %28
%28 = sext i32 %25 to i64
%29 = sext i32 %26 to i64          //根据规则D1进行冗余
%30 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %25
%31 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %26     //根据规则D1进行冗余
%32 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %30, i32 0, i32 0
%33 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %31, i32 0, i32 0       //根据规则D1进行冗余
%34 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %32)
%35 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %33)
%36 = icmp eq i32 %34, 1
%37 = icmp eq i32 %35, 1    //根据规则D1进行冗余
%38 = icmp ne i32 %36,%37          //根据规则C7设置检查点
br i1 %38, error, label %39
%39 = load i32, i32* %36, align4
br i1 %36 label %39, label %42
```

IRVAR-M

```
%21 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8
%22 = load i32, i32* %9, align 4
%23 = load i32, i32* %9, align 4     //根据规则D2进行冗余
%24 = icmp ne i32 %22,%23          //根据规则C4设置检查点
br i1 %24, error, label %25
%25 = sext i32 %22 to i64
%26 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %22
%27 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %26, i32 0, i32 0
%28 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %27)
%29 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %27)
%30 = icmp eq i32 %28, 1
%31 = icmp eq i32 %29, 1    //根据规则D2进行冗余
%32 = icmp ne i32 %30,%31     //根据规则C7设置检查点
br i1 %32, error, label %33
%33 = load i32, i32* %30, align4
br i1 %30, label %34, label %37
```

IRVAR-S

```
%21 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8
%22 = load i32, i32* %9, align 4
%23 = load i32, i32* %9, align 4     //根据规则D2进行冗余
%24 = icmp ne i32 %22,%23     //根据规则C4设置检查点
br i1 %24, error, label %25
%25 = sext i32 %22 to i64
%26 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %22
%27 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %26, i32 0, i32 0
%28 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %27)
%29 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %27)
%30 = icmp eq i32 %28, 1
%31 = icmp eq i32 %29, 1    //根据规则D2进行冗余
br i1 %30, label %32, label %35
```

IRVAR-C

```
%21 = load %struct._IO_FILE*, %struct._IO_FILE** %7, align 8
%22 = load i32, i32* %9, align 4
%23 = load i32, i32* %9, align 4     //根据规则D2进行冗余
%24 = sext i32 %22 to i64
%25 = getelementptr inbounds [60000 x %struct.myStringStruct], [60000 x %struct.myStringStruct]* %6, i64 0, i64 %22
%26 = getelementptr inbounds %struct.myStringStruct, %struct.myStringStruct* %25, i32 0, i32 0
%27 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %26)
%28 = call i32 (%struct._IO_FILE*, i8*, ...) @__isoc99_fscanf(%struct._IO_FILE* %21, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str.2, i32 0, i32 0), [128 x i8]* %26)
%29 = icmp eq i32 %27, 1
%30 = icmp eq i32 %28, 1    //根据规则D2进行冗余
%31 = icmp ne i32 %29,%30     //根据规则C7设置检查点
br i1 %31, error, label %32
%32 = load i32, i32* %29, align4
br i1 %30, label %33, label %36
```

图 3. 7 各冗余情况对比示例

(3) 指令加固实例

图 3.7 给出了对上述示例进行冗余加固处理的对比示例，展示了这四种 VAR 技术的加固代码示例。假设原始代码中的指令均为 I_{ecSDC} 指令，为普通字体，复制规则为灰色字体，检查规则为斜体灰色字体。对于本文提出的不同规则的冗余方法中，IRVAR-L 需要对程序中所有的指令进行复制并在 load/store 指令和跳转指令前插入比较指令进行检测。IRVAR-M，IRVAR-S 和 IRVAR-C 则是对示例程序中指令 SDC 脆弱性较高的 $I_{selected}=\{I_{51},I_{55},I_{56}\}$ 集合中的指令进行冗余操作。

3.5 实验与结果分析

3.5.1 实验设计

为了验证 SDCVA-OCSVM 方法的性能，本文设计并实施了两项性能评估实验。SDCVA-OCSVM 方法首先建立并训练指令识别模型，用以识别 I_{ecSDC} 指令。为了评估模型识别 I_{ecSDC} 指令的能力，需要评估模型分类的准确率，召回率和 f-score。在识别出 SDC 脆弱性高的指令后，就可以使用冗余加固策略对程序进行加固。为了评估四种冗余加固策略的加固效果，需要使用能够模拟单粒子翻转现象的故障注入工具对加固后的程序进行故障注入实验。实验主要目的是观察程序被注入故障后，能否自动检测出发生的 SDC 错误，并统计 SDC 错误检测率。

实验环境如下所示：CPU 为 Intel(R) Core(TM) i7-6700HQ，内存 16G，硬盘空间 500G，操作系统为 Ubuntu18.04。实验首先使用基于 LLVM 4.0 的 Clang 编译实验用测试程序，得到程序中间代码，并以此搜集程序指令；然后提取计算指令特征，生成测试数据集。本文共选择了 5 个测试程序，FFT.c，dijkstra.c，susan.c，comprecal.c 以及 communication.c，算法描述如表 3.9 所示。

表 3.9 实验测试程序

名称	描述
FFT	快速傅里叶变换及其逆变换
dijkstra	迪杰斯特拉算法实现
Bubble_sort	冒泡排序算法实现
comprecal	计算机负载资源计算程序
datamanage	数据管理软件

五个测试程序中，前三个出自 mibench 的通信分类与网络分类。后两个出自一套分布式计算机集群管理系统，comprecal.c 主要用于计算计算机 CPU 利用率，内存使用率，网络带宽等负载资源。datamanage.c 用于计算机的一个数据管理软件。使用本文设计的数据流故障注入工具随机对程序指令注入故障来模拟单粒子翻转对程序运行的影响。本文主要从三个方面来评估 SDCVA-OCSVM 方法的有效性：(1)检测器判别易引发 SDC 错误指令的准确性。(2)程序 SDC

错误的检测能力。（3）加固成本。

3.5.2 数据流故障注入平台构建

在 3.1.1 节中我们分析了单粒子翻转的损伤机理和单粒子多位翻转的发生情况。在空间环境中单粒子多位翻转发生概率在 10%以上[63]，已经不属于偶然事件，在设计故障注入工具和错误检测方法时，必须考虑多位翻转的情况。

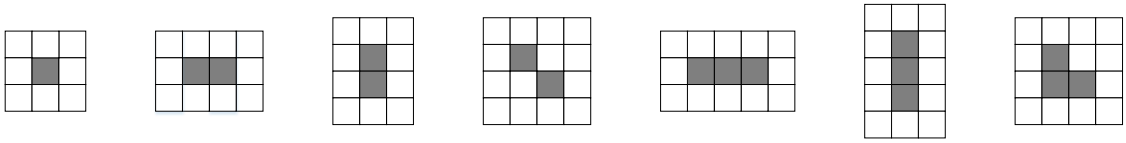


图 3.8 单粒子翻转可能发生形式

参考文献[64][65]的实验结果，我们发现单粒子多位翻转大多集中在 2 位翻转和 3 位翻转上，而且多位翻转有着较为固定的可能发生形式。如图 3.8 所示，数据流错误是由于单粒子翻转发生在了软件存储或使用数据的位置时引发的错误，软件可以泛指操作系统本身或者在其上运行的应用程序。依照单粒子翻转的位数和翻转的形式，我们将数据流错误分为以下几种类型。

1） 单位数据流错误翻转。该错误是指程序使用的数据仅有 1 位发生了翻转，在空间环境引发的单粒子翻转事件中，有 80%以上为单位翻转数据流错误。

2） 二位横向数据流错误翻转。该错误对应于横向二位翻转现象，当程序发生此错误时，代表它使用的数据有相邻的两位同时发生了翻转。

3） 二位竖向数据流错误翻转。该错误对应于竖向二位翻转现象，当程序使用顺序数据结构（如数组）时，该数据结构中的两个相邻数据同一位发生了翻转。若程序使用的是非顺序数据结构，由于其内部存储的数据地址不连续，相邻数据不易受到影响。所以本文将之同单位数据流错误翻转处理。

4） 二位交叉数据流错误翻转。该错误对应交叉二位翻转现象，同样在程序使用顺序数据结构时，翻转相邻两个数据的交叉相邻位。

5） 三位横向数据流错误翻转。该错误对应于“一”字型翻转现象，代表翻转数据中三个相邻的位数。

6） 三位竖向数据流错误翻转。该错误对应于“1”字型翻转现象，代表翻转顺序数据结构中三个相邻数据的同一位。

7） 三位 L 型数据流错误翻转。该错误对应于“L”型翻转现象，代表顺序结构的相邻两个数据中，一个翻转了一位，另一个翻转了两位。且这两个数反转的位数中，至少有一位是相同的。

本文依照上述数据流错误类型模拟空间中单粒子翻转所引发的数据流错误。为了尽量贴近真实错误，我们参考文献[64][65]的实验内置了所有错误类型的发生概率，再使用随机数来模拟事件发生情况。本文所用的故障注入框架，是我们基于 GDB 进行二次开发，专用于注入单粒子翻转故障的工具。GDB 是 GNU 软件系统的标准调试器，它具备可移植性，能够在许多类 Unix 系统上运行。GDB 本身也适用于许多编程语言，本文所用的所有测试程序均可由它调试。数据流故障注入工具对目标程序注入故障的流程详细描述如下：

step 1. 设置随机数，根据随机概率选择数据流错误类型模拟单粒子翻转引起的数据流错误。

step 2. 启动 GDB 并加载要在其中注入数据流错误的程序

step 3. 在程序中随机选择一条将注入错误的指令，查看指令内部是否使用数据，若未使用则重新随机选取，直至找到使用了数据的指令为止。

step 4. 指令在程序执行中往往会执行多次，为保持随机性，随机选择故障注入时的指令执行次数，这里记为 n。

step 5. 在所选指令处插入一个断点。

step 6. 运行程序，由于插入了断点每次程序会在执行所选指令时停止执行。此时继续执行 n-1 次程序，直至第 n 次执行该指令时根据所选择的数据流错误类型，适当的注入故障。

step 6.1 若选择单位数据流错误翻转，则在指令数据的二进制位中随机选择一位及逆行翻转。

step 6.2 选择二位横向数据流错误翻转，则在指令数据中随机选择相邻两二进制位进行翻转。

step 6.3 选择二位竖向数据流错误翻转，则需要分析指令数据类型，若为顺序结构数据，则选择此数据与相邻下一数据的同一二进制位进行翻转。若为非顺序结构数据，则只翻转当前数据的一位随机二进制位。

step 6.4 选择二位交叉数据流错误翻转，也需要分析指令数据类型，若为顺序结构数据，则选择此数据与相邻下一数据的相邻两二进制位进行翻转。若为非顺序结构数据，则只翻转当前数据的一位随机二进制位。

step 6.5 选择三位横向数据流错误翻转，则在指令数据中随机选择相邻三位二进制位进行翻转。

step 6.6 选择三位竖向数据流错误翻转，分析指令数据类型，若为顺序结构数据，则选择此数据与相邻下两数据的同一二进制位进行翻转。若为非顺序结构数据，则只翻转当前数据的一位随机二进制位。

step 6.7 选择三位 L 型数据流错误翻转，分析指令数据类型，若为顺序结构数据，则选择此数据与相邻下一数据进行翻转，一个翻转一位另一个翻转相邻两位，且至少有一位重叠。若为非顺序结构数据，则随机翻转当前数据的一位或两位二进制位。

step 7. 根据程序的输出来统计故障注入的结果。

3.5.3 I_{ecSDC}指令识别性能实验及结果分析

对 I_{ecSDC} 指令的识别是 SDCVA-OCSVM 方法的核心所在，也是大部分基于人工智能的 SDC 错误选择保护技术的关键。本文建立的模型参数设置如表 3.10 所示：

表 3.10 算法参数设置

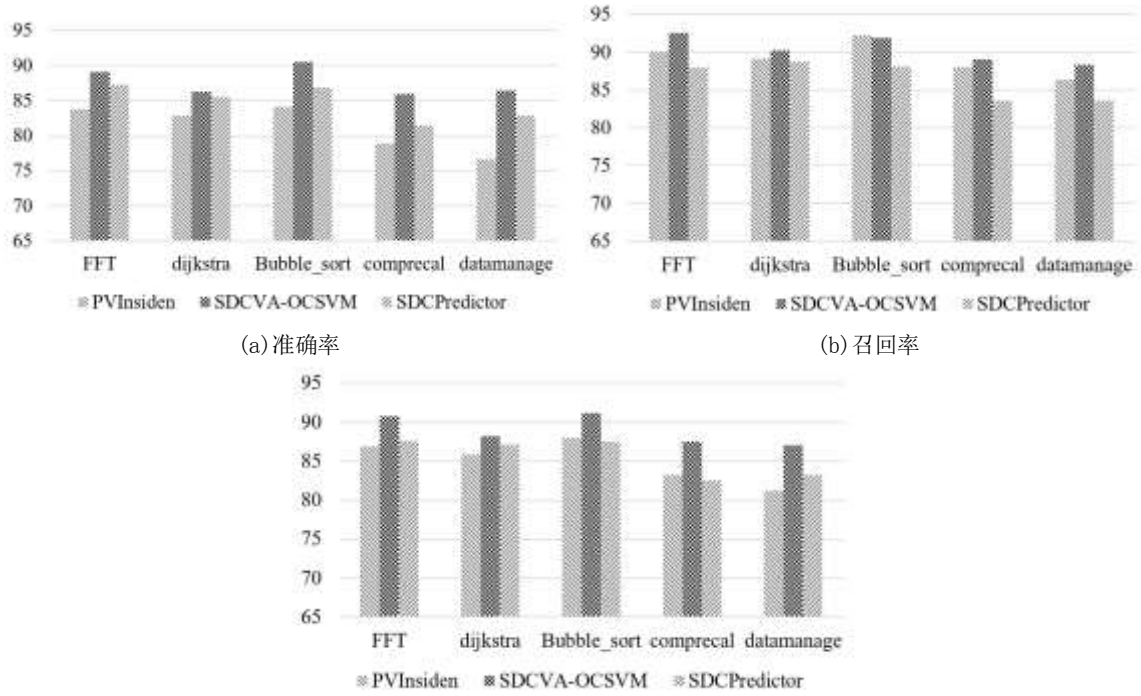
参数名称	参数取值
权衡参数 v	0.15～0.85
高斯核函数参数 σ	0.001～100
随机因子 r_1	0～1
随机因子 r_2	0～1
狼群规模	20
迭代次数	200

结合上述参数及 OCSVM 检测模型，本文进行了指令检测实验。我们将本文提出的 SDCVA-OCSVM 方法和 PVInsiden 方法[75]以及 SDCPredictor[38]方法三者进行对比实验。SDCPredictor 是由 wang 于 2018 年提出的方法，该方法使用随机森林对所有指令的 SDC 脆弱性进行预测，预测出的 SDC 脆弱性较高的指令就可被定义为 I_{ecSDC} 指令。PVInsiden 方法则是于 2019 年提出，该方法则是使用故障注入实验与机器学习相结合的方法来确定 I_{ecSDC} 指令。

评估指标使用精确率(precision)、召回率(recall)以及 f-score。由于 SDCPredictor 方法是对指令的 SDC 诱发率进行预测而非分类，所以对于该方法需要将结果进行二次判别，即将预测结果与 SDC 诱发率阈值 s 进行比较，高于阈值的指令为 I_{ecSDC}，低于阈值的指令为普通指令。

最终得到的实验结果如图 3.9 所示，从实验结果来看，本文所提出的方法要全面优于 PVInsiden，同时大体上也要比 SDCPredictor 方法性能更高。PVInsiden 方法尽管使用了支持向量机，但是其训练样本是根据目标程序随机提取的 30%的指令，方法存在着训练样本不足，特征应用不全等缺点。SDCPredictor 方法存在的主要问题是对指令特征提取存在局限性。SDCPredictor 只提取了指令的静态特征，对于错误在程序中的传播影响考虑不足。本文提出的 SDCVA-OCSVM 方法充分考虑了特征和训练样本两方面的因素。故有较好的效果。

本文提出的 SDCVA-OCSVM 方法尽管效果较好，但对进行指令分类也会产生误差。这是因为 SDC 错误有时会也被其他操作掩盖，也会被一些不影响程序正确结果的意外跳转屏蔽，这些因素往往很难考虑完全，因此本文细化了此类特征以尽量减小误差。



(a)准确率　　(b)召回率

(c) f-score

图 3. 9 不同方法的指令检测性能

3. 5. 4 数据流错误检测实验结果分析

本文使用SDC错误检测率来评估OCSVM-IRVAR方法的SDC错误检测能力。我们定义SDC错误检测率为受到保护的程序在出现 SDC 错误时被加固代码中的检查点检测出来的概率。SDCVA-OCSVM 方法可以通过调整指令 SDC 诱发率阈值 来控制加固开销，阈值S越高加固开销越小，所以 SDCVA-OCSVM 方法可以在用户指定开销的情况下对指令集进行保护。我们随机对测试程序的指令注入故障，以此来模拟程序发生软错误时的状态，实验中向 5 个测试程序分别注入了 5000 次共 25000 次故障来验证 SDCVA-OCSVM 方法的性能。

图 3. 10 给出了采用不同的加固方法下，SDCVA-OCSVM 方法的 SDC 错误检测率的对比。其中 IRVAR-L，IRVAR-M 和 IRVAR-S 是基于数据传输思想的加固方法。IRVAR-C 则是基于程序流程思想的加固方法。从实验结果可以看出，所有的加固方法中 IRVAR-L 具有最高的 SDC 错误检测率，基于数据传输思想的系列加固方法中 IRVAR-M 要比 IRVAR-C 方法的 SDC 错误检测率高。但 IRVAR-C 方法要比 IRVAR-S 方法的错误检测率高。

如图 3. 11 和图 3. 12 所示，IRVAR-C 方法的平均空间加固成本为 60.54%，平均时间加固成本为 70.86%。IRVAR-L 的平均空间加固成本分别为 170.74%，平均时间加固成本为 199.14%。IRVAR-M 的平均空间加固成本分别为 73.72%，平均时间加固成本为 90.54%。IRVAR-S 的平均

空间加固成本分别为 62.44%，平均时间加固成本为 70.84%。

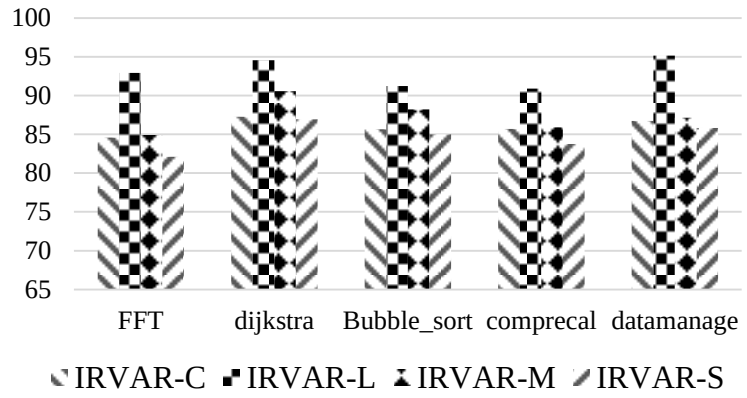


图 3. 10 不同加固方法的 SDC 错误检测率

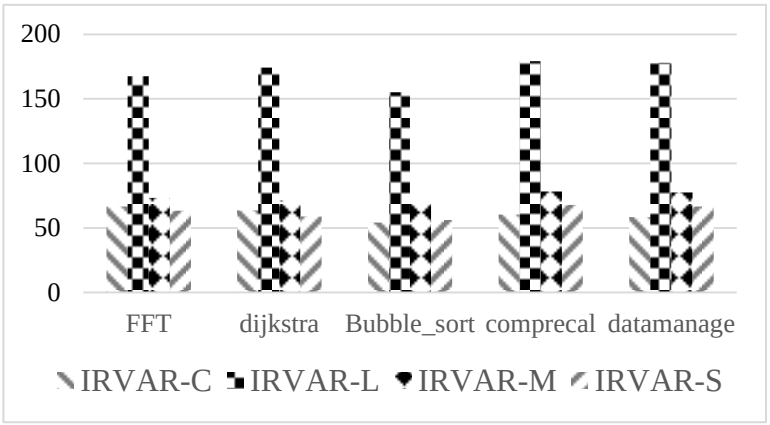


图 3. 11 不同加固方法的空间加固成本

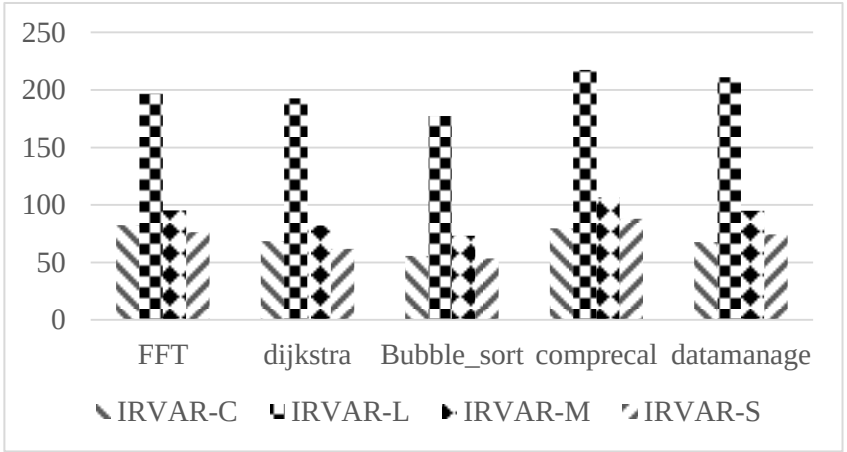


图 3. 12 不同加固方法的时间加固成本

可以看出数据传输系列方法 IRVAR-L 和 IRVAR-M 的加固成本要比 IRVAR-C 方法高。综合图 3. 11 和图 3. 12 的实验结果，我们发现 IRVAR-M 方法固然成本要比 IRVAR-C 方法高，但检错率也有了很大的提升，尤其是针对于类似 comprecal.c 和 FFT.c 这样含有大量计算操作的程序，

提升非常明显。另一方面，和原始 VAR 技术至少两倍以上的加固成本以及未经筛选的全冗余技术 IRVAR-L 接近两倍的成本相比，结合了人工智能识别技术的 SDCVA-OCSVM 方法，又可以将开销控制在一个可以接受的范围。

3.6 本章小结

针对目前基于冗余的 SDC 检错技术开销较大的问题，本文提出了一种新型的 SDC 检错方法 SDCVA-OCSVM。该方法首先通过 Clang 编译器得到目标程序的中间代码指令，再使用指令脆弱性分析及脆弱性特征提取方法计算并提取指令脆弱性特征，这也是算法的核心。通过机器学习，建立了一个具有高准确度的 I_{ecSDC} 指令检测模型，以此识别出程序中易引发 SDC 错误的部分；然后结合了 IRVAR 冗余加固技术，对 指令进行冗余处理；最后，通过实验验证了 SDCVA-OCSVM 方法的有效性。相比同类技术，SDCVA-OCSVM 可以提高更准确的指令识别，更高的错误检测率，以及较低的时空成本。

第四章 基于多层分段标签的控制流错误检测方法

4.1 控制流错误检测理论

4.1.1 相关定义

目前已有的面向软件的控制流错误检测方法大多需要依靠程序控制流程图，它由基本块及基本块之间跳转关系构成的二元组组成。在传统技术中，程序控制流程图只是一种抽象无实体的概念，这严重影响了人们对控制流程图的理解与研究。借用LLVM平台及相关图片生成工具，本文实现了程序控制流程实体化，并成功嵌入到程序加固系统中。在控制流错误检测方面，前人已打下了众多基础，下文给出了本章参考文献[47][48][49][50][54]引入的相关定义：

基本块(Basicblock, b)：基本块是一组满足特定条件的最小有序指令集合，记为 $b=\{I_{in},\ldots I_i\ ,..I_{bra}\}$。程序会按照顺序从基本块内第一条指令 I_{in} 开始执行，直至最后一条指令 I_{bra}。基本块内部除去最后一条指令外，没有其它的跳转指令。除去第一条指令外，没有其他指令跳转对象。本文定义基本块集合为 $B=\{b_{in},...,b_i,...,b_{end}\}$，其中 b_{in} 为程序入口基本块，b_{out} 为程序出口基本块，该集合一般为有序集合，依照程序控制流顺序进行排序。

跳转边(Edge, e)：跳转边记录了程序基本块间的跳转关系，是一条存在于基本块间的有向边，在本文中由 e 来表示，其中 e_{ij} 表示程序控制流为基本块 b_i 至基本块 b_j。

控制流(control-flow, CF)：控制流为软件运行时程序的执行流程，它记录着指令乃至基本块之间的跳转顺序。一般来说，这种跳转顺序是固定不变的，一旦发生改变就有可能产生控制流错误。本文定义控制流为 $E=\{e_{ij}\mid 1\le i,j\le n\}$，其中 e_{ij} 为 b_i 到 b_j 之间的跳转边，n 代表了整个程序中基本块的总数量。

程序控制流程图(Program control-flow graph, PG)：任意一个程序均可表示为程序控制流程图，由二元组 $PG=\langle B,E\rangle$ 表示。其中 $B=\{b_{in},...,b_i,...,b_{end}\}$ 代表程序基本块的集合，$E=\{e_{ij}\mid 1\le i,j\le n\}$ 则代表了程序的控制流。在本文中的程序控制流程图不仅指抽象二元组概念，同时也指代依托LLVM及相关绘图工具所绘制出的程序控制流程图。

前驱(Predecessor)：前驱是指程序控制流中任意一段跳转关系的起点基本块，如在 e_{ij} 中，基本块 b_i 为 b_j 的前驱。

前驱集合(Predecessor set, pred)：前驱集合是指基本块所有前驱基本块的集合，本文使用 $pred$ 来表示，如果 $e_{ij}\in E$，则 $b_i\in pred\left(b_j\right)$。

后继(Successor, succ)：与前驱对应，是指程序控制流跳转时的终点基本块。

后继集合(Predecessor set, pred)：后继集合是指基本块所有后继基本块的集合，本文使用

使用 $succ$ 来表示。如果 $e_{ij} \in E$，则 $b_j \in succ(b_i)$。

控制流错误(control-flow error, CFE)：软件运行时，因单粒子翻转影响使计算机系统发生了瞬时故障，从而导致软件脱离了正常的程序流程，发生错误跳转。这种软件故障被称为控制流错误。设指令 $I_i \in b_i = \{I_{i_1}, I_{i_2}, ..., I_{i_{end}}\}$，在程序中存在指令 $I_i \to I_j$ 之间的合法跳转，则指令 I_j 需满足以下两个条件：(1)若 $i < i_{end}$，则 $j = i+1$;(2)若 $i = i_{end}$，则存在 $b_j = \{I_{j_1}, I_{j_2}, ..., I_{j_{end}}\} \in succ(b_i)$，其中 $I_j \in b_j$ 且 $j = j_1$。若不满足上述两个条件，证明程序发生了控制流错误。

基本块标签(Basicblock signature, BS)：本文中会为每一个基本块匹配一个具有唯一性的标签，当程序正确执行时，动态全局标签需要与其相等。

动态全局标签(Global signature, GS)：程序运行时的动态标签，会根据相应的标签规则进行动态更新，当程序正确执行时，更新后的 GS 要与 BS 相等。

除去上述引入的定义外，本文创建的相关定义如下：

前驱序列(Predecessor sequence, prseq)：与前驱集合不同，前驱序列是一种基本块有序集合。某基本块的前驱序列是指在程序的某一条控制流中，按照执行顺序所有处于该基本块前方的基本快的集合，并依照与该基本块的距离从近至远排序。如控制流 $\{b_1 \to b_2 \to b_3 \to b_4\}$ 中，b_4 的前驱序列为 $\{b_3, b_2, b_1\}$，另外需要注意的是，基本块的前驱序列不唯一。

基本块类型(Basicblock type, BT)：本文中任意一个基本块均可被分为 O 型或者 M 型。

O 型基本块(Type O basicblock, TO)：O 型基本块代表它至多有一个前驱并且至多只有一个后继。

M 型基本块(Type M basicblock, TM)：基本块若不是 O 型，则为 M 型。

差值参数(difference parameter, d): 差值参数与每个基本块都有所关联，该值主要用于更新全局动态标签 GS，使程序在正确执行时，GS 能更新成期望值 BS 而不会出现错误，若标签更新语句为二元运算时，差值参数一般作为除 GS 外的另一元出现。在本文使用 d 来表示差值参数，如 $d(b_i)$ 代表的是基本块 b_i 的差值参数。

分段差值参数：由于标签的特殊性，本文所用的差值参数也分为前后两段，且与标签相同，分为层号段与标签值段。层号段记为 d_1，标签值段记为 d_2。

层级(level, Lv)：层级是一组非空基本块的集合，根据后文中基本块按层级划分相关规则，重划分后的程序不会连续执行 M 型基本块。以 M 型基本块为核心，所有该 M 型基本块的前驱序列中从头至尾连续的 O 型基本块和该 M 型基本块处于同一层级，直至遇到另一个 M 型基本块为止。若某一基本块同时处于多个层次，则该基本块属于该层级集合，即基本块可以同时属于多个层级。

基本块多前驱矛盾：也被称为别名问题，是目前所有控制流标签算法所遇到的普遍难题。具体是指当一个基本块同时具备多个前驱时，不同前驱跳转至同一基本块需要让不同的标签值

通过同一更新语句获得同一标签值。但由于标签具有唯一性，标签更新语句也大多具备唯一性，这种唯一性与上述要求产生了矛盾。

M 不连续原则：对于程序中的任意一条控制流，在依照正确顺序运行的情况下，不存在两个或两个以上连续执行的 M 型基本块。

1 位&1 位集合：1 位是指本文所设计的二进制标签中数值为 1 的位置(位数从 0 开始，并且从右向左计算)，1 位集合是指标签中所有 1 位的集合，本文中将 1 位集合表示为1_{set}。如：011 的 1 位集合为{0,1}，110 的 1 位集合为{1,2}，即$1_{set}(011)=\{0,1\},1_{set}(110)=\{1,2\}$。

1 位包含：1 位包含是指某一标签的 1 位集合包含于另一标签的 1 位集合。如：010 的 1 位集合包含于 011 的 1 位集合，即$\{2\}\subset\{1,2\}$。

多层分段标签：本文所设计的多层分段标签是一种较为精巧的标签，本文通过分段设计使其具备了双标签的功效。标签共分为前后两段，分别为层号段与标签值段。层号段标识基本块所属层次，记为BS_1。标签值段则在同层内标识不同的基本块，记为BS_2。

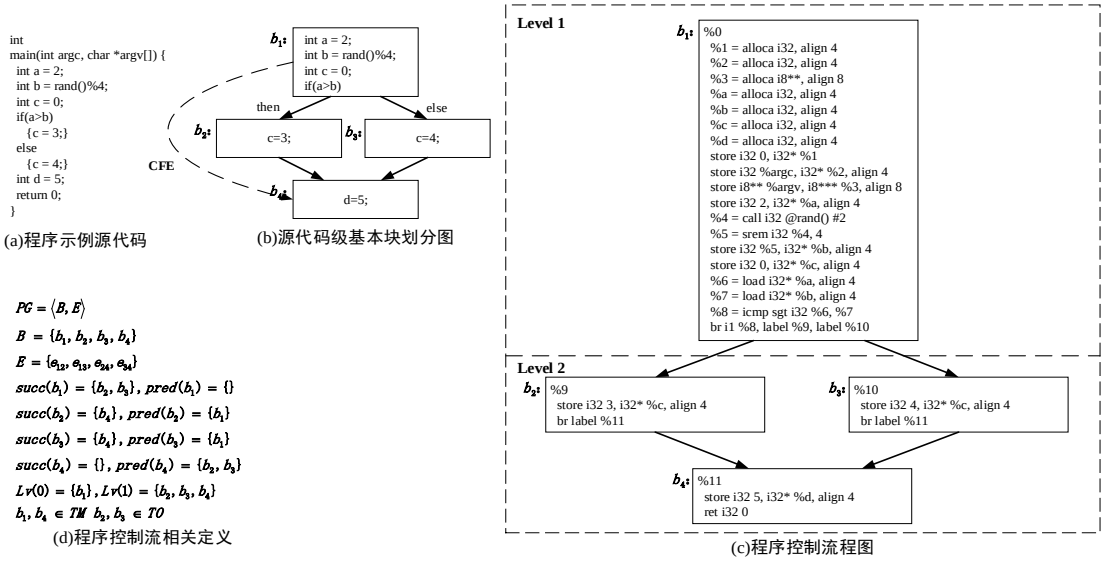


图 4. 1 程序控制流示例

图 4. 1 程序控制流示例 以数据管理软件的部分逻辑代码为例，给出上述程序控制流相关定义的进一步说明。图 4. 1 程序控制流示例 (a)为示例代码，整个代码跳转由 if 语句实现。图 4. 1 程序控制流示例 (b)为源代码级基本块划分图，但是在程序源代码内部插桩标签开销过大，所以一般会将程序进一步编译成中间代码。图 4. 1 程序控制流示例 (c)为程序控制流程图 PG，图 4. 1 程序控制流示例 (d)为基本块定义之间的符号关系表示。整个程序由%0，%9，%10 和%11 四个基本块构成分别记为b_1,b_2,b_3,b_4，其中b_1,b_4为 M 型基本块，b_2,b_3为 O 型基本块。依照程序正常执行流程，基本块b_1的后继为b_2和b_3。由于程序内变量 b 为 0 至 3 之间的随机数，与 a 之间的比较也有着随机性，故跳转至b_2或b_3均为正常控制流。若程序执行过程中受单粒子翻

转影响发生错误跳转，如图 4. 1 程序控制流示例 (b)中由 b_1 跳至 b_4 ，此时变量 c 的值仍为 0，发生控制流错误。

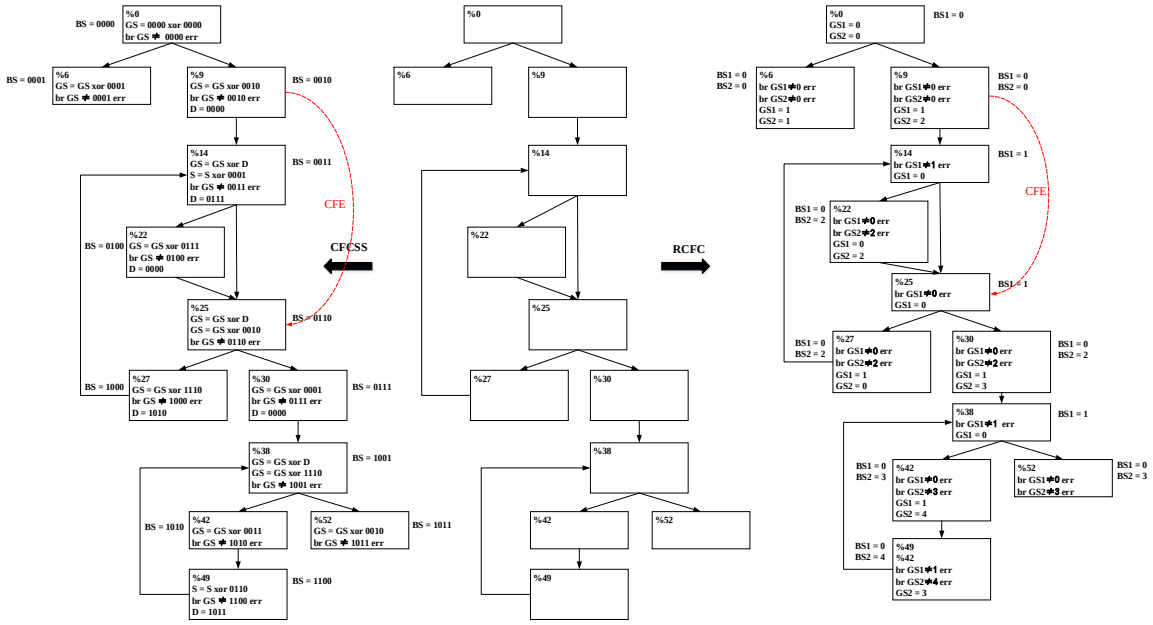
4. 1. 2 传统的控制流错误检测技术

就目前的控制流错误检测技术中，已经有许多纯软件技术。而在这些技术中，较为经典的是 CFCSS 技术[47]和 RCFC 技术[51]。前者作为最早的一批标签控制流错误检测方法，已经验证了该技术的有效性。后者则是在标签验证基础上，提供了一种新型的标签设计思路，下面本文将详细介绍两种算法。

CFCSS 属于一般典型的控制流错误检测技术，它会维护一个实时的标签寄存器，RCFC 则因为要使用两个标签的原因，多使用了一个寄存器。CFCSS 寄存器内部存储着一个全局动态标签 GS，RCFC 的两个寄存器则分别存储着标签 GS1 和 GS2。它们均会随着程序的运行以及节点间的跳转而不断更新和验证。两种技术首先会解析整个程序的结构并以此构建程序控制流程图，之后再为程序控制流程图中的每一个基本块分配标签 BS，对于 RCFC 则是 BS1 和 BS2，它们也是动态标签运行至该基本块的期望值。无论程序执行到内部任何基本块的任意一条指令，这种全局动态标签与期望的值必须相等。在程序运行时，若 GS(GS1 与 GS2 同理)等于其期望值 BS，代表程序正确执行，否则代表程序发生了控制流错误。

图 4. 2 给出了 CFCSS 技术和 RCFC 技术标签分配的示例。图 4. 2(a)为数据管理软件的排序算法的基本块框架，本文以之为例，给出了两种标签分配策略的具体流程及详细说明。图 4. 2 (b)为 CFCSS 标签分配示例图，CFCSS 方法会对程序中的每一个基本块分配一个具备唯一性的基本块标签 BS，如%0 基本块的 BS 为 0000，该标签一般为二进制 32 位，位数可以随着平台更换而改变。BS 同时也是全局动态标签 GS 运行至该基本块时的期望值，当程序正确执行时，要使得 GS 与 BS 保持相等。标签更新指令选择异或运算，与其它二元运算不同，异或运算是占据资源最少且具有结果唯一性的运算。但是在使用异或运算作为标签更新指令时，存在基本块多前驱矛盾问题。CFCSS 会为每一个基本块分配唯一的标签，异或运算又具有唯一性。当一个基本块同时存在多个前驱时(如%14、%22 和%25 基本块)，无法使不同的标签通过同一异或运算更新成同一标签。为了解决这一矛盾，CFCSS 引入了变量 D。这样在多前驱基本块更新标签前，首先和变量 D 进行异或运算，统一此时的全局动态变量 GS，此方法虽然解决了多前驱矛盾，但也增加了部分开销。CFCSS 将检查指令放在更新指令之后，在程序运行时可以比较全局动态标签 GS 与基本块标签 BS，相等则证明程序正确执行，否则发生了控制流错误。如图 4. 2(b)所示，%9 的标签值为 0010，若按照正确控制流程跳转到%14 基本块，通过运算 0010 xor 0001 = 0011 更新至%14 的基本块标签值，通过检查指令结果正确。若发生如图中的 CFE，此时通过运算 0010 xor 0010 = 0000，通过检查指令发现结果错误，证明此时程序发生了控制流错误。与 CFCSS 不

同，RCFC 设计了两种标签 GS1 与 GS2，标签分配示例如图 4. 2 (c)所示。同样为了解决基本块多前驱矛盾问题，RCFC 在含有多前驱的基本块上只插桩了 GS1 标签，同时更新语句选用赋值语句，而非运算。虽然也解决了多前驱的矛盾，但失去了标签的唯一性。这导致了部分控制流错误无法检测，降低了方法的错误检测率。



(b) CFCSS 标签分配示例　　(a) 示例程序控制流程图　　(c) RCFC 标签分配示例

图 4. 2 标签技术示例图

目前大部分的控制流错误检测方法的原理和上述两种方法的原理大体相似，只是标签设计的结构、规则和位置有所不同，各个方法的检测率与时空开销的差异也来源于此，就目前而言，基于标签的控制流错误检测算法普遍存在着开销与漏检率矛盾的问题，如何权衡两者以达到最好的检测效果，是本文研究的重点。本文对已有的控制流方法进行改进，提出了一种多层分段标签控制流错误检测方法(CFMSL)，后文将对此展开详细介绍。

4. 2 多层分段标签控制流错误检测方案设计

本文设计的分层多段标签控制流错误检测方案架构如图 4. 3 所示，由数据层、加固层、评估层组成。

数据层为插桩检错代码的加固过程提供数据支撑服务。其中，测试程序集主要作为目标程序通过本文实现的多层分段标签控制流错误检测方法转换成具有检错能力的目标程序；程序信息库则包含了方法所需要的目标程序前置信息，包括其基本块结构，前驱后继关系等。故障注入结果库存储了控制流故障注入的实验结果，在本文中共提供了两种故障注入的方式：随机控

制流故障注入与程序内部控制流故障注入；标签库用来存储为测试程序集中的目标程序所分配的所有标签值，是多层分段标签控制流错误检测方法进行错误检测和软加固的基础。

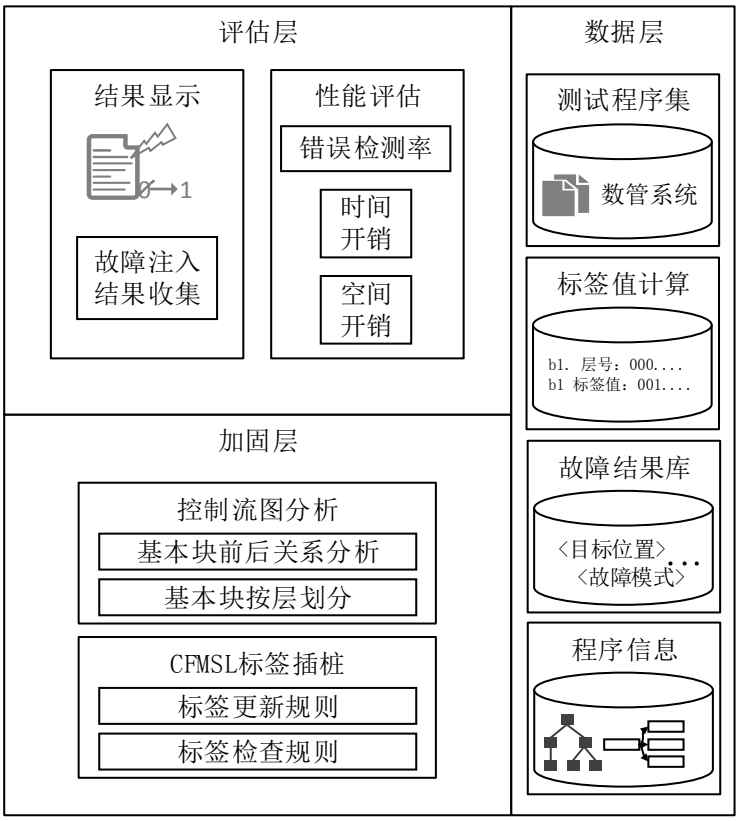


图 4.3 多层分段标签控制流错误检测架构图

加固层位整个控制流检错的核心层次，这一层主要功能是利用多层分段标签控制流错误检测方法对目标程序进行加固。本层包括控制流图分析模块，主要用于分析目标程序的基本块前后关系和层次划分。CFMSL 插桩模块会依据方法的标签更新与检查规则，插入多层分段标签控制流错误检测指令，使程序具有控制流检错能力。

评估层包括了结果显示模块和性能评估模块。结果显示模块会收集故障注入的结果，并对结果进行统计。本文中的控制流故障注入工具使用 GDB 等工具进行二次开发实现。性能评估模块将会对 CFMSL 加固后的程序进行性能和开销分析，为评估 CFMSL 方法进行参考。

开销进行评估，最终验证方法的有效性。

使用本文实现的分层多段标签控制流错误检测技术可以对目标程序进行控制流检错加固，具体流程如图 4.4 所示。详细描述如下：

（1）程序控制流分析重构阶段

CFMSL 方法检错加固开始阶段，首先要使用 LLVM 中的前端编译器 Clang 将目标程序编译成中间代码形式。在此阶段，也可使用 dot-cfg 工具生成中间代码的控制流程图，分析整个程序中间代码的结构，划分基本块。依据基本块按层划分规则，按层划分目标程序的基本块。

（2）程序控制流加固阶段

综合所有程序控制流信息，包括基本块类型、相互间依赖关系、层次结构等，为每一个基本块分配多层分段标签；并根据分配的标签插桩标签更新指令以及标签检查指令，最后输出具有控制流错误检测能力的目标程序。

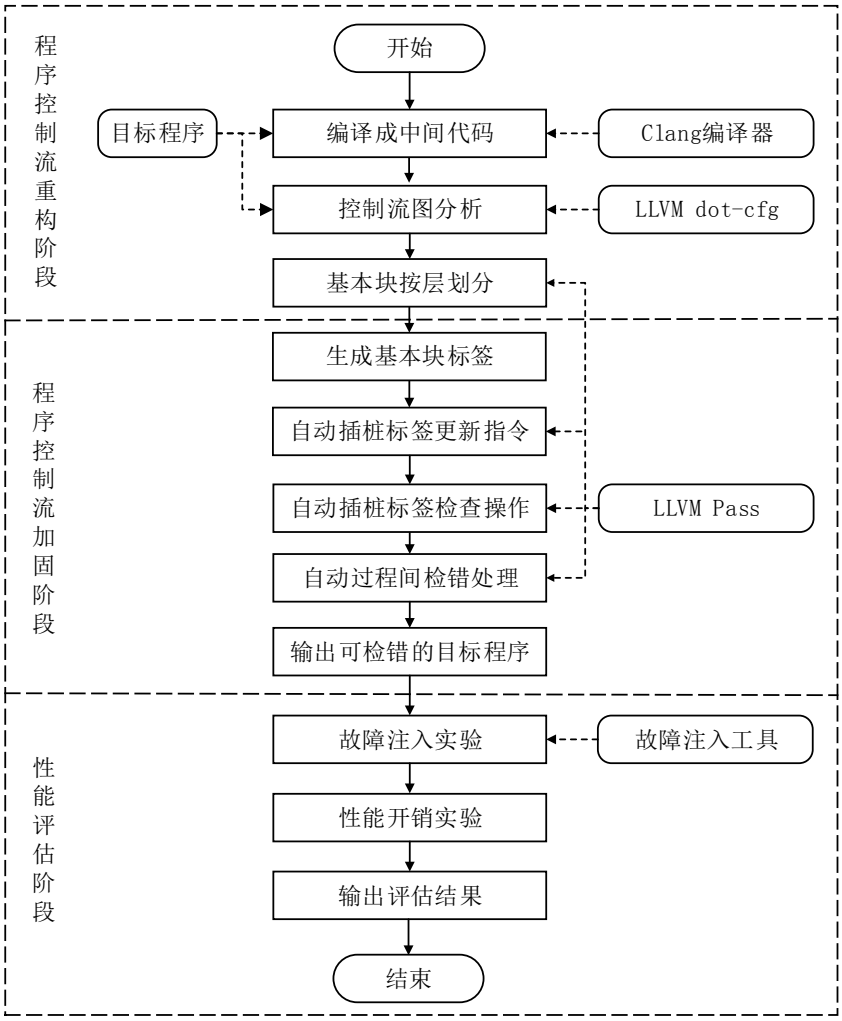


图 4.4 多层分段标签控制流错误检测流程图

（3）性能评估阶段

对加固后的程序使用控制流故障注入工具进行故障注入实验。根据实验结果，对 CFMSL 方法的性能和多层分段标签控制流错误检测规则设计

4.2.1 基本块按层次划分规则

依照程序控制流程图中基本块间的前驱后继关系，将基本块分类两类，M 型基本块与 O 型基本块。为了后续标签进行层次划分以及层号设置，本文设计的分层标签需要程序满足一种特定的控制流跳转规律：M 型基本块和 M 型基本块不可连续执行（M 不连续原则），并以此为基

础设计标签生成、更新及校验规则。因此，对于不满足跳转规律的程序，则需要使用基本块按层次划分规则来改变程序控制流结构。

规则 1：若程序中存在一条控制流，其中有两个 M 型基本块连在一起，则替换其原本跳转路径，在二者之间插入一个空基本块。

记为：$b_i, b_j \in TM$ and $b_i \in pred(b_j)$．Then $b_p \in pred(b_j), b_p \in succ(b_i)$ and $b_i \notin pred(b_j)$，$b_j \notin succ(b_i)$.

规则 2：基本块按照规则 1 划分完成后，以 M 型基本块为核心，所有该 M 型基本块的前驱序列中从头至尾连续的 O 型基本块和该 M 型基本划分为同一层，直至遇到另一个 M 型基本块为止。若一基本块同时位于多个层级，则该基本块所属层级为层级集合。对于任意一个程序，若入口基本块为 M 型基本块，则将其单独划分为 1 个层级。若入口基本块为 O 型基本块，则将之归入程序流中的第一个 M 型基本块所处的层级。除去入口基本块外，程序中每一层级均由 M 型基本块以及其前驱序列中的 O 型基本块构成。

引理 1：通过规则 1 对程序控制流重构后，得到的新程序控制流程图中，不存在可以连续执行两个 M 型基本块的控制流。

即：任意 $b_i \in B$，若 $b_i \in TM$，则 $\left(succ\left(b_i\right) \subset TO\right) \vee \left(pred\left(b_i\right) \subset TO\right)$.

证明：对任意 $b_i \in TM$，若存在 $b_j \in succ(b_i) \subset TM$，则根据规则 1 调整，必存在 O 型空基本块 b_p，使得 $b_p \in pred(b_j), b_p \in succ(b_i)$ 并且 $b_i \notin pred(b_j), b_j \notin succ(b_i)$，这与 $b_j \in succ(b_i)$ 产生矛盾。因此，任何经过规则 1 调整后的程序，一定不存在可以连续执行两个 M 型基本块的控制流。

图 4. 5 为基本块层次划分规则应用的典型情况。其中，图 4. 5 (a)，图 4. 5 (b)和图 4. 5 (c)分别对应规则 1 与规则 2。虚线所代表的基本块表示新插入的空基本块。图 4. 5 (a)中 b_2 和 b_3 均为 O 型基本块，且同为 b_5 的连续非 M 型前驱序列，故根据规则 2 将 b_5 与 b_2 和 b_3 划分到了同一层中。图 4. 5 (b)中 b_1 和 b_3 均为 M 型基本块，整个程序有两条控制流 $\{b_1 \to b_2 \to b_3\}$ 与 $\{b_1 \to b_3\}$,其中 $\{b_1 \to b_3\}$ 不满足 M 不连续原则，故根据规则 1 在控制流 $\{b_1 \to b_3\}$ 中插入空基本块 b_p 以重构控制流。图 4. 5 (c)中有一条控制流从 b_4 又跳回了 b_1，导致 b_1 同时归属两个层级 Lv1, Lv3。

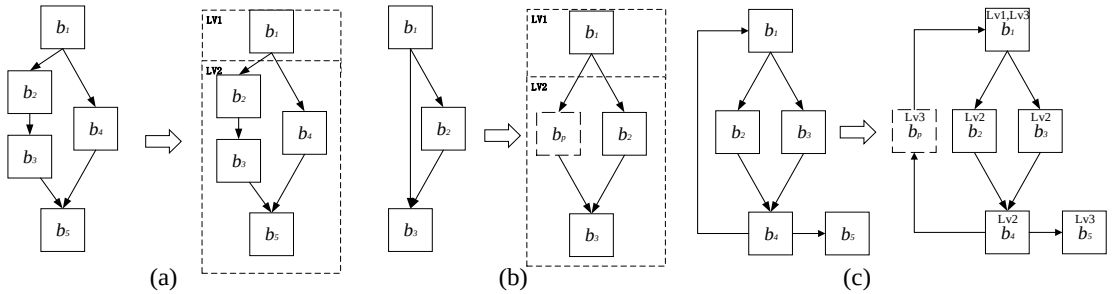


图 4. 5 基本块按层次划分示例

4.2.2 多层分段标签分配规则

本文采用静态标签插桩技术，插桩全局动态标签 GS(Global signature)，使程序执行时连续的更新 GS，并随时检查标签以监视程序的运行状态。这一过程可以被概括的分成三阶段：标签分配阶段、标签更新阶段以及标签检查阶段。其中标签分配阶段会为程序中的每一基本块静态的分配或计算相对应的基本块标签值；标签更新阶段则是根据标签分配规则为每一基本块内添加标签计算语句，使程序正确运行时得到正确的标签，错误运行时只能得到错误的标签。标签检查阶段会为程序基本块内部添加检查指令，以保证一旦发生标签错误，程序可以尽早的检测出来。本文设计的多层分段标签与传统的标签方法不同，它使用科学合理的标签分配策略以及特殊标签形式，在提高错误检测率的同时降低了标签开销。

本文设计了用于标签分配的具体规则，给出了相关定理，为了方便描述，此处引入了下列术语：

作为标签检错技术的一种，多层分段标签控制流错误检测方法也不可避免地遇到了一个问题：基本块多前驱矛盾。CFCSS 为了解决这一问题增加开销引入了变量 D，RCFC 为了解决这一问题放弃了标签唯一性从而造成了检错率的下降。多层分段标签控制流错误检测方法利用特殊的标签机制及科学的更新策略，在解决这一矛盾的同时未增加任何开销，同时也保证了标签的唯一性，以下为我们设计的多层分段标签的具体规则。

规则 3：程序执行时，全局动态标签 GS 的更新语句在基本块入口位置，O 型基本块执行异或运算，M 型基本块执行或运算。即：

$$GS = GS\ xor\ d(b_i) \quad if\ b_i \in TO$$

$$GS = GS\ or\ d(b_i) \quad if\ b_i \in TM$$

规则 4：程序内所有 O 型基本块标签的 1 位集合包含于其后继 M 型基本块标签的 1 位集合。即：$\forall b_i, b_j \in B$，若 $b_i = pred(b_j)$，$b_i \in TO$ 且 $b_i \in TM$，则 $1_{set}(BS(b_i)) \subset 1_{set}(BS(b_j))$。

规则 5：若基本块为 O 型基本块，其分段差值参数为该基本块分配标签与其前驱基本块分配标签异或运算的结果。即：

$\forall b_i, b_j \in B$，若 $b_i \in TO$，$b_j \in pred(b_i)$ 且 $b_i \in succ(b_j)$，则 $d(b_i) = BS(b_i)\ xor\ BS(b_j)$。

若基本块为 M 型基本块，其分段差值参数层号段为 0，标签值段为该基本块所有前驱标签值段或运算的结果。即：$\forall b_i, b_{j1}, b_{j2}, ..., b_{jn} \in B,$ 若 $b_i \in TM$，$b_{j1}, b_{j2}, ..., b_{jn} \in pred(b_i)$，则 $d(b_i) =$

$$BS(b_{j1})\ or\ BS(b_{j2})\ or\ ...\ or\ BS(b_{jn})。$$

规则 6：所有 M 型基本块标签中标签值段为该基本块所有前驱基本块标签中标签值段或运算的结果。即：$\forall b_i, b_{j1}, b_{j2}, ..., b_{jn} \in B,$ 若 $b_i \in TM$，$b_{j1}, b_{j2}, ..., b_{jn} \in pred(b_i)$，则 $BS_2(b_i) =$

$$BS_2(b_{j1})\ or\ BS_2(b_{j2})\ or\ ...\ or\ BS_2(b_{jn})。$$

规则 7：将程序内部所有基本块按层次划分，并在标签层号段标识层号。层号之间必须唯

一，且互相满足 1 位不包含原则。一般来说，层号仅取标签层号段的某一位为 1。即：

$\forall b_i, b_j \in B$, 若 $Lv(b_i) \neq Lv(b_j)$，则 $(1_{set}(BS_1(b_i)) \not\subset 1_{set}(BS_1(b_j))) \cup (1_{set}(BS_1(b_j)) \not\subset 1_{set}(BS_1(b_i)))$

规则 8：若程序中存在循环控制流，即某一基本块同时处于多个层级中，则该基本块必为 M 型基本块，且其层号为其所有前驱基本块层号 or 运算的结果。即：

$\forall b_i \in B,\ b_{j1}, b_{j2}, ..., b_{jn} \in pred(b_i)$, 则 $BS_1(b_i) = BS_1(b_{j1})\ or\ BS_2(b_{j2})\ or\ ...\ or\ BS_2(b_{jn})$

规则 9：不同层级间基本块标签值段可重复，但同一层基本块间标签值段必须唯一，且该层所有O型基本块之间互相满足1位不包含原则，M型基本块之间也互相满足1位不包含原则，非前驱后继关系的 O 型基本块与 M 型基本块也互相满足 1 位不包含原则。即：

$\forall b_i, b_j \in B$, 若 $Lv(b_i) \neq Lv(b_j)$，则 $BS_2(b_i) \neq BS_2(b_j)$；

若 $b_i, b_j \in TO$, 则 $(1_{set}(BS_2(b_i)) \not\subset 1_{set}(BS_2(b_j))) \cup (1_{set}(BS_2(b_j)) \not\subset 1_{set}(BS_2(b_i)))$；

若 $b_i, b_j \in TM$, 则 $(1_{set}(BS_2(b_i)) \not\subset 1_{set}(BS_2(b_j))) \cup (1_{set}(BS_2(b_j)) \not\subset 1_{set}(BS_2(b_i)))$；

若 $b_i \in TO$，$b_j \in TM$，且 $b_i \notin pred(b_i)$, 则 $(1_{set}(BS_2(b_i)) \not\subset 1_{set}(BS_2(b_j))) \cup (1_{set}(BS_2(b_j)) \not\subset 1_{set}(BS_2(b_i)))$

规则 10：程序内每一个基本块均要有标签检查指令，位置在标签更新指令之后，用于比较全局动态标签 GS 与基本块标签 BS 的值以检测控制流错误。

上述规则为多层分段标签控制流错误检测方法的基本规则，也是必要规则。规则 3-6 是多层分段标签控制流错误检测方法可以获得较低开销的基础，本文将程序内部基本块划分为两种类型：O 型基本块与 M 型基本块。程序执行过程中，由于 O 型基本块只具备一个前驱，故基本块多前驱矛盾都集中发生在 M 型基本块中。所以我们的多层分段标签控制流错误检测方法所做的第一个改进就是在 M 型基本块的标签更新语句中使用 or 运算来替换 xor 运算。但是 or 运算本身并不具备唯一性，盲目使用 or 运算会导致某些控制流错误无法检测出来。如图 4. 6(a)所示，依照规则 3-6 分配了标签，通过 or 运算解决了 $b_2 \to b_4$ 和 $b_3 \to b_4$ 的基本块多前驱矛盾问题。但若发生 $b_1 \to b_4$ 的控制流错误跳转，就无法通过该标签检测出来。为了解决该问题，本文引入了层号的概念，并在标签的设计上做出了调整。如图 4. 6 (b)所示，随着层号的引入，标签已经可以检测出程序中发生的 $b_1 \to b_4$ 的控制流错误。由于层号本身的引入是为了解决 or 运算结果不唯一，造成标签更新不唯一的问题，故在设计时也要遵循一定规则。否则如图 4. 6(c)所示，即便引入了层号，也无法解决问题。因此，本文提出了规则 7。对于整个程序而言，层号之间必须有所区分且每一个层号都是唯一的。在此基础上，也不允许存在 1 位集合包含的情况，即任意的两个不同的层号之间，1 位集合互不包含。层号不允许设置为 0，而一般来说，层号仅取标签层号段中的某一位为 1 即可。如程序共分为三层，则层号可分别设为：001，010，100。程序的循环结构是一种较为常见的结构，体现到程序控制流程图中就代表会出现如图 4. 6(d)中 b_1 的结构。这种情况下使得基本块 b_4 同时处于了 010 层和 100 层中，故层号也包含了 010 和 100。为了解决这一现象，本文引入了规则 8，依照此规则此时基本块 b_4 的所属层号为 $010\ or\ 100 = 110$，

同时兼顾了 $b_2 \to b_4$、$b_3 \to b_4$ 和 $b_5 \to b_4$ 的控制流跳转。

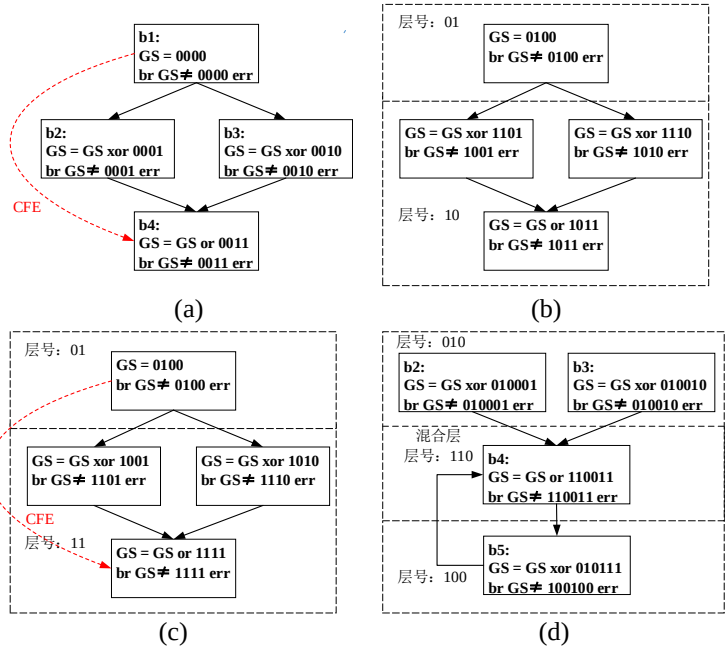


图 4. 6 多层分段标签控制流错误检测规则 3-8 示例

规则 3-8 解决了控制流跨层错误跳转的问题，但未顾及同层错误跳转问题，尤其是当同一层含有 2 个或 2 个以上的 M 型基本块的情况。如图 4. 7(a)所示，这种“W”型控制流虽然在程序中并不常见，却是一种很好的标签检验结构。b_2，b_4 和 b_5 均为 M 型基本块，在程序中存在两条控制流 $b_2 \to b_4$ 和 $b_2 \to b_5$，它们均连续执行了两个 M 型基本块，这违反了 M 不连续原则。依照基本块按层划分规则 1，本文在上述两条控制流中插入了空基本块 b_{p1} 和 b_{p2}，重构了程序的控制流层，并按照规则 2 划分层次。如图 4. 7 (b)所示，在仅考虑规则 3-8 的情况下，本文设计了一款样例标签。可以明显看出，在同层次中层号不起作用的情况下，或运算结果不唯一特性又导致了程序控制流错误漏检。图 4. 7(b)中，标签无法检测出 $b_1 \to b_5$ 或者 $b_4 \to b_5$ 的控制流错误跳转。为了解决这种同层次标签漏检问题，我们设计了规则 9。依照规则 9 的规定，同一层的 O 型基本块间标签值段必须唯一，且满足互相 1 位不包含原则，M 型基本块同理。图 4. 7 (b)程序中的 O 型基本块b_1的 1 位集合被同层 O 型基本块 b_2 和 b_3 所包含，M 型基本块 b_4 的 1 位集合被同层 M 型基本块 b_5 所包含，故产生了上述两条无法被检测的控制流错误。在重新考虑规则 3-9 后，本文重新设计了如图 4. 7 (c)所示的标签方案，解决了上述问题。

接下来，本文将会引入一些定理来证明本文提出的多层分段标签方法的安全性及完备性。安全性是指当程序正确执行时，多层分段标签方法所分配的标签不会检测出错误。完备性则是指若程序在执行过程中发生了基本块间错误跳转，多层分段标签方法可以将其检测出来。

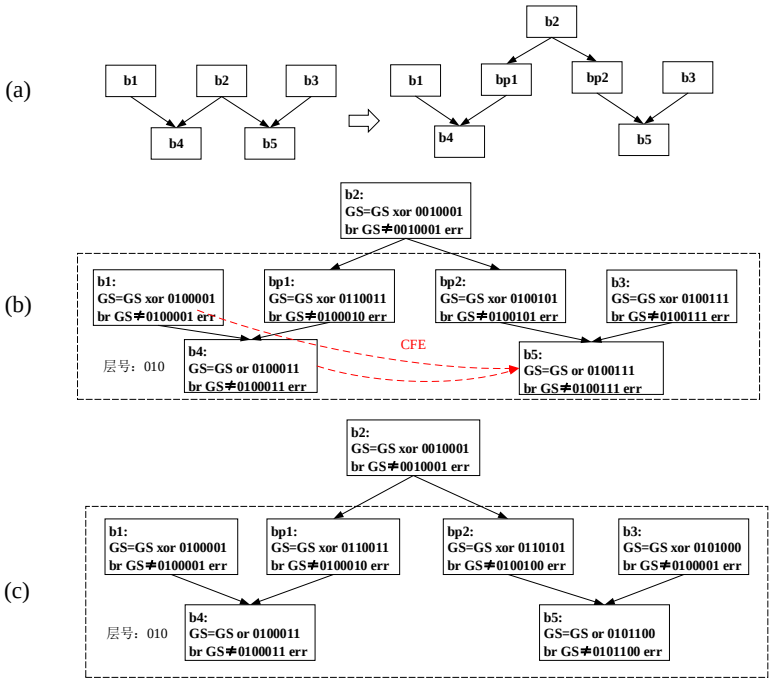


图 4. 7 多层分段标签控制流错误检测规则 9 示例

引理 2 通过规则 1 对程序控制流重构后，得到的新程序控制流程图中，任意 M 型基本块均与其前驱基本块处于同一层次。

证明： M 型基本块为层次划分的核心。若存在一基本块为该 M 型基本块前驱，且和该基本块不属于同一层次。根据规则 2 所有 M 型基本块的前驱序列中从头至尾连续的 O 型基本块和该 M 型基本划分为同一层可知，该前驱基本块只能为 M 型基本块。根据引理 1，M 型基本块的前驱不可以为 M 型基本块，产生矛盾。因此，不存在这样的基本块前驱，即任意 M 型基本块均与其前驱基本块处于同一层次。

定理 1 当程序正确执行时，若标签在进入基本块前满足 $GS = BS$，则无论程序执行至至基本块内任何位置，仍满足 $GS = BS$。

证明： 设 $\forall b_i, b_j \in B,$，且程序中存在控制流 e_{ij}，即 $b_j \in pred(b_i)$ 且 $b_i \in succ(b_j)$。当全局动态标签 GS 到达基本块 b_i 时 $GS = BS(b_j)$，所执行的标签更新语句取决于此时基本块的类型。根据定义，程序中任意基本块均属于 O 型或者 M 型，故分两种情况讨论：

(1) $b_i \in TO$

根据规则 3，此时基本块的更新语句为

$$GS = GS \ xor \ d(b_i) = BS(b_j) \ xor \ BS(b_i) \ xor \ BS(b_j) = BS(b_i)$$

(2) $b_i \in TM$

根据规则 3，此时基本块的更新语句为 $GS = GS \ xor \ d(b_i)$。分段来看，根据规则 8，GS 层号段 $GS_1 = GS_1 \ xor \ d_1(b_i) = BS_1(b_{j1}) \ or \ BS_1(b_{j2}) \ or \ ... \ or \ BS_1(b_{jn}) = BS_1(b_i)$；根据规则 6，$GS$

标签值段 $GS_2 = GS_2 \ or \ d_2(b_i) = BS_2(b_{i1}) \ or \ BS_2(b_{i2}) \ or \ldots \ or \ BS_2(b_{in}) = BS_2(b_i)$。综合两者，此时 $GS=GS_1+GS_2=BS_1(b_i)+BS_2(b_i)=BS(b_i)$。

综上所述，此时的全局动态标签在基本块任何位置均满足 $GS=BS$。

引理 3 若因程序发生错误跳转而导致全局动态标签层号段在程序中任何一处与基本块标签层号段不同，则此错误会被检查出来。

证明：若因程序发生错误跳转而导致层号与期望值不同，证明此时发生了跨层错误跳转。错误产生后，*GS* 会在基本块入口处进行标签更新。*GS* 抵达不同类型的基本块，会遇到不同的标签更新运算。若抵达基本块为 O 型基本块，则更新语句为 $GS=GS \ xor \ d$。异或运算的结果具备唯一性，已有的层号段错误不会被掩盖，故此错误可以被检查出来。若抵达基本块为 M 型基本块，则更新语句为 $GS=GS \ or \ d$。根据规则 7 可知，此时层号段虽然与期望值不同，但也满足与期望值层号段 1 位互不包含的原则。根据规则 8，更新语句中的差值参数 d 层号段 1 位与此时基本块所处层次 1 位相同，即 d_1 和 *GS* 的 1 位不同。显然，1 位是无法被或运算掩盖的，故此错误也能被检查出来。

引理 4 若因程序发生错误跳转而导致全局动态标签标签值段在程序中任何一处与基本块标签标签值段不同，则此错误会被检查出来。

证明：根据引理 3，全局动态标签层号段出现错误会被检查出来，所以现在考虑层号段未出错，标签值段出错的情形。由于层号段未出错，证明程序发生的是同层错误跳转。依旧根据 *GS* 抵达基本块的类型进行分类讨论。若抵达基本块为 O 型基本块，则更新语句为 $GS=GS \ xor \ d$。异或运算结果具备唯一性，而根据规则 9 同层基本块标签值段也具备唯一性，故此时的标签值段错误不会被掩盖。若抵达基本块为 M 型基本块，则更新语句为 $GS=GS \ or \ d$。因发生了同层错误跳转，又跳到了 M 型基本块，故此时也有两种情况：第一种是 O 型基本块错误跳转到 M 型基本块，由于是同层错误跳转，此 O 型基本块和 M 型基本块不是前驱后继关系。第二种是 M 型基本块错误跳转到 M 型基本块，此时也是同层错误跳转。而根据规则 9，无论是上述哪种情况，起点基本块和错误跳转基本块的标签值段均满足 1 位不包含原则，故二者之间的标签值段定会有不同的 1 位，或运算无法掩盖 1 位的不同，故此错误可以被检查出来。

引理 5 所有抵达 O 型基本块的错误跳转，均会被该基本块内的检查语句检测出来。

证明：设 $\forall b_i,b_j \in B$，程序发生错误跳转 $b_j \to b_i$，且程序运行在至 b_j 前均为正确执行。因发生的是错误跳转，则 $b_j \neq pred(b_i)$，根据规则 6-9，$BS(b_j) \neq BS(pred(b_i))$。程序运行在至 b_j 前均为正确执行，故此时全局动态标签与其期望值相等，即 $GS=BS(b_j)$。跳转后，全局动态标签 *GS* 进行更新操作，则 $GS=GS \ xor \ d_i=BS(b_j) \ xor \ BS(b_i) \ xor \ BS(pred(b_i))=BS(b_j) \ xor \ BS(pred(b_i))$，$BS(b_j) \ xor \ BS(pred(b_i))=0 \Rightarrow GS=BS(b_i)$。根据引理 3 与引理 4，无论 *GS* 是哪一位与 $BS(b_i)$ 不相等，均会被基本块内部的检查语句检测出来。

引理 6 所有抵达 M 型基本块的错误跳转，均会被该基本块内的检查语句检测出来。

证明：同样，设 $\forall b_i, b_j \in B$，程序发生错误跳转 $b_j \to b_i$，且程序运行在至 b_j 前均为正确执行。同理，$b_j \notin pred(b_i)$，$BS(b_j) \neq BS(pred(b_i))$，$GS = BS(b_j)$。此后跳转情况与 b_j 的类型与位置有很大的关系。

若 b_j 与 b_i 所属不同层次，则 GS 在错误抵达 b_i 前 $GS_1 = BS_1(b_j) \neq BS_1(b_i)$。根据 M 型基本块标签更新语句，此时 $GS_1 = GS_1 \ or \ d_1(b_i) = BS_1(b_j) \ or \ BS_1(b_{j1}) \ or \ ... \ or \ BS_1(b_{jn})$，其中 $b_{j1}, b_{j2}, ..., b_{jn} \in pred(b_i)$。根据规则 2，$BS_1(b_{j1}) = BS_1(b_{j2}) = ... = BS_1(b_{jn}) = BS_1(b_i)$，故 $GS_1 = BS(b_j) \ or \ BS(b_i)$。而根据规则 7，$BS_1(b_j)$ 与 $BS_1(b_i)$ 互相 1 位不包含，故 $GS_1 = BS_1(b_j) \ or \ BS_1(b_i) \neq BS_1(b_i)$。根据引理 3，层号段的错误会被基本块的检查语句检测出来。

若 b_j 与 b_i 所属相同层次，则错误跳转不会引发层号错误。根据规则 9，GS 在错误抵达 b_i 前 $GS_2 = BS_2(b_j) \neq BS_2(b_i)$。根据 M 型基本块标签更新语句，此时 $GS_2 = GS_2 \ or \ d_2(b_i) = BS_2(b_j) \ or \ BS_2(b_{j1}) \ or \ ... \ or \ BS_2(b_{jn}) = BS_2(b_j) \ or \ BS_2(b_i)$，其中 $b_{j1}, b_{j2}, ..., b_{jn} \in pred(b_i)$。根据规则 9，无论 b_j 是 O 型基本块还是 M 型基本块，$BS_2(b_j)$ 与 $BS_2(b_i)$ 互相 1 位不包含。故 $GS_2 = BS_2(b_j) \ or \ BS_2(b_i) \neq BS_2(b_i)$。根据引理 4，标签值段的错误会被基本块的检查语句检测出来。

定理 2 多层分段标签控制流错误检测技术是完备的。在程序中任何违反正确控制流的基本块间错误跳转均可被检测出来。

证明：程序中的控制流错误会导致两种情况，程序非法跳转至程序基本块内部或跳出程序至非代码部分。跳转至非代码部分代表程序进入了数据部分或者未经初始化的存储区域，根据参考文献[48]，若跳转至这一部分，处理器在尝试执行数据或未初始化的存储区域时会引发指令异常而导致系统报错。如果跳转至程序内部，则将跳转至 O 型基本块或者 M 型基本快。根据引理 5 与引理 6，无论跳转至那种类型的基本块，均会在检查指令中检测出错误。综上所述，多层分段标签控制流错误检测算法是完备的。

4.3 多层分段标签控制流错误检测算法

本文设计了多层分段标签控制流错误检测算法，针对以 C 语言编程的数管软件，通过相应规则进行基本块层次划分，并分配基本块标签层号段及标签值段。插桩标签更新指令和标签检查指令，最终生成具有检错能力的目标程序。具体的步骤如下：

Step1. 将目标程序编译成中间代码，以进行控制流分析。

Step2. 分析程序控制流关系，根据规则 1 重构程序基本块，使整个程序满足 M 不连续原则。

Step3. 根据规则 2 按层划分基本块，根据规则 7、规则 8 分配基本块标签层号。

Step4. 依照基本块层次遍历基本块，同层基本块依照规则 6，规则 8 和 9 分配基本块标签

值段。

Step5. 遍历所有基本块，依照规则 3、规则 4 和规则 5 插桩基本块更新指令。

Step6. 根据规则 10 插桩标签检查指令。

表 4. 1 符号定义

符号	描述
P_{target}	待处理的目标程序
$P_{protected}$	具有检错能力的目标程序
Clang	使用 LLVM 前端编译
IRP_{target}	目标程序中间代码
Cfg	程序控制流程图分析打印工具
B_{target}	目标程序基本块集合
E_{target}	目标程序控制流集合
$E(k)$	集合 E 中的第 k 个元素
e_{ij}	$b_i \rightarrow b_j$的程序控制流
$PG\langle B_{target}, E_{target}\rangle$	目标程序控制流程图
size()	集合内元素的数目
b_i	程序中第i个基本块
b_j	程序中第i个基本块
b_p	空基本块
BS_1	基本块多层分段标签层号段
BS_2	基本块多层分段标签标签值段
BS	基本块多层分段标签
GS	全局动态标签
$Lv()$	基本块所属层次层号的集合
$Lv(b)(i)$	基本块b所属层次层号集合的第i个元素

为了有效的进行控制流错误检测标签的分配和插桩，本文提出了多层分段标签算法，给出了标签分配，计算乃至插桩的全过程。表 4. 1 符号定义为算法中所用到的符号定义。整个算法从对目标程序预处理开始，直至完成对目标程序插桩控制流错误检测指令后结束，如算法 4. 1 所示。

算法 4. 1 多层分段标签算法

输入：	目标程序
输出：	具有检错能力的目标程序

1	**Clang** P_{target} 以获得目标程序中间代码 IRP_{target}
2	**Cfg** IRP_{target} 以获得基本块集合B_{target}，控制流集合E_{target}以及程序控制流程图$PG\langle B_{target}, E_{target}\rangle$
3	根据规则 1：**for** $k = 0 \rightarrow size(E_{target})$ **do** $E_{target}(k) = e_{ij}, b_i, b_j \in B_{target}$
4	**if** $b_i, b_j \in TM$ **Then**
5	$b_p = pred(b_j)$，$b_p = succ(b_i)$ and $b_i \notin pred(b_j)$，$b_j \notin succ(b_i)$
6	**end if**
7	**end for**
8	更新基本块集合B_{target}，控制流集合E_{target}以及程序控制流程图$PG\langle B_{target}, E_{target}\rangle$
9	根据规则 2：对程序基本块按层划分，确定每一个基本块所在层次。
10	根据规则 7：对目标程序所有层次赋予层号。
10	根据规则 8： **for** $i = 0 \rightarrow size(B_{target})$ **do**
11	**if** $size(Lv(b_i)) == 1$ **Then**
12	$BS_1(b_i) = Lv(b_i)(1)$
13	**else if** $size(Lv(b_i)) > 1$ **Then**
14	$BS_1(b_i) = Lv(b_i)(1)$ or $Lv(b_i)(2)$ or ... or $Lv(b_i)(size(Lv(b_i)))$
15	**end if**
16	**end for**
17	根据规则 6，规则 8 与规则 9： **for** $i = 0 \rightarrow size(B_{target})$ **do**
18	**Updated** $BS_2(b_i)$
19	**end for**
20	根据规则 3，规则 4 和规则 5：**for** $i = 0 \rightarrow size(B_{target})$ **do**
21	**if** $b_i \in TM$ **Then**
22	**for** $k = 0 \rightarrow size(pred(d_i))$ **do**
23	$d(b_i) = d(b_i)\ or\ BS(pred(b_i)(k))$
24	插桩指令 $GS = GS\ or\ d(b_i)$
25	**else if** $b_i \in TO$ **Then**
26	$d(b_i) = BS(b_i)\ or\ BS(pred(b_i))$
27	插桩指令 $GS = GS\ xor\ d(b_i)$
28	**end if**
29	**end for**
30	根据规则 10：**for** $i = 0 \rightarrow size(B_{target})$ **do**
31	插桩指令 br $GS \neq BS\ err$
32	**end for**

首先，对目标程序进行预处理(Line 1-2)，使用 LLVM 编译器前端 Clang 和程序控制流程图生成工具 Cfg 分析程序控制流，得出程序控制流程图。其次，使用规则 1-2 对程序内所有基本块按层次划分，并重构程序控制流程(Line 3-9)。接下来依据规则 4-7，为程序内所有基本块分配标签，包括确定标签内部层号段与标签值段(Line 10-19)。然后，根据规则 3 向程序内插桩标

签更新指令(Line 20-29)。最后，再根据标签值插桩检查指令，完成整个程序的控制流检错(Line30-32)。

4.4 多层分段标签控制流错误检测算法实验

4.4.1 实验设计

实验环境如下所示：CPU 为 Intel(R) Core(TM) i7-6700HQ，内存 16G，硬盘空间 500G，操作系统为 Ubuntu18.04。实验首先使用基于 LLVM 4.0 的 Clang 编译实验用测试程序，得到程序中间代码；然后使用 LLVM pass 文件对目标程序中间代码进行基本块按层次划分以及多层分段标签分配。

本文共选择了 3 个测试程序，qsort.c，dijkstra.c，datamanage.c，算法描述如表 4.2 所示。

表 4.2 实验测试程序

名称	描述
qsort	快速排序算法
dijkstra	迪杰斯特拉算法实现
datamanage	数据管理软件

三个测试程序中，前两个出自 mibench 的通信分类与网络分类。datamanage.c 用于计算机的一个数据管理软件。使用本文设计的控制流故障注入工具随机对程序指令注入故障来模拟单粒子翻转对程序运行的影响。为了进一步评估 CFMSL 的性能，实验选用了另外三种方法和 CFMSL 进行横向对比：CFCSS[47]，CEDBR[54]，RCFC[51]。所有方法如表 4.3 所示。

表 4.3 实验所用方法

名称	年份	描述
CFCSS	2002	经典单标签控制流错误检测方法
RCFC	2016	双标签控制流错误检测方法
CEDBR	2018	双标签控制流错误检测方法
CFMSL	2019	多层分段标签方法

本文主要从两个方面来评估 CFMSL 方法的有效性：(1)程序控制流错误的检测能力。（2）时空开销。

4.4.2 控制流故障注入平台构建

与数据流错误有所不同，控制流错误是一种更为注重结果的故障类型。尽管之前提到过，单粒子翻转引发地址错误可以导致控制流错误，但是很多情况下的控制流错误并非单纯是地址

出错引起的。如操作码错误中，有时会改变指令类型，若错误扰乱了程序指令的执行顺序，则可算为控制流错误。本文根据单粒子翻转引发的控制流错误结果分类，分析了两种错误类型来模拟软件控制流错误：控制流错误跳转，控制流错误跳出。

1) 控制流错误跳转，是指程序控制流执行时发生控制流错误，控制流发生错误跳转，但仍然在程序内部。
2) 控制流错误跳出，是指程序控制流执行时发生控制流错误，使控制流错误跳出至程序外。

本文使用的故障注入工具是基于 GDB(GNU Project debugger)[58]二次开发实现的。工具分为两种类型：控制流错误跳转故障注入工具和控制流错误跳出故障注入工具，分别对应上述两种控制流错误类型。

1）控制流跳转故障注入工具

Step1 启动 GDB 并加载要注入控制流跳转错误的程序

Step2 完整运行一次程序，记录所有程序指令跳转的 PC(Program Counter)值

Step3 随机选择一条 PC 值，记录下来，作为控制流错误跳转的起点。

Step4 使用 GDB 重新加载程序。

Step5 运行程序，至 Step3 所选起点指令 PC 值处设置断点。

Step6 在剩余 PC 值中随机选择 1 条，控制程序跳转过去。

Step7 根据程序的输出来统计故障注入的结果。

2）控制流跳出故障注入工具

Step1 启动 GDB 并加载要注入控制流跳出错误的程序

Step2 完整运行一次程序，记录所有程序指令跳转的 PC(Program Counter)值

Step3 随机选择一条 PC 值，记录下来，作为控制流错误跳出的起点。

Step4 使用 GDB 重新加载程序。

Step5 运行程序，至 Step3 所选起点指令 PC 值处设置断点。

Step6 在程序计数器内的地址随机选择一位或者多位进行翻转，同时保证翻转后的地址不在 Step2 记录的程序内部 PC 值中，将程序跳转至翻转后的地址。

Step7 根据程序的输出来统计故障注入的结果。

4.4.3 控制流方法评估实验及结果分析

本文使用错误检错率来评估 CFMSL 方法的错误检测能力。CFMSL 方法可以通过标签检测控制流错误。我们对测试程序注入控制流跳转错误与控制流跳出错误，以此来模拟程序发生控制流错误时的状态。为了验证 CFMSL 方法的性能，我们另外选择三种控制流错误加固方法：

CFCSS[47]，CEDBR[54]，RCFC[51]来进行横向对比。分别使用这三种方法对快速排序算法，最短路径算法和数据管理软件进行加固，为了体现鲁棒性和有效性，我们对每种方法加固后的程序分别注入了 2500 次控制流跳出错误和 2500 次控制流跳转错误，共 60000 次故障来验证 CFMSL 方法的性能。实验结果如图 4. 8 不同加固方法错误检测性能所示。

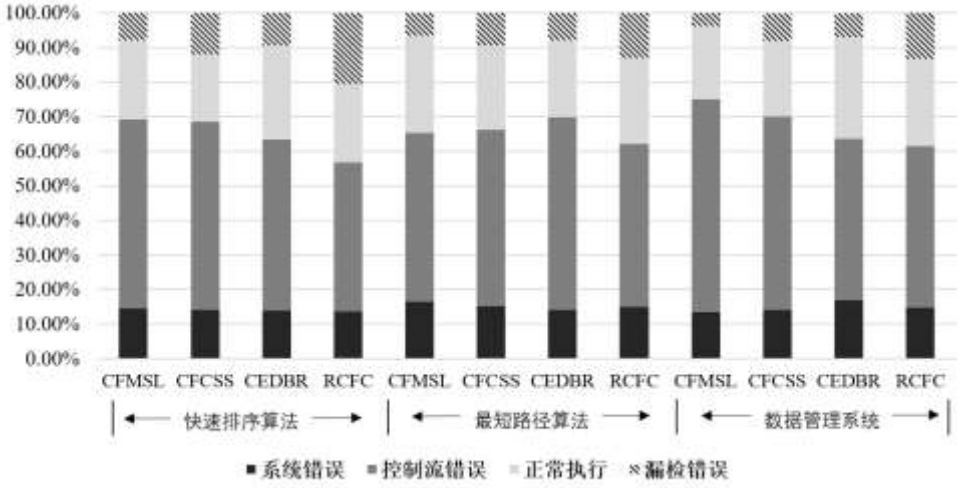


图 4. 8 不同加固方法错误检测性能

表 4. 4 故障注入实验数据

程序	方法	Sys	Find_err	Correct	miss	检错率
快速排序算法	RCFC	682	2161	1135	1022	20.44%
	CFCSS	709	2722	978	591	11.82%
	CEDBR	699	2471	1359	471	9.42%
	CFCMSL	726	2729	1157	388	7.76%
最短路径算法	RCFC	751	2359	1242	648	12.96%
	CFCSS	754	2552	1229	465	9.30%
	CEDBR	709	2779	1116	396	7.92%
	CFCMSL	822	2444	1403	331	6.62%
数据管理系统	RCFC	734	2340	1266	660	13.20%
	CFCSS	707	2796	1096	401	8.02%
	CEDBR	847	2342	1472	339	6.78%
	CFCMSL	675	3071	1064	190	3.80%

图 4. 8 给出采用不同的加固方法下，错误检测性能的对比。漏检率与检错率的统计是综合了控制流错误跳转与控制流错出两种故障注入实验数据的综合结果。其中 CEDBR 和 RCFC 是基于双标签的控制流加固方法，CFMSL 和 CFCSS 则是单标签多规则的加固方法。表 4. 4 给出了具体实验数据，Sys 表示系统错误的次数，Find_err 表示方法发现错误的次数，Correct 表示

程序尽管注入了故障，但仍正确执行的次数。miss 表示未检测出错误的次数。从实验结果可以看出，控制流错误跳出故障类型大部分可以通过系统直接检测出来，小部分对程序没有影响，只有极少的一部分错误会漏检。与之相反的是，控制流错误跳转故障中方法检测出的错误占据了大部分。与图 4. 8 相同，表 4. 4 中最后两栏的错误检测率与错误漏检率是综合了控制流错误跳转与跳出两种故障的实验结果。根据图 4. 8 和表 4. 4，所有的加固方法中 CFMSL 具有最高的错误检测率和最低的错误漏检率。

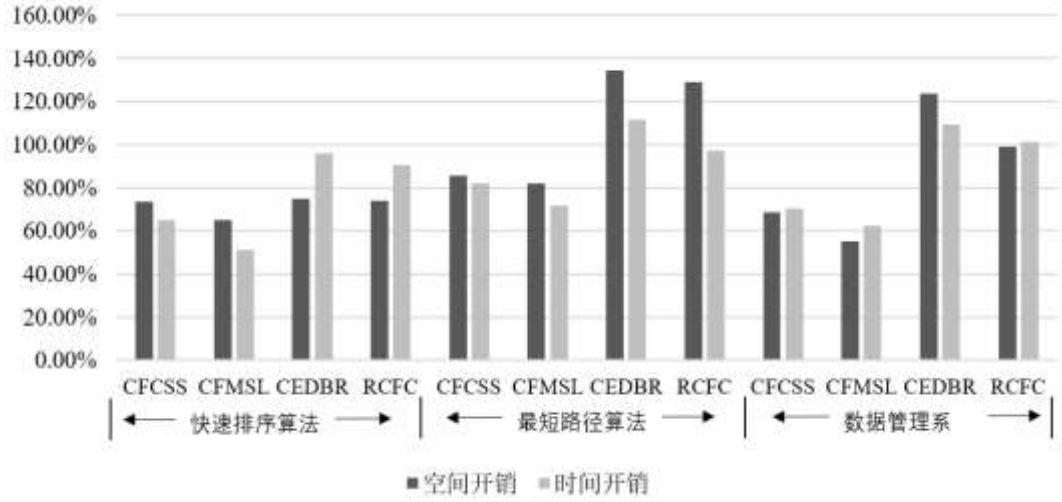


图 4. 9 各方法的时空开销

各方法的时空开销如

图 4. 9 所示，具体实验数据如表 4. 5 所示。从图中可以看出 SEDBR 方法的开销是最高的，RCFC 方法比它稍低一些。CFCSS 和 CFMSL 方法比上两个方法开销要低。

表 4. 5 方法开销数据

程序	方法	空间开销	时间开销
快速排序算法	CFCSS	73.37%	65.18%
	CFMSL	64.86%	51.25%
	CEDBR	74.98%	95.89%
	RCFC	74.12%	90.69%
最短路径算法	CFCSS	85.58%	82.12%
	CFMSL	81.96%	71.70%
	CEDBR	134.19%	111.62%
	RCFC	129.16%	97.13%
数据管理软件	CFCSS	68.83%	70.59%
	CFMSL	55.40%	62.30%

	CEDBR	123.74%	109.44%
	RCFC	99.16%	101.33%

综合图 4. 8 和

图 4. 9 的实验结果，我们发现基于双标签的方法成本一般要比单标签多规则的方法高，而且只要规则设置的合适，单标签的方法错误漏检率也可降低。而在上述四个方法中，单标签的 CFMSL 开销最低，漏检率也是最低的。

RCFC 是一种双标签算法，在 O 型基本块插桩标签 G1，在 M 型基本块插桩 G2。其 G2 标签仅使用 1 和 0 两数来区分标签类型，不具备唯一性。且标签更新函数为“store”赋值语句而非运算，容易将标签错误掩盖，故存在着较高的漏检率。CEDBR 是基于 RCFC 的改进算法，在标签更新上进行了改进。SEDBR 在基本块内部使用加减运算进行 G1 标签更新，尽管开销增加，但显著提高了检错率。另一方面，SEDBR 的 G2 标签仍使用 1 和 0 两数，只能区分基本块类型，不具备唯一性，这导致 SEDBR 仍然存在漏检情况。CFCSS 则是一传统的单标签方法，但是为了解决基本块多前驱矛盾，引入了 D 变量来对标签进行更新，导致开销增大。CFMSL 作为一种单标签方法，却通过分段的形式起到了双标签的作用。将基本块划分层级后，跨层的错误跳转可以通过层号段检测出来，同层的错误跳转可以通过标签值段进行检测。CFMSL 使用 XOR 和 OR 运算的组合，成功在不添加任何指令和标签的情况下解决了基本块多前驱矛盾问题，最后又使用标签更新规则保证了标签更新过程的唯一性。这使得 CFMSL 在具备较高检错率的同时具有较低的开销。

4. 5 本章小结

本章介绍了本文提出的一种多层分段标签控制流错误检测方法 CFMSL。通过分析程序中的控制流信息和基本块结构，按层次重新划分基本块。对重构后的基本块，分配多层分段标签，插桩标签更新指令和标签检查指令，最终可得到控制流检错加固后的程序。本文设计了基本块按层划分规则，基本块标签更新规则，基本块标签检查规则。通过定理证明和逻辑分析证实了方法的正确性和完备性。最终通过控制流故障注入工具对程序进行故障注入实验，最终通过实验结果验证了方法的有效性。

第五章 SEDHS-SEU 软件的实现

本章对提出的 SEDHS-SEU 软件的实现技术进行研究。结合相关项目背景，介绍 SEDHS-SEU 软件所实现的具体功能，进一步给出 SEDHS-SEU 软件开发时所使用的软硬件环境；其次给出为实现 SEDHS-SEU 软件所设计的数据结构。对实现软件功能的核心代码进行描述，最后，对综合错误检测及加固后的目标程序进行故障注入实验，测试了程序的错误检测率以及时间成本和空间成本，以评估软件的性能与开销。

5.1 软件实现概述

5.1.1 软件实现环境

空间环境中的单粒子翻转现象严重威胁了电子设备的正常运行与工作。为了能够使程序在运行时可以具备抗单粒子翻转效应的能力，本文设计了 SEDHS-SEU 软件。依照第二章设计的软件总体框架和总体流程，对软件进行实现。本文实现的功能主要包括：指令传播性特征提取功能，指令固有性特征提取功能，指令冗余加固功能，基本块类型判别功能，基本块按层划分功能，基本块标签分配功能，标签更新指令插桩功能，标签检查指令插桩功能。SEDHS-SEU 软件是基于 LLVM 编译器框架通过实现分析 Pass 和转换 Pass 实现的，因此在实现过程中使用的主要开发语言是 C++，在涉及数据处理、自动化脚本控制的模块中，使用 Python 语言实现。

本文实现的 SEDHS-SEU 软件的软硬件运行环境如下：

1）宿主机硬件环境

操作系统：Ubuntu 16.04.1

CPU：Inter® Core™ i7-6700HQ CPU @ 2.60GHz

内存：8G

硬盘：1TB

2）软件环境

开发环境：Ubuntu 16.04.1

开发平台：LLVM 4.0

开发工具：QT 5.14

开发语言：C++/Python

5.1.2 软件功能实现

本文初步实现了 SEDHS-SEU 软件，实现的主要功能包括：数据流错误检测及加固功能，

控制流错误检测及加固功能，软错误综合检测及加固功能，自动化处理功能和运行评估显控功能。

(1) 数据流错误检测及加固功能

本文通过对程序指令进行特征分析以及单位和多位翻转实验，找出了易受单粒子翻转影响的指令进行针对性加固。提出了 IRVAR(Intermediate Representation VAR)加固方法，设计了冗余加固策略，能够在开销较低的情况下，及时有效的检测出程序中的数据流错误。

(2) 控制流错误检测及加固功能

与数据流错误加固方法类似，基于标签的控制流错误检测技术在插桩标签更新与检查语句时，会带来额外的开销。双标签控制流错误检测技术由于比传统标签技术多了一个标签，开销也相应的升高。为了解决这一问题，本文采取“基本块分层，标签分段”的机制，通过相应的规则实现了一种可以用单标签模拟双标签的多层分段控制流错误检测方法。

(3) 综合检测及加固功能

在加固过程中，数据流错误检测及加固方法会对指令进行冗余但不会影响程序结构，控制流错误检测及加固方法会对基本块层次进行重划分而改变程序结构，所以综合加固过程中先进行数据流错误检测及加固再进行控制流错误检测及加固，使实现软件具备抗单粒子翻转能力。

(4) 自动化处理功能

本文提出的数据流错误检测方法在完成指令识别模型的训练后，可自动将程序中所有易受单粒子翻转影响的指令识别出来，再进行冗余加固处理，满足自动化与批量化要求。控制流错误检测方法尽管相应规则与标签设计较为复杂，但在 LLVM(Low Level Virtual Machine)[66]平台通过 pass 文件便可自动完成程序按层次划分和标签更新检查指令插桩，也满足自动化与批量化要求。

(5) 运行评估显控功能

本文设计的 SEDHS-SEU 软件对目标程序进行错误检测及加固时，每一加固步骤结束后用户均可在显控模块中得到反馈。若出现错误，软件会立刻停止错误检测及加固过程，并对用户示警。已完成错误检测及加固后的程序，可以通过故障注入工具进行试验以评估错误检测及加固性能。

5.2 软件相关数据结构设计

本节主要介绍在实现 SEDHS-SEU 软件主要功能时所用到的数据结构，包括数据结构的名称，功能以及数据结构的类型。

指令传播性特征提取功能是本文数据流加固方法的核心功能，这一部分所用到的数据结构如表 5.1 所示，其中包括：记录目标程序的静态指令编号的数据结构 SInsID、能够按执行顺序

记录目标程序动态指令编号的数据结构 VInsID、代表当前分析的指令所在的基本块的数据结构 BB、代表当前分析的指令所在的函数的数据结构 FUN、代表当前分析指令的数据结构 IN、记录当前指令所含有操作数的数目数据结构 op_num、判断指令内操作数是否是整型的数据结构 is_int、判断指令内操作数是否是浮点型的数据结构 is_float、记录当前指令操作数位数的数据结构 data_width、记录当前指令的屏蔽概率参数的数据结构 p_mask。

表 5. 1 指令传播性特征提取功能数据结构

结构名称	结构类型	结构说明
SInsID	int	目标程序的静态指令编号
VInsID	vector<int>	按执行顺序的动态指令编号
BB	Basicblock*	当前分析的指令所在的基本块
FUN	Function*	当前分析的指令所在的函数
IN	Instruction*	当前分析的指令
op_num	vector<int>	当前指令所含有操作数的数目
is_int	vector<int>	判断指令内操作数是否是整型
is_float	vector<int>	判断指令内操作数是否是浮点型
data_width	vector<int>	指令内操作数总位数
p_mask	vector<double>	当前指令的屏蔽概率参数

指令固有性特征的提取所需要的数据结构如表 5. 2 所示。其中包含的数据结构包括：判断指令是否是比较指令的数据结构 is_cmp、判断指令中是否包含全局变量的数据结构 is_global、判断指令中是否包含函数调用的数据结构 is_func_call、判断指令是否包含堆栈分配的数据结构 is_stack_allocation、记录指令所在基本块包含的指令数目的数据结构 bb_size、记录指令所在基本块的后继基本块数量的数据结构 num_of_suc_bb、记录嵌套此指令的最大循环深度的数据结构 loop_depth、判断指令中是否包含左移运算的数据结构 is_shl、判断指令中是否包含乘法运算的数据结构 is_multiplication、判断指令是否包含与运算的数据结构 is_and、判断指令是否包含与运算的数据结构 is_or、判断指令是否包含从存储器读取数据的操作的数据结构 is_read_with_mem、判断指令是否包含从存储器读取数据的操作的数据结构 is_write_with_mem、判断指令是否包含从存储器读取数据的操作的数据结构 is_read_with_reg。

表 5. 2 指令固有性特征提取功能数据结构

结构名称	结构类型	结构说明
is_cmp	vector<int>	判断指令是否是比较指令
is_global	vector<int>	判断指令中是否包含全局变量
is_func_call	vector<int>	判断指令中是否包含函数调用
is_stack_allocation	vector<int>	判断指令是否包含堆栈分配

bb_size	vector<int>	指令所在基本块包含的指令数目
num_of_suc_bb	vector<int>	指令所在基本块的后继基本块数量
loop_depth	vector<int>	嵌套此指令的最大循环深度
is_shl	vector<int>	判断指令中是否包含左移运算
is_multiplication	vector<int>	判断指令中是否包含乘法运算
is _and	vector<int>	判断指令是否包含与运算
is_or	vector<int>	判断指令是否包含或运算
is_read_with_mem	vector<int>	判断指令是否包含从存储器读取数据的操作
is_write_with_mem	vector<int>	判断指令是否包含从存储器读取数据的操作
is_read_with_reg	vector<int>	判断指令是否包含从存储器读取数据的操作

实现控制流检错功能所需要的相关数据结构如表 5. 3 所示。其中包含的数据结构包括：代表基本块类型的数据结构 BBToType、代表基本块层次的数据结构 BBToLevel、代表基本块序号的数据结构 BBToSig、代表基本块标签的标签值段 BBToGS2、代表基本块标签的数据结构 BBToGS、代表基本块分段差值参数的数据结构 BBToDiff、代表全局变量的数据结构 GS。

表 5. 3 基本块按层划分功能数据结构

结构名称	结构类型	结构说明
BBToType	std::map<BasicBlock*, uint32_t>	基本块类型
BBToLevel	std::map<BasicBlock*, uint32_t>	基本块层次
BBToSig	std::map<BasicBlock*, uint32_t>	基本块序号
BBToGS2	std::map<BasicBlock*, uint32_t>	基本块标签的标签值段
BBToGS	std::map<BasicBlock*, uint32_t>	基本块标签
BBToDiff	std::map<BasicBlock*, uint32_t>	基本块分段差值参数
GS	GlobalVariable*	全局变量

5. 3 软件关键模块的实现

本节主要描述 SEDHS-SEU 软件中最为核心的两个功能模块——数据流加固模块和控制流加固模块的实现以及其相关的核心代码。

5. 3. 1 数据流错误检测及加固模块的实现

SEDHS-SEU 软件中的数据流加固模块主要实现了第三章介绍的 SDCVA-OCSVM 方法。

开发过程中最为核心的是指令多位 SDC 脆弱性特征的提取。本文中提到的指令均为 LLVM 的前端编译器 Clang 编译出的 IR 指令，故在特征提取时也需要使用 LLVM。提取特征时主要分为指令传播性特征提取和指令固有特征提取两个方面，其中传播性特征尤为重要。

```
for(Basicblock::Iterator ins_Iter = B.begin(); ins_Iter != B.end(); ++ins_Iter)
{
  Instruction* IN = &(*ins_Iter);
  std::string Inscode= getInstructioncode(IN);
  if(Inscode == "add") //计算add指令中的屏蔽概率参数
  {
        for(User::op_Iterator  op = IN->op_begin();op != IN->op_end(); op++) //遍历指令内部操作数
        {
            xvalue = getnum(op);
            yvalue = getnum(op++);
            break;
        }
        pmask = (f1(xvalue^y,1,getdatawidth(xvalue))+2*f0(xvalue&yvalue,1,getdatawidth(xvalue)))
              /(getdatawidth(xvalue)+getdatawidth(yvalue));
  }

  if(Inscode == "or")//计算or指令中的屏蔽概率参数
  {
        for(User::op_Iterator  op = IN->op_begin();op != IN->op_end(); op++) //遍历指令内部操作数
        {
            xvalue = getnum(op);
            yvalue = getnum(op++);
            break;
        }
        pmask = (f1(xvalue^y,1,getdatawidth(xvalue))+2*f1(xvalue|yvalue,1,getdatawidth(xvalue)))
              /(getdatawidth(xvalue)+getdatawidth(yvalue));
  }
```

图 5. 1 指令传播特征提取核心代码

屏蔽概率参数和屏蔽特征的计算是传播性特征的核心，这两项参数是探究数据错误在程序中的传播的关键。本文找到了三类可以直接对数据流错误进行屏蔽的计算操作指令：与运算指令，或运算指令和比较运算指令。由于计算屏蔽概率参数时需要分情况讨论，且需要分析的情况较多，故这一部分代码的判断条件也比较多。在实现过程中，需要借助 LLVM 平台的相关函数对指令进行识别。LLVM 本身自带有指令操作数迭代器也可以完成对指令操作，部分核心代码如图 5. 1 所示。

指令传播性特征中还包含数据变量特征，整个数据变量特征是由一个四元组构成，分别为数据变量的生命周期、数据变量的二进制位数，数据变量整型判断特征，数据变量浮点类型判断特征。

```
std::vector<int> op_num;
std::vector<int> is_int;
std::vector<int> is_float;
std::vector<int> data_width;
int datawidth = 0;
for(Basicblock::Iterator ins_Iter = B.begin(); ins_Iter != B.end(); ++ins_Iter)
{
  Instruction*  op_Iterator  = &(*ins_Iter);
  int opnum = 0;
  int isint = 0;
  for(User::op_Iterator  op = IN->op_begin();op != IN->op_end(); op++) //遍历指令内部操作数
  {
     op_num++;
     if(Instruction *U =dyn_cast<Instruction>(op->get()))
        {
          opvalue = getnum(op);
          if(typeId(opvalue) == typeId(0))
                    isint = 1;                          //判断操作数是否是整型数
          if(typeId(opvalue)==typeId(0.0f))
                    isfloat = 1;                        //判断操作数是否是浮点数
          if(typeId(opvalue)==typeId(0.0))
                    isfloat = 1;                        //判断操作数是否为浮点数
          datawidth = datawidth+ getdatawidth(opvalue);  //计算操作数位数
        }
  }
  op_num.pushback(op_num);
  is_int.pushback(isint);
  is_float.pushback(is_float);
  data_width.pushback(datawidth);
  datawidth = 0;
}
```

图 5. 2 数据传播特征提取核心代码

图 5. 2 数据传播特征提取核心代码是数据传播特征提取的核心代码。对于数据变量生命周期的计算，在于对于程序执行时指令动态执行次数的统计。计算指令生命周期需要统计指令动

态执行的次数与位置。通过第四章基本块的定义可知，基本块是一组在程序正确执行时内部无跳转的指令集合。只要执行了基本块的入口指令，程序一定会依照基本块内部顺序一直执行至基本块出口指令为止。故统计指令动态执行次数可转换为统计指令基本块动态执行次数。

5.3.2 控制流错误检测及加固模块的实现

SEDHS-SEU 软件中的控制流加固模块主要实现了第四章介绍的 CFMSL 方法，开发过程主要分为以下几个过程：基本块按层次划分，基本块标签分配，基本块更新与检查语句插桩。

```
//确定目标程序中所有基本块类型
for(Module::iterator I = M.begin(), E = M.end(); I != E; ++I){  //使用LLVM中自带的迭代器遍历整个程序所有函数
        if(!I->isDeclaration()) {
                for(Function::iterator bb_iter = I->begin(); bb_iter != I->end(); ++bb_iter){//使用LLVM中自带的迭代器遍历整个程序所有基本块
                int i=0,j=0;
                BasicBlock* bb = &(*bb_iter);
                for (succ_iterator si = succ_begin(bb), se = succ_end(bb); si != se; ++si) {
                   i++;     //统计当前基本块后继个数
                }
                for (pred_iterator pi = pred_begin(bb), pe = pred_end(bb); pi != pe; ++pi) {
                   j++;     //统计当前基本块前驱个数
                }
                if(i == 1 && j ==1)                  //根据统计出来的节本块前驱和后继的个数确定基本块类型，
                    BBToType[bb] = 1;                //其中数字1，2均为O型基本块，4，8为M型节本块。
                else if(i==2 && j ==1)               //3为整个程序的入口基本块，较为特殊需要单独处理。
                    BBToType[bb] = 2;
                else if (j==0)
                    BBToType[bb] = 3;
                else if (i>1 && j==1)
                    BBToType[bb] = 4;
                else
                    BBToType[bb] = 8;
        }
        }
}
```

图 5.3 基本块分类核心代码

在实现基本块按层次划分过程中，首先需要确定基本块类型和相互之间的前驱后继关系。严格按照 4.1 节中关于前驱，后继的相关定义，统计每一基本块的前驱与后继的个数。依照基本块前驱和后继的个数划分基本块的类型，核心代码如

```
for(Module::iterator I = M.begin(), E = M.end(); I != E; ++I){ //遍历程序中所有函数
        if(!I->isDeclaration()) {
                for(Function::iterator bb_iter = I->begin(); bb_iter != I->end(); ++bb_iter){ //遍历程序中所有基本块
                BasicBlock* bb = &(*bb_iter);
                for (succ_iterator si = succ_begin(bb), se = succ_end(bb); si != se; ++si) { //遍历基本块的所有后继
                   BasicBlock* bbs = (*si);
                   if(std::find(BBToInsert.begin(), BBToInsert.end(), BBToSig[bb]) != BBToInsert.end() &&
                      (BBToType[bb] == 8||BBToType[bb] == 4) && (BBToType[bbs] == 8||BBToType[bbs] == 4)) //看基本块和其后继是否均为M型基本块
                     insertProxyBlock(bb,bbs);      //使用基本块插桩函数插入空基本块
                }
                }
        }
}

BasicBlock* DLV::insertProxyBlock(BasicBlock *source, BasicBlock *target) { //空基本块插桩函数
    BasicBlock *proxyBlock = BasicBlock::Create(
        getGlobalContext(),
        "proxyBlock",
        source->getParent());
    BranchInst::Create(target, proxyBlock);
    PHINode *phiNode = NULL;
    for (BasicBlock::iterator ii = target->begin(); (phiNode = dyn_cast<PHINode>(ii)); ++ii) {
      Value *value = phiNode->getIncomingValue(phiNode->getBasicBlockIndex(source));
      while (true) {
        int idx = phiNode->getBasicBlockIndex(source);
        if (idx == 0) {
          phiNode->removeIncomingValue(idx, false);
        } else {
          break;
        }
      }
      phiNode->addIncoming(value, proxyBlock);
    }
    TerminatorInst *terminator = source->getTerminator();
    for (unsigned int idx = 0; idx < terminator->getNumSuccessors(); ++idx) {
      if (terminator->getSuccessor(idx) == target) {
        terminator->setSuccessor(idx, proxyBlock);
      }
    }
    return proxyBlock;
}
```

图 5.4 空基本块插桩核心代码

图 5. 3 所示。尽管本文将基本块类型分为 O 型基本块和 M 型基本块两类，但仍有一些基本块需要在大分类不变的情况下单独细分。例如入口基本快和出口基本块。

基本块重划分的过程，是为了使整个程序控制流满足 M 不连续原则。这一过程需要遍历程序中所有基本块，并对遍历到的基本块的所有前驱后继关系进行排查。若遍历的基本块为 M 型基本块，其前驱或后继中也包含 M 型基本块，则需要在二者之间插入一个空的基本块，以达到重构目的。部分核心代码如图 5. 4 所示。

```
for(Module::iterator I = ++M.begin(), E = M.end(); I != E; ++I){
        Function * Fun = &(*I);
        if(II->isDeclaration()) {
                for(Function::iterator bb_iter = I->begin(); bb_iter != I->end(); ++bb_iter){
                        BasicBlock* bb = &(*bb_iter);
                        if(BBToType[bb]==3)//入口基本块的标签分配于植桩。
                        {
                            BasicBlock* bb = &(*bb_iter);
                            Instruction* InsertPt = (Instruction*)(bb->getFirstNonPHI());
                            IRBuilder<> Builder(InsertPt);
                            uint32_t currsig = BBToGS[bb];//分配基本块标签值
                            LLVMContext &ctxt = bb->getParent()->getContext();
                            Type* uintTp = Type::getInt32Ty(ctxt);
                            Value* constsig = ConstantInt::get(uintTp,currsig);
                            Builder.SetInsertPoint(bb->getTerminator());//寻找插入位置
                            Builder.CreateStore(constsig, GS);//标签赋值
                        }
                        else if(BBToType[bb]==1||BBToType[bb]==2||BBToType[bb]==4)//O型基本块标签更新与植桩
                        {
                            BasicBlock* bb = &(*bb_iter);
                            Instruction* InsertPt = (Instruction*)(bb->getFirstNonPHI());
                            IRBuilder<> Builder(InsertPt);
                            uint32_t currsig = BBToGS[bb];//分配基本块标签值
                            uint32_t dIffsig = BBToDIff[bb];//分配差值参数
                            LoadInst* G = Builder.CreateLoad(GS);
                            Value* xorInst = Builder.CreateXor(G,dIffsig);//插桩标签更新指令，主要为XOR运算
                            Value* currSigConst = ConstantInt::get(xorInst->getType(),currsig);
                            Value* cmpInst = Builder.CreateICmpNE(xorInst,currSigConst);//插桩检查指令
                            BBToInsertFind_err[bb] = (Instruction*)cmpInst;//在件检查指令中插入报错函数
                            Builder.SetInsertPoint(bb->getTerminator());
                            Builder.CreateStore(xorInst, GS);
                        }
```

图 5. 5 基本块更新与检查指令插桩核心代码

为目标程序划分层次，为每一基本块分配层号，完成基本块按层次划分过程。继续为目标程序的每一个基本块分配标签值号，首先考虑最低层，这层更新规则较为特殊，若层次太多，层号也有可能为 0，此时层号计算不能使用乘法，需要单独进行更新。根据相关规则重新分配基本块标签值号，包括 O 型基本块和 M 型基本块，并将标签值号与层号结合。在标签值分配过程中，还需要确定每一个基本块的分段差值参数，这一部分既是基本块标签分配过程的结束部分，也是标签更新和检查语句插桩的开始部分，最后一部分为标签更新与检查语句插桩过程，依据基本块类型将各部分标签更新与检查指令插桩进程序，部分核心代码如图 5. 5 所示。

5. 4 综合错误检测及加固方法

面向单粒子翻转的综合错误检测方法集成了数据流错误检测及加固方法 SDCVA-OCSVM 和控制流错误检测及加固方法 CFMSL，能够对单粒子翻转引起的随机错误进行检测及加固。由于 SDCVA-OCSVM 方法和 CFMSL 方法均需要在程序中间代码层对程序进行操作，且会在错误检测及加固过程中插桩额外代码，如何保证在对这两种方法进行集成时不让方法失效，是综合错误检测及加固方法需要解决的主要问题。

在 5.3 节中实现的两个关键模块：数据流错误检测及加固模块和控制流错误检测及加固模块，是实现软件综合错误检测及加固的基础。两个关键模块在运行时，均会使用 CLang 对目标程序进行编译，生成中间代码文件，这一部分二者是相同的。之后，二者开始有所区分。数据流错误检测及加固模块会分析目标程序中的指令信息，提取并计算需要的指令特征，建立并训练指令识别器，识别出程序中所有的 I_{ecSDC} 指令。针对程序中的 I_{ecSDC} 指令，选取冗余加固策略插桩代码，最终可生成受到保护的程序。控制流错误检测及加固模块则是分析目标程序中的控制流信息，使用基本块按层次划分规则对目标程序的结构进行重构并划分层次。根据多层分段标签分配规则，为程序中的每一基本块计算多层分表标签的标签值段、层号段、分段差值。根据计算结果，为程序插桩标签更新和检查指令，最终生成受到保护的程序。

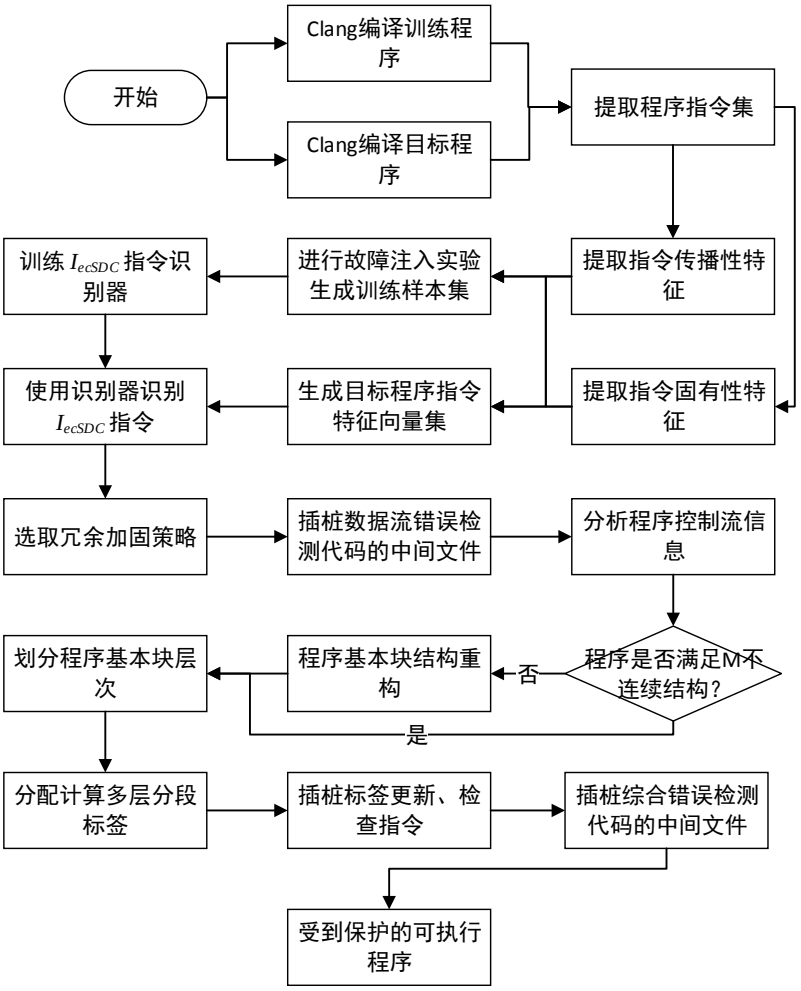


图 5. 6 综合错误检测及加固方法流程

从上述对两个模块的运行过程中可以看出，控制流错误检测及加固模块除了会对目标程序进行代码插桩之外，还会改变整个程序的结构。为了尽量减少程序重构对数据流错误检测及加固模块的影响，综合错误检测及加固方法会先使用 SDCVA-OCSVM 的数据流方法，后使用 CFMSL 的控制流方法进行加固。

综合错误检测及加固方法流程如图 5.6 所示。将目标程序和训练程序编译成中间代码文件，提取指令信息。计算并提取目标程序和训练程序的指令传播性特征和指令固有性特征，对训练程序进行故障注入实验，生成训练样本集。使用训练样本训练 I_{ecSDC} 指令识别器。目标程序提取出的指令特征组成指令特征向量，通过 I_{ecSDC} 指令识别器识别出程序中所有的 I_{ecSDC} 指令。选取冗余加固策略，在程序中插桩冗余指令和检查指令，生成受到数据流错误检测及加固的中间代码文件。分析中间文件的控制流信息，判断程序是否满足 M 不连续结构。若是则直接划分程序基本块层次，否则还需对程序基本块结构进行重构。根据多层分段标签分配规则，计算每一基本块多层分段标签的层号段、标签值段、分段差值。根据计算的结果在中间代码文件中插桩标签更新、检查指令，最终生成受到综合错误检测及加固方法保护的可执行程序。

5.5 测试实验及分析

5.5.1 实验设计

综合错误检测及加固加固是 SEDHS-SEU 软件实现全面对抗单粒子翻转的主要部分。该方法综合了数据流加固方法 SDCVA-OCSVM 和控制流加固方法 CFMSL，对目标程序实现双重加固，无论单粒子翻转发生在程序数据还是地址上，均有所防护。在第三章和第四章中，本文分别对两种方法进行了故障注入实验，验证了方案的有效性，接下来使用同样的思路对 SEDHS-SEU 软件的综合加固方法进行验证。

由于 CFMSL 方法在对目标程序进行加固时，会永久性的改变目标程序的结构，所以综合加固中先使用 SDCVA-OCSVM 方法，再使用 CFMSL 方法。故障模拟实验主要使用 2.2.3 节中设计的基于 GDB 的综合故障注入工具，并按照概率随机模拟单粒子翻转。目标程序可继续使用之前的快速排序算法，最短路径算法，冒泡排序算法，快速傅里叶变换，计算机负载资源计算程序以及数据管理软件。目标程序进行实验时所用的数据集来自于 mibench[83]中的测试数据和 dota-v1.5[84]的卫星数据。

本文测试 SEDHS-SEU 软件的实验环境如下：

1）实验机硬件环境

操作系统：Ubuntu 16.04.1

CPU：Inter® Core™ i7-6700HQ CPU @ 2.60GHz

内存：8G

硬盘：1TB

2）实验机软件环境

实验环境：Ubuntu 16.04.1

3）实验工具及数据集

实验工具：基于 GDB 进行二次开发的故障注入工具

实验数据集：mibench，dota-v1.5.

5.5.2 摸拟单粒子翻转故障注入工具

本文设计的软错误检测及加固软件主要针对空间辐射中的单粒子翻转，若想真正模拟单粒子翻转对设备引起的软错误，也需要将数据流错误和控制流错误二者结合起来。本文设计的综合故障注入详细流程描述如下所示：

step 1. 依据概率随机选择一种单粒子翻转错误类型来模拟位翻故障。

step 2. 启动 GDB 并加载要在其中注入单粒子翻转错误的程序

step 3. 在程序中随机选择一条将注入错误的指令。

step 4. 指令在程序执行中往往会执行多次，为保持随机性，随机选择故障注入时的指令执行次数，这里记为 n。

step 5. 在所选指令处插入一个断点。

step 6. 运行程序，由于插入了断点每次程序会在执行所选指令时停止执行。此时继续执行 n-1 次程序，直至第 n 次执行该指令时根据所选择的错误类型，适当的注入故障。将该指令使用的数据与地址统一起来，根据单粒子翻转类型，随机选择统一数据中的位数来进行翻转。

step 7. 根据程序的输出来统计故障注入的结果。

5.5.3 软件测试实验与分析

在对目标程序进行故障注入后，程序最终运行结果可以分成 4 类：1) correct: 程序正确执行且输出结果无误，此类型表示单粒子翻转并未影响到程序的运行。2) system: 程序运行时出现错误，由系统报错。此类型表示单粒子翻转影响了程序运行，但系统检测出了错误。3) find_error: 程序运行时出现错误，但是被本文的检错方法检测出来，并由报错函数报错。4) miss：程序运行时出现错误，但是并未被检测出来。若程序运行超时或者进入死循环，检错方法和系统均未报错，则也归为此类。

表 5.4 综合加固错误检测实验结果

程序	实验次数	系统错误次数/比率		方法报错次数/比率		程序正确执行次数/比率		错误漏检次数/比率		错误检测次数/比率	
快速排序算法(qsort)	5000	1778	35.56%	2022	40.44%	682	13.64%	518	10.36%	4482	89.64%
迪杰斯特拉算法(dijkstra)	5000	1729	34.58%	2179	43.58%	754	15.08%	338	6.76%	4662	93.24%
冒泡排序算法(BB)	5000	1616	32.32%	2244	44.88%	675	13.50%	465	9.30%	4535	90.70%
快速傅里叶	5000			2342	46.84%			460	9.20%	4540	90.80%

变换(FFT)		1472	29.44%			726	14.52%				
计算机负载资源计算程序(comp)	5000	1296	25.92%	2552	51.04%	822	16.44%	330	6.60%	4670	93.40%
数据管理软件(DataManage)	5000	1203	24.06%	2579	51.58%	947	18.94%	271	5.42%	4729	94.58%

如表 5. 4 综合加固错误检测实验结果和图 5. 7 所示，分别是 SEDHS-SEU 软件对快速排序算法，迪杰斯特拉算法，冒泡排序算法，快速傅里叶变换算法，计算机负载资源计算程序和数据管理软件进行综合加固后，使用故障注入工具对其进行故障注入实验的实验结果。

图 5. 7 综合加固错误检测比率

从实验结果可以看出，综合加固方法对于所有实验程序均有不错的加固效果，平均错误检测率在 90%以上。具体到程序，我们发现综合加固方法对内部含有复杂计算的程序有着更好的加固效果，而对于结构简单的程序效果稍差。同时还可以看出使用综合故障注入工具后，程序正确执行的比率大幅度降低，系统报错次数增多，方法检测次数仍占主体，也从侧面说明了加固的必要性。

表 5. 5 综合加固实验空间开销与时间开销

	原始存储空间	加固存储空间	空间开销比率	加固前 usr 时间	加固前 sys 时间	加固前总时间	加固后 usr 时间	加固后 sys 时间	加固后总时间	时间开销比率
快速排序算法(qsort)	8099	17352	114.25%	3.218	1.656	4.874	7.342	5.206	12.548	157.45%
迪杰斯特拉算法(dijkstra)	20275	52483	158.86%	8.911	3.038	11.949	19.722	11.565	31.287	161.84%
冒泡排序算法(BB)	10042	26115	160.06%	7.026	2.491	9.517	15.253	9.302	24.555	158.01%
快速傅里叶变换(FFT)	27062	75462	178.85%	14.279	7.483	21.762	41.619	24.972	66.591	206.00%
计算机负载资			196.27			31.548				199.51

源计算程序(comp)	36893	109304	%	20.342	11.206		62.558	31.931	94.489	%
数据管理软件(DataManage)	57344	178304	210.94%	27.439	14.527	41.966	85.751	52.141	137.892	228.58%

表 5. 6 综合加固实验内存开销

	加固前 VIRT 内存	加固前 RES 内存	加固前 SHR 内存	加固后 VIRT 内存	加固后 RES 内存	加固后 SHR 内存	内存开销比率
快速排序算法(qsort)	26028	3869	3172	26428	4088	3200	1.96%
迪杰斯特拉算法(dijkstra)	26020	3852	3164	26228	3904	3316	1.25%
冒泡排序算法(BB)	26428	4092	3204	26388	3984	3528	0.52%
快速傅里叶变换(FFT)	26228	3888	3304	26932	4200	3212	2.76%
计算机负载资源计算程序(comp)	26388	3984	3528	26932	4760	3172	2.84%
数据管理软件(DataManage)	43756	5120	3120	44784	5176	3232	2.30%

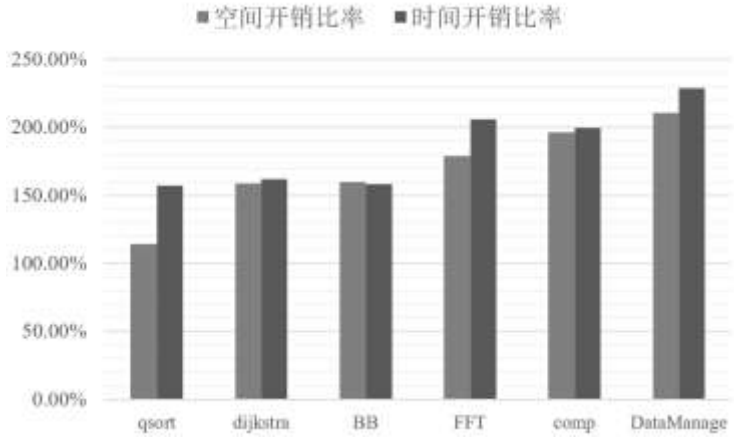


图 5. 8 综合加固实验时空开销

经过加固后程序的时间开销和空间开销如表 5. 5 和图 5. 8 所示。原始存储空间代表目标程序经过 LLVM 的前端编译器 Clang 编译后的中间代码文件所占硬盘空间。加固存储空间代表目标程序经过综合加固后生成的中间代码文件所占硬盘空间。usr 时间和 sys 时间分别为程序用户态使用时间和内核态使用时间，本文使用二者之和来统计时间开销。real 时间在统计过程中有较大的变动，不适合开销计算，故列表中并未提及。综合统计结果可以发现，由于综合加固需要同时使用数据流错误检测及加固方法和控制路错误检测及加固方法，所以其空间开销和时间开销较第三章和第四章的实验结果要大得多。

经过加固后程序的内存开销如表 5. 6 所示。VIRT 内存代表目标程序运行时的进程所需要的虚拟内存的大小。RES 内存表示目标程序运行时，系统为其分配的物理内存的大小，一般统计出进程所占物理内存比率%MEM，就是依照 RES/TOTAL*100%计算的。SHR 实际上是 RES 的一部分，代表程序和别的进程共享的内存。本文综合上述三项内存最终得出了内存开销。从实验结果可以看出，加固后目标程序所增加的内存开销极低，说明加固方法并未增加明显的进程或线程。

5. 6 本章小结

本章介绍了为实现本文所设计的 SEDHS-SEU 软件，所需要的数据结构，以及实现系统主要功能时所用的核心代码。综合了第三章介绍的数据流错误检测方法 SDCVA-OCSVM 和第四章介绍的控制流错误检测方法 CFMSL，设计并实现了综合加固方法。通过综合故障注入工具，对 7 个测试程序进行了 35000 次实验，对综合加固方法的错误检测率和空间、时间以及内存开销进行了评估和计算。

第六章 总结与展望

6.1 论文工作总结

论文对面向辐射环境的软错误的检测技术进行了研究，具有较好的理论意义和应用价值。论文的主要研究工作有：

(1) 系统研究了现有的面向单粒子翻转引起的软错误的检测技术，分析了国内外对数据流错误检测技术和控制流错误检测技术的研究现状，分析了现有的研究成果的优点与不足。

(2) 对单粒子翻转的损伤机理开展了研究，基于 GDB 进行二次开发，搭建了模拟单粒子翻转的故障注入工具及实验验证平台。

(3) 研究了指令的 SDC 多位脆弱性分析方法，提出了一种数据流错误检测及加固方法：SDCVA-OCSVM，可对程序进行数据流错误检测及加固，并通过对比实验验证了该方法的有效性。

(4) 提出了一种多层分段标签控制流错误检测算法 CFMSL，设计了标签结构和更新检查规则，该算法可对程序进行控制流错误检测及加固，并通过实验验证了方法的有效性。

(5) 设计并实现了 SEDHS-SEU 软件，设计了相关数据结构，给出了数据流错误检测和控制流错误检测及加固的核心功能模块的设计与实现技术。并对该软件的功能进行了测试，测试结果表明 SEDHS-SEU 软件具有较好的性能。

6.2 进一步研究工作

本文设计并实现了 SEDHS-SEU 软件，集合了数据流错误检测方法与控制流错误检测方法，能够在达到较高的错误检测率的同时保持较低的时间与空间开销，但是在现有的基础上，仍有研究工作可开展的空间。具体可以从以下几个方面入手：

(1) 本文借助了 LLVM 平台进行研究，使得对程序的分析与加固只停留在中间指令层，未涉及其他层次对软错误传播与加固的影响。接下来可以分别从源代码层、线程层、进程层的角度继续对单粒子翻转造成软错误的损伤机理和检错方法进行讨论。

(2) 本文所提出的数据流检错方法和控制流检错方法均重在检错，如何能在低开销的情况下实现容错，也是下一步需要解决的问题。

(3) 由于条件所限，本文仅进行了模拟仿真工具进行试验，下一步可采用硬件辐射实验验证方法效果。

参考文献

[1] 高博. 从羸弱到崛起，中国科技前进步伐未曾停歇[N]. 科技日报, 2019-10-01(006).

[2] Kelso T S. Satcat boxscore[J]. CelesTrak: SATCAT Boxscore. Accessed April, 2017, 19.

[3] Monreal R M, Alvarez J, Dennis G, et al. Impact of Single Event Effects on Key Electronic Components for COTS-based Satellite Systems[C]. 2019 IEEE Radiation Effects Data Workshop. IEEE, 2019: 1-7.

[4] Mahmood S M, Roeed K. Investigation of Single Event Latch-up effects in the ALICE SAMPA ASIC[J]. PoS, 2019: 023.

[5] Luo X, Wang Y, Hao Y, et al. Research of Single-Event Burnout and Hardening of AlGaN/GaN-Based MISFET[J]. IEEE Transactions on Electron Devices, 2019, 66(2): 1118-1122.

[6] Caron P, Inguimbert C, Artola L, et al. Physical mechanisms inducing electron single-event upset[J]. IEEE Transactions on Nuclear Science, 2018, 65(8): 1759-1767.

[7] Nicolaidis, Michael. Soft errors in modern electronic systems [M]. [New York]: Springer Science \& Business Media, 2010, 41.

[8] Sabet M A, Ghavami B, Raji M. GPU-Accelerated Soft Error Rate Analysis of Large-Scale Integrated Circuits[J]. IEEE Design & Test, 2018, 35(6): 78-85.

[9] Petersen E. Single event effects in aerospace[M]. John Wiley & Sons, 2011.

[10] Australian Transport Safety Bureau.n-flight upset - Airbus A330-303, VH-QPA, 154 km west of Learmonth, WA, 7 October 2008[R].

[11] 林莉君. 到火星“串门”为什么这么难[N]. 科技日报,2011-11-24(005).

[12] Sebestyen G, Fujikawa S, Galassi N, et al. Radiation Hardening, Reliability and Redundancy[M]. Low Earth Orbit Satellite Design. Springer, Cham, 2018: 203-208.

[13] Baumann R. Soft error rate overview and technology trends[J]. Reliability Physics Tutorial Notes, Reliability Fundamentals, 2002.

[14] Houston T W. Single event upset hardened memory cell: U.S. Patent 5,905,290[P]. 1999-5-18.

[15] Abazari M A, Fazeli M, Patooghy A, et al. An efficient technique to tolerate MBU faults in register file of embedded processors[C]. The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS 2012). IEEE, 2012: 115-120.

[16] Anghel L, Alexandrescu D, Nicolaidis M. Evaluation of a soft error tolerance technique based on time and/or space redundancy[C]. Proceedings 13th Symposium on Integrated Circuits and Systems Design (Cat. No. PR00843). IEEE, 2000: 237-242.

[17] Nicolaidis M. Time redundancy based soft-error tolerance to rescue nanometer technologies[C]. Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146). IEEE, 1999: 86-94.

[18] Almukhaizim S, Makris Y. Soft error mitigation through selective addition of functionally redundant wires[J]. IEEE Transactions on Reliability, 2008, 57(1): 23-31.

[19] Xin W. Partitioning triple modular redundancy for single event upset mitigation in FPGA[C]. 2010 International Conference on E-Product E-Service and E-Entertainment. IEEE, 2010: 1-4.

[20] Xie H, Chen L, Evans A, et al. Synthesis of redundant combinatorial logic for selective fault tolerance[C]. 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing. IEEE, 2013: 128-129.

[21] Ban T, Naviner L. Progressive module redundancy for fault-tolerant designs in nanoelectronics[J]. Microelectronics Reliability, 2011, 51(9-11): 1489-1492.

[22] 胡长青. 空间载荷的软件可靠性建模分析与多级冗余防护方法研究[D].西安：西安电子科技大学,2018.

[23] 南京辰. 基于动态可重构的双模冗余系统可靠性设计[D].成都：电子科技大学,2010.

[24] 谢燕,张超洋,周启忠,成奎.基于 FPGA 动态可重构计算机的三模冗余改进法[J].宜宾学院学报:1-7, 2019.

[25] 陈玉坤,冯忠伟,张声艳,刘冬.具备重构能力的三模冗余器载计算机研究[J].计算机测量与控制,2017,25(02):201-203+212.

[26] 刘蕾,常亮,李华旺.星载计算机双冗余 CAN 总线模块设计与实现[J].电子设计工程,2015,23(21):99-102.

[27] Hoque K A, Mohamed O A, Savaria Y. Dependability modeling and optimization of triple modular redundancy partitioning for SRAM-based FPGAs[J]. Reliability Engineering & System Safety, 2019, 182: 107-119.

[28] Bolchini C, Miele A, Santambrogio M D. TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs[C]. 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007). IEEE, 2007: 87-95.

[29] Goloubeva O, Rebaudengo M, Reorda M S, et al. Software-implemented hardware fault tolerance[M]. Springer Science & Business Media, 2006.

[30] Chielle E, Rosa F, Rodrigues G S, et al. Reliability on ARM processors against soft errors through SIHFT techniques[J]. IEEE Transactions on Nuclear Science, 2016, 63(4): 2208-2216.

[31] Reis G A, Chang J, Vachharajani N, Vachharajani N, et al. SWIFT: Software implemented fault tolerance[C]. Proceedings of the international symposium on Code generation and optimization. IEEE Computer Society, 2005: 243-254.

[32] Hari S K S, Adve S V, Naeimi H. Low-cost program-level detectors for reducing silent data corruptions[C]. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, 2012: 1-12.

[33] 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]. ACM SIGPLAN Notices. ACM, 2012, 47(4): 123-134.

[34] Yang X J, Gao L. Error flow model: Modeling and analysis of software propagating hardware faults[J]. Ruan Jian Xue Bao(Journal of Software), 2007, 18(4): 808-820.

[35] Xu X, Li M L. Understanding soft error propagation using efficient vulnerability-driven fault injection[C]. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, 2012: 1-12.

[36] Pattabiraman K, Nakka N, Kalbarczyk Z, et al. SymPLFIED: Symbolic program-level fault injection and error detection framework[C]. 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN). IEEE, 2008: 472-481.

[37] Yang N, Wang Y. Identify Silent Data Corruption Vulnerable Instructions Using SVM[J].

IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, 445 HOES LANE, PISCATAWAY, NJ 08855-4141 USA, 2019.

[38] Liu LP, Cie LL. Identifying SDC-Causing Instructions Based on Random Forests Algorithm[J]. KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS. 2019.

[39] Wang C, Dryden N. Neural network based silent error detector[C]. 2018 IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING (CLUSTER). 2018.

[40] Baffreau S, Bendhia S, Ramdani M, et al. Characterisation of microcontroller susceptibility to radio frequency interference[C]. Proceedings of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems (Cat. No. 02TH8611). IEEE, 2002: I031-I031.

[41] Choi K, Park D, Cho J. SSCFM: Separate Signature-Based Control Flow Error Monitoring for Multi-Threaded and Multi-Core Environments[J]. Electronics, 2019, 8(2): 166.

[42] Zhu D, Aydin H. Reliability effects ofprocess and thread redundancy on chip multiprocessors[C]. IEEE/IFIP International Conference on Dependable Systems and Networks, 2006.

[43] Ohlsson J, Rimen M, Gunneflo U. A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog[C]. [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing. IEEE, 1992: 316-325.

[44] A RHISHEEKESAN, R JEYAPAUL, A SHRIVASTAVA.Control Flow Checking or Not? (for Soft Errors) [J]. ACM Transactions on Embedded Computing Systems (TECS),2019,18:11.

[45] Yau S S, Chen F C. An approach to concurrent control flow checking[J]. IEEE transactions on Software Engineering, 1980 (2): 126-137.

[46] Alkhalifa Z, Nair V S S, Krishnamurthy N, et al. Design and evaluation of system-level checks for on-line control flow error detection[J]. IEEE Transactions on Parallel and Distributed Systems, 1999, 10(6): 627-641.

[47] Oh N, Shirvani P P, McCluskey E J. Control-flow checking by software signatures[J]. IEEE transactions on Reliability, 2002, 51(1): 111-122.

[48] Vemu R, Abraham J. CEDA: Control-flow error detection using assertions[J]. IEEE Transactions on Computers, 2011, 60(9): 1233-1245.

[49] Azambuja J R, Altieri M, Becker J, et al. HETA: Hybrid error-detection technique using assertions[J]. IEEE Transactions on Nuclear Science, 2013, 60(4): 2805-2812.

[50] Chielle E, Rodrigues G S, Kastensmidt F L, et al. S-SETA: Selective software-only error-detection technique using assertions[J]. IEEE transactions on Nuclear Science, 2015, 62(6): 3088-3095.

[51] Zhu Z, Callenes-Sloan J. Towards low overhead control flow checking using regular structured control[C]. 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016: 826-829.

[52] 张鹏,朱利,杜小智,贺朝会,陈皓.基于结构化标签的控制流错误检测算法[J].计算机工程,2016,42(06):37-42.

[53] 李爱国,洪炳熔,王司.一种软件实现的程序控制流错误检测方法[J].宇航学报,2006(06):1424-1430.

[54] 张倩雯. 恶劣环境下嵌入式系统软件错误检测技术的研究[D]. 南京航空航天大学, 2018.

[55] 帕尔哈提江 斯迪克,马建峰,孙聪.一种面向二进制的细粒度控制流完整性方法[J]. 计算机科

学,2019,46(11A): 417-420, 432.

[56] 姬秀娟, 孙晓卉, 许静. 基于复杂控制流的源代码内存泄漏静态检测[J]. 计算机科学, 2019，46(11A): 517-523.

[57] Schölkopf B, Platt J C, Shawe-Taylor J, et al. Estimating the support of a high-dimensional distribution[J]. Neural computation, 2001, 13(7): 1443-1471.

[58] Dolinay J, Dostalek P, Vašek V. Arduino Debugger[J]. IEEE Embedded Systems Letters, 2016, 8(4): 85-88.

[59] 魏志超. InP HBT 器件单粒子效应研究[D]. 西安：西安电子科技大学, 2014.

[60] Pflanz M, Walther K, Galke C, et al. On-line error detection and correction in storage elements with cross-parity check[C]. Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002). IEEE, 2002: 69-73.

[61] Fang Y P, Oates A S. Characterization of single bit and multiple cell soft error events in planar and FinFET SRAMs[J]. IEEE Transactions on Device and Materials Reliability, 2016, 16(2): 132-137.

[62] Goerl R C, Villa P R C, Poehls L B, et al. An efficient EDAC approach for handling multiple bit upsets in memory array[J]. Microelectronics Reliability, 2018, 88: 214-218.

[63] Abazari M A, Fazeli M, Patooghy A, et al. An efficient technique to tolerate MBU faults in register file of embedded processors[C]. The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS 2012). IEEE, 2012: 115-120.

[64] Yoshimoto S, Amashita T, Okumura S, et al. Multiple-bit-upset and single-bit-upset resilient 8T SRAM bitcell layout with divided wordline structure[J]. IEICE transactions on electronics, 2012, 95(10): 1675-1681.

[65] 李晓花, 王雅云, 于锋, 等. 星载计算机 SRAM 抗单粒子多位翻转技术[J]. 计算机工程与设计, 2015, 36(6): 1519-1523.

[66] Shigenobu K, Ootsu K, Ohkawa T, et al. A Translation Method of ARM Machine Code to LLVM-IR for Binary Code Parallelization and Optimization[C]. 2017 Fifth International Symposium on Computing and Networking (CANDAR). IEEE, 2017: 575-579.

[67] Sung H, Chen T, Sura Z, et al. Leveraging OpenMP 4.5 Support in CLANG for Fortran[C]. International Workshop on OpenMP. Springer, Cham, 2017: 33-47.

[68] Maglaras L A, Jiang J, Cruz T. Integrated OCSVM mechanism for intrusion detection in SCADA systems[J]. Electronics Letters, 2014, 50(25): 1935-1936.

[69] Wang Z, Fu Y, Song C, et al. Power System Anomaly Detection Based on OCSVM Optimized by Improved Particle Swarm Optimization[J]. IEEE Access, 2019, 7: 181580-181588.

[70] 黄功,赵永平,谢云龙.基于局部密度的加权一类支持向量机算法及其在涡轴发动机故障检测中的应用[J].计算机应用, 2019:1-10.

[71] Mittal N, Singh U, Sohi B S. Modified grey wolf optimizer for global engineering optimization[J]. Applied Computational Intelligence and Soft Computing, 2016, 2016: 8.

[72] Mirjalili S, Mirjalili S M, Lewis A. Grey wolf optimizer[J]. Advances in engineering software, 2014, 69: 46-61.

[73] Yao Yuan-yuan, Ye chun-ming, Yang feng. Improved Grey Wolf Optimizer for the TFT-LCD Module Assembly Scheduling Problem[J]. Journal of Chinese Computer Systems, 2018(10)：2146-2153.

[74] Ma J, Duan Z, Tang L. A Methodology to Assess Output Vulnerability Factors for Detecting Silent Data Corruption[J]. IEEE Access, 2019, 7: 118135-118145.

[75] Yang N, Wang Y. Identify Silent Data Corruption Vulnerable Instructions Using SVM[J]. IEEE Access, 2019, 7: 40210-40219.

[76] Bem E Z, Petelczyc L. MiniMIPS: a simulation project for the computer architecture laboratory[C]. ACM SIGCSE Bulletin. ACM, 2003, 35(1): 64-68.

[77] Fernanda Kastensmidt, Paolo Rech. FPGAs and Parallel Architectures for Aerospace Applications[M]. Germany: Springer-Verlag, 2016.

[78] Oh N, Shirvani P P, McCluskey E J. Error detection by duplicated instructions in super-scalar processors[J]. IEEE Transactions on Reliability, 2002, 51(1): 63-75.

[79] Huang H, Guyer S Z, Rife J H. Improving run-time bug detection in aviation software using program slicing[C]. 2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER). IEEE, 2017: 1252-1256.

[80] 阳子. “太空抢险”：中国故障卫星抢救纪实[J]. 炎黄春秋, 2019, 9:19-23.

[81] 白照广. 中国现代小卫星发展成就与展望[J]. 航天器工程, 2019(02).

[82] Wei J, Thomas A, Li G, et al. Quantifying the accuracy of high-level fault injection techniques for hardware faults[C]. 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2014: 375-382.

[83] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: A free, commercially representative embedded benchmark suite[C]. Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538). IEEE, 2001: 3-14.

[84] Xia G S, Bai X, Ding J, et al. DOTA: A large-scale dataset for object detection in aerial images[C]. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 3974-3983.

[85] dos Santos, A. F., Tambara, L. A., Benevenuti, F., Tonfat, J., & Kastensmidt, F. L. Applying TMR in hardware accelerators generated by high-level synthesis design flow for mitigating multiple bit upsets in SRAM-based FPGAs[C]. In International Symposium on Applied Reconfigurable Computing. Springer, Cham. 2017: 202-213.

[86] Appathurai, Ahilan, and P. Deepa. Radiation induced multiple bit upset prediction and correction in memories using cost efficient CMC[J]. Informacije MIDEM. 2017, 46, 4: 257-266.

致 谢

时间就像指尖的细沙匆匆流逝，想要抓住些什么，握的越紧却消逝的越快。眨眼间，我的硕士生涯就要结束了。从一个彷徨学子成长为有了明确科研方向和目的的研究者，不过 2 年多的时间。我内心充斥着不舍与感激，感到满足却也有些许的遗憾。

我由衷的感谢我的导师庄毅教授。我钦佩庄老师严谨、认真、刻苦的科研态度，感谢她在科研工作中给与我的悉心指导。庄老师对待学术认真严谨的态度就像一座灯塔，通过言传身教指引着我的科研之路。在我论文写作过程中，从论文总体结构到章节细节，庄老师一遍遍的帮我把关，让我在一遍遍的修改中获益良多。在科研工作中，庄老师总用她深厚的学术造诣帮我改进那些不成熟的想法，和她的交流总会让我的眼界和思维得到进一步的开阔。她的科研热情总能在焦虑疲惫时让我重新振作，继续投入到学术之中。除了在学术上的帮助，庄老师还会教授我们为人处世的道理，关心我们的生活。在人生的道路上，能够遇到这样一位好导师是我的幸运，在此，向庄老师致以诚挚的祝福，愿您身体健康，生活幸福。

也感谢在 317 实验室中所有的师兄弟，师姐妹们。你们的陪伴让我在科研道路上不断前进。你们的帮助让我的科研工作进展顺利。你们敏锐的思维总会给我带来新的启迪，你们既是我的朋友也是我的榜样。感谢晏祖佳师兄、周强师兄、王秋红师姐、旺自旺师兄，你们的帮助和建议让我收获良多。感谢和我同届的张夏豪、韦传讲、黄锐聪、凌超、柳滕、王达卫、黄海涛同学，能够和你们共同学习、交流、合作，是我的荣幸。

感谢我的父母和亲人们对我的支持，尽管我们相隔千里，但是带给我的精神上的鼓舞和物质上的满足是让我能专心科研的不竭动力。

最后，对各位评审专家能够抽出宝贵的时间对本文进行评审表示衷心的感谢，我会认真接受您对本文的批评和改正，对论文的不足和缺点加以完善和改进。

在学期间的研究成果及发表的学术论文

攻读硕士学位期间发表（录用）论文情况

1. **郑伟宁**，庄毅. 一种检测控制流错误的多层分段标签方法[J]. 计算机与现代化. 2020. (校定核心期刊，第一作者，已录用)
2. Gu Jingjing, **Zheng Weining,** et al. Vulnerability Analysis of Instructions for SDC-Causing Error Detection[J]. IEEE Access, 2019, 7: 168885-168898. (SCI，第二作者，已发表)

攻读硕士学位期间参加科研项目情况及获奖情况

1. 2017 年 9 月至今，参与十三五预研“XXX 可信软加固方法研究（编号保密）”国家级项目，参与完成软件开发及平台搭建。
2. 2018 年 9 月至今，参与“软件流监控技术研究与应用（编号 WXWB027020180006）”科研研发项目，主要负责监控技术的开发。
3. 2017 年 9 月至今，参与“高可信计算环境与可信属性验证技术研究（编号保密）” 装备预研领域基金（共用技术）项目，主要参与技术开发及相关文档的撰写工作。
4. 2017 年 9 月至 2018 年，参与“云存储终端应用安全分析技术研究（编号 1015-KFA16327）”科研研发项目，负责技术研究并参与相关文档的撰写工作。
5. 2017 年 9 月至今，参与“高可信嵌入式软件建模与验证方法的研究(编号 1015-GAA1512201)”国家自然科学基金项目，主要负责软件设计与开发并搭建实验平台。
6. 2018 年 12 月获得第十五届全国研究生数学建模竞赛二等奖。