软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(10):2977−2992 [doi: 10.13328/j.cnki.jos.006024]

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

航天嵌入式软件整数溢出的形式化验证方法*

高 猛[1,2], 滕俊元[1,2], 王 政[1,2]

[1](北京控制工程研究所,北京 100190)

[2](北京轩宇信息技术有限公司,北京 100190)

通讯作者: 滕俊元, E-mail: tengjunyuan@sunwiseinfo.com

摘 要: 整数溢出引起的软件系统安全性问题屡见不鲜,已有的模型检测技术由于存在状态空间爆炸、不能有效支持中断驱动型程序检测等缺点而少有工程应用.结合真实案例,对航天嵌入式软件整数溢出问题的分布和特征进行了系统性的分析.在有界模型检测技术的基础上,结合整数溢出特征,提出了基于整数溢出变量依赖的程序模型约简技术;同时,针对中断驱动型程序,结合中断函数特征抽象,提出了基于干扰变量的中断驱动程序顺序化方法.经过基准测试程序和真实航天嵌入式软件实验,结果表明:该方法在保证整数溢出问题检出率的前提下,不仅能够提高分析效率,还使得已有的模型检测技术能够适用于中断驱动型程序整数溢出检测.

关键词: 航天嵌入式软件;整数溢出;有界模型检测;中断驱动型程序;顺序化

中图法分类号: TP311

中文引用格式: 高猛,滕俊元,王政.航天嵌入式软件整数溢出的形式化验证方法.软件学报,2021,32(10):2977−2992. http://www.jos.org.cn/1000-9825/6024.htm

英文引用格式: Gao M, Teng JY, Wang Z. Formal verification method for integer overflow in aerospace embedded software. Ruan Jian Xue Bao/Journal of Software, 2021,32(10):2977−2992 (in Chinese). http://www.jos.org.cn/1000-9825/6024.htm

Formal Verification Method for Integer Overflow in Aerospace Embedded Software

GAO Meng[1,2], TENG Jun-Yuan[1,2], WANG Zheng[1,2]

[1](Beijing Institute of Control Engineering, Beijing 100190, China)

[2](Beijing Sunwise Information Technology Ltd., Beijing 100190, China)

Abstract: The security problems of software systems caused by integer overflow are common, while the existing model checking techniques have few engineering applications due to the shortcomings of state space explosion and failure to support interrupt-driven program detection. This paper systematically analyzes the distribution and characteristics of integer overflow in aerospace embedded software through some real cases. On the basis of bounded model checking, a program model reduction technique based on integer overflow variable dependence is proposed based on the characteristics of integer overflow variables. At the same time, we present a interference variables dependency sequentialization method for interrupt-driven programs based on the characteristic abstraction of interrupt functions. The results of benchmark programs and real aerospace embedded software experiments show that this method can not only improve the analysis efficiency, but also make the existing model checking techniques applicable to integer overflow detection of the interrupt-driven programs under the premise of guaranteeing the detection rate of integer overflow.

Key words: aerospace embedded software; integer overflow; bounded model checking; interrupt-driven program; sequentialization

我国新一代航天器中广泛采用软件密集系统(software-intensive system),软件在保证航天器安全稳定运行、可靠完成任务方面起着至关重要的作用.航天器许多关键任务、复杂功能的完成均依赖于软件,软件规模和复

* 基金项目: 国家自然科学基金(61802017); 装备预研领域基金(61400020407)

Foundation item: National Natural Science Foundation of China (61802017); Equipment Pre-research Field Fund Project (61400020407)

收稿时间: 2019-08-12; 修改时间: 2019-10-10, 2020-01-19; 采用时间: 2020-02-28

杂度呈几何级增长,这种特点和趋势在空间站、载人航天、深空探测等重大航天工程中尤为凸显.同时,航天器属于典型的安全苛求系统,对软件可靠性和安全性有极高的要求,软件一旦发生失效,将会导致任务失败或人员伤亡.

航天器控制功能涉及大量复杂的数学运算,其运算结果的正确性作为航天器可靠性和安全性的关键部分,不仅取决于算法所对应的数学模型的正确描述,也取决于运算过程的正确性.受计算机内存和 CPU 等硬件环境的限制,软件中变量和数据只能使用有限字节,意味着变量和数据的取值范围是有严格要求的,运算结果一旦失真,即使是精心设计的算法,也无法正确实现相关功能.

整数溢出错误是导致软件运算结果出错的重要原因之一,由其引发的事故也屡见不鲜.1996 年,欧洲 Ariane 5 火箭爆炸灾难[1];2004 年,Comair 航空公司航班停飞事故[1]等等.国际权威漏洞披露组织 CVE(common vulnerabilities and exposures)在 2007 年发布的年度报告中,将整数漏洞(integer-based vulnerability)列为威胁软件安全的第二大类漏洞[2],其中,整数溢出错误最为常见.MITRE 也在 2011 年的报告中,将整数溢出错误列为“25 个最危险的软件错误”之一[3].据中国空间技术研究院软件产品保证中心统计,近 10 年,因整数溢出引发的航天器在轨质量问题就有 3 例;在航天器总装测试(A.I.T.)阶段发现的在研质量问题中,约 8%都与整数溢出密切相关,仅 2019 年就发生了 2 起整数溢出导致的质量问题.

航天嵌入式软件通常采用中断驱动机制实现被控对象的控制算法,若直接使用面向顺序程序的整数溢出检测技术来分析航天嵌入式软件,分析结果将不正确;而使用面向多线程程序的整数溢出检测技术来分析航天嵌入式软件,分析结果误报率较高、适用性不好.另外,由于控制算法涉及大量的数值型数据以及数学运算,且中断触发的不确定性导致软件可执行路径分析变得异常复杂,以危险路径分析为主的整数溢出检测方法也存在运行开销大、分析精度不高等问题.因此,目前主流的整数溢出检测方法都无法有效应用.

本文以航天嵌入式软件真实整数溢出案例为基础,对嵌入式软件整数溢出分布及特征进行了系统分析,提出了基于整数溢出变量依赖的程序模型约简技术;针对中断驱动型程序,提出了基于干扰变量的中断驱动程序顺序化方法,使得现有模型检测技术能够有效地对中断驱动型程序的整数溢出问题进行检测;最后,通过实验验证本文方法的有效性.

本文第 1 节基于航天嵌入式软件整数溢出真实案例对整数溢出分布和特征进行分析和总结.第 2 节结合整数溢出特征给出整数溢出有界模型检测方法.第 3 节给出实验评估结果.第 4 节介绍相关研究工作.第 5 节对全文进行总结,并对未来研究方向进行初步探讨.

1 航天嵌入式软件整数溢出特征研究

针对整型溢出案例进行研究,不仅能够补充理论基础,还能为整数溢出检测方法提供有效的指导意见[4].Dietz 等人[5]针对 spec 2000 中的整数溢出问题,从故意使用和未定义操作两个角度进行了分析;Wang 等人[6]从 Linux 内核使用出发,对整数溢出问题进行了系统性研究.

本节选取 2010 年~2019 年航天嵌入式软件评测过程中发现的 75 个整数溢出真实案例作为研究对象(这些整数溢出案例可以通过 https://github.com/JunYuanTeng/Case-Set 下载),分析其分布趋势,并从整数溢出变量、数学运算、循环和递归操作和中断使用方式上进行深入讨论,以更全面地认识航天嵌入式软件整数溢出特征.

1.1 整数溢出分布趋势

整数溢出的表现形式一般可以分为如下 4 类:无符号整数上溢、无符号整数下溢、有符号整数上溢、有符号整数下溢,这 4 类整数溢出错误在本文所研究的航天嵌入式软件 75 个典型实例中均会存在.图 1 描述了 4 类整数溢出错误的百分比堆积柱状图.其中,横坐标记录整数溢出错误发现的年份和总数,纵坐标表示每类整数溢出错误的分布比例,柱状图上的数字表示 4 类整数溢出错误的数量.

总体来说,无符号整数上溢和下溢是整数溢出的主要部分,占比达 81.3%;其次是有符号整数上溢,占比达 16%;有符号整数下溢数目最少.从逐年的数量趋势来看,无符号整数上溢和下溢都占有绝大部分的比例.

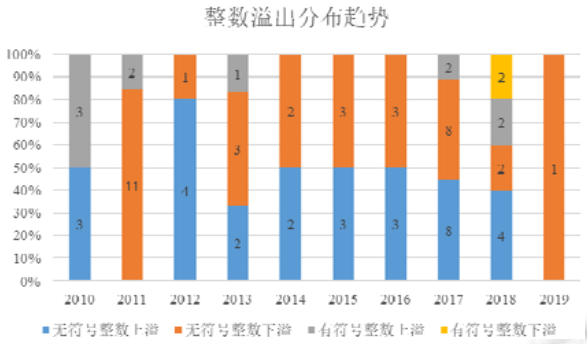


Fig.1 Distributions trend of integer overflow

图 1 4 类整数溢出分布趋势

1.2 整数溢出特征分析

1) 整数溢出变量分析

这里所谓的整数溢出变量是指与整数溢出操作存在数据依赖关系的变量,符合数据流依赖性关系,即具有传递性和无后效性的特点.

a. 传递性.对于整数溢出操作 T,若操作数为变量 a 和 b,则整数溢出操作 T 依赖于变量 a 和 b.若变量 a、b 在整数溢出操作前进行了定义(即变量写操作)或条件判断,记为

$$a=g(m_i)?func(m_j):func(m_k),b=g(n_i)?func(n_j):func(n_k),$$

其中,m_i、m_j、m_k、n_i、n_j、n_k 为变量集合,则整数溢出操作 T 也依赖于集合 m_i、m_j、m_k、n_i、n_j、n_k 中的变量,即整数溢出数据依赖关系具有传递性;

b. 无后效性.假设在程序起始点到整数溢出点 T 的任一条执行路径上的某一整数溢出变量为 m,则变量 m 的值信息仅取决于该路径上最近时刻该变量的定义,而与其他时刻变量 m 的定义无关.

图 2(a)给出了整数溢出变量传递性的示例,其中,程序第 5 行为整数溢出操作,操作数为 m_2 和 m_1.而 m_2 的取值依赖于变量 b 和 c,m_1 的取值依赖于变量 a 和 c,因此,变量 a、b、c、m_1、m_2 称为整数溢出变量.图 2(b)给出了整数溢出变量 a、b、c、m_1、m_2 之间的依赖关系.

Fig.2 Example: Transitivity of integer overflow variables

图 2 示例:整数溢出变量的传递性

图 3 给出了整数溢出变量无后效性的示例,其中,程序第 6 行为整数溢出操作,操作数为 a 和 b.而变量 a 的值仅取决于第 5 行的赋值操作,与第 2 行的赋值操作无关;变量 b 的值仅取决于第 3 行的赋值操作.

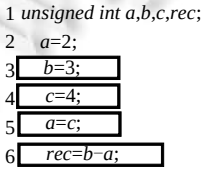


Fig.3 Example: No aftereffect of integer overflow variables

图 3 示例:整数溢出变量的无后效性

2) 数学运算分析

在所研究的 75 个整数溢出实例中,溢出操作均是在对整数进行简单的算术运算过程中引发的.其中,加法运算占 12%,减法运算占 41%,乘法运算占 4%,除法运算占 20%,混合运算占 20%,移位运算占 3%(如图 4 所示).

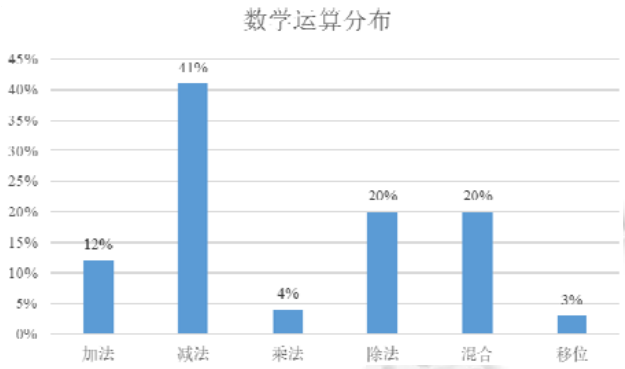


Fig.4 Distribution of mathematical operations

图 4 数学运算分布

混合运算通常表现出以下两种形式.

a. C_1*v_1 op s_1 形式,其中,$C_1 \in N$;op∈{add,sub},v_1 为变量,s_1 为变量或自然数.

一般来说,此类混合运算产生整数溢出错误的原因可以归结为两种情况:(1) 乘法运算发生整数溢出;(2) 混合运算发生整数溢出.

b. (v_1 op v_2)/v_3 形式,其中,op∈{add,sub,mul},v_1、v_2、v_3 为变量.

此类混合运算产生整数溢出错误的原因按操作数类型的不同,又可以有以下区分.

- 操作数为无符号整数.当操作数为无符号整数时,发生整数溢出可以归结为两种情况:(1) 除数为 0 发生整数溢出;(2) 被除数算术运算发生整数溢出;
- 操作数为有符号整数.当操作数为有符号整数时,发生整数溢出可以归结为 3 种情况:(1) 除数为 0 发生整数溢出;(2) 被除数算术运算发生整数溢出;(3) 被除数取值为 *SINTMIN*(有符号最小值),而除数为–1 发生整数溢出.

3) 循环和递归操作分析

当整数溢出变量处于循环体之内时,整数溢出变量的取值会受到循环次数的影响.根据我们对典型整数溢出实例的研究,92%的整数溢出错误可以在循环 20 次内重现,即:只需要限制循环次数最大为 20 次,就可以重现整数溢出错误.另外,8%的整数溢出错误均是由于对无符号整数按周期进行累加操作后发生上溢出所致,重现此类问题需设置的循环次数最大超过 42 亿次.

由于航天嵌入式软件设计时要求谨慎使用递归函数,实例中所有程序均不含有递归操作.

4) 中断使用方式分析

中断驱动是航天嵌入式软件设计常用的模式,实例中所有程序均为中断驱动型程序,其中,受中断使用影响的整数溢出错误占比为 9.3%.另外,根据我们对实例研究后发现:由中断使用引起整数溢出的程序均可抽象为一类常见的中断类型(如图 5 所示),即在相应的中断中对全局变量 V_d 进行赋值;在主程序中对 V_d 进行读访问,并在读访问前禁用中断、读访问后使能中断以避免发生数据访问冲突.另外,值得注意的是:当在中断中对全局变量进行读写访问时,主程序除了对全局变量 V_d 执行初始化操作外,不存在其他对 V_d 的写访问,这样通过中断和主程序之间的相互协作来实现特定的功能.

主程序

```
...
disableISR(i);
...
Read(V_d); //对全局变量V_d读访问
...
enableISR(i);
...
```

中断服务程序

```
voidISR(·){
 ...
 Write(V_d); //对全局变量V_d写访问
       or
 Read and Write(V_d); //对全局变量V_d读写访问
 ...
}
```

Fig.5 A common type of interrupt-driven programs

图 5 一类常见的中断驱动程序

2 基于有界模型检测的整数溢出检测方法

在传统的整数溢出有界模型检测[7,8]下,需要将运行时各变量取值信息以及当前运行位置作为程序模型状态构建状态迁移系统,进而验证系统是否满足整数溢出性质.然而,该方法存在一个显著的问题:模型状态随着程序规模的扩大而成倍增加,从而产生状态空间爆炸,无法有效地针对大规模软件进行检测.另外,航天嵌入式软件的特征之一是使用中断作为软件设计常用的模式,而目前的有界模型检测技术大都无法适用于中断驱动型程序整数溢出检测,因此很难在软件工程实践中直接应用.

本文针对以上问题开展了相关研究工作,首先给出整数溢出有界模型检测执行框架,而框架中的基于整数溢出变量依赖的程序模型约简技术以及基于干扰变量的中断驱动型程序顺序化将在后续的各节中详细描述.

2.1 整数溢出有界模型检测执行框架

有界模型检测理论的基本思想是:用状态迁移系统 M 表征程序的行为,用模态/时序公式 F 描述程序的性质,在给定的 k 界限之内检查系统 M 是否满足模态/时序公式 F:若不满足,则表明在界限 k 之内不存在错误;若满足,则表明存在一条错误执行路径并给出相应的反例.

本文在 Clarke 等人[7,9]有界模型检测研究的基础上,基于整数溢出特征提出了基于整数溢出变量依赖的程序模型约简技术以及基于干扰变量的中断驱动程序顺序化方法,在保证整数溢出分析精度的前提下,尽可能地提高分析效率.

下面我们给出整数溢出有界模型检测执行的基本框架,如图 6 所示.

(1) 对程序进行约简和顺序化操作.框架中提到的程序模型约简和中断驱动程序顺序化方法将分别在后续的第 2.2 节和第 2.3 节中进行详细描述;

(2) 将处理后的程序转化为静态单赋值形式 SSA.将程序按指定的界限 k 进行展开并得到相应的控制流图(control flow graph,简称 CFG),控制流图可以进一步转化为静态单赋值形式.其中,本文中的界限 k 定义为循环执行次数的上界;

(3) 基于断言的属性规约描述整数溢出性质.整数溢出性质属于模型检测特性中的安全性(safty property)范畴,即:在指定的范围内,给定的状态或事件(本文中指的是整数溢出错误)不会发生.基于断言的属性规约能够表征程序运行至断言时给定变量取值集合 S,若存在从程序起始位置到断言位置的一条执行路径,在该执行路径上经过一系列变量定义(即变量写操作),使得最终变量取值不属于集合 S,即发生整数溢出.针对航天嵌入式软件整数溢出操作,可以依据其算术运算过程插入相应的断言,对变量取值集合进行刻画,例如:若判断程序是否满足无符号整数加溢出错误,则在程序的位置 l 处插入断言 $assert(arg_1+arg_2 \geqslant arg_1)$即描述了无符号整数未发生加溢出时变量取值的集合;

(4) 构建逻辑公式,并使用 SMT 求解器求解.将模型各状态之间的转移关系与待验证的整数溢出性质进行合取操作,构成待求解的公式,通过 SMT 求解器求解[10].若有解,则表明存在一条错误执行路径并给出相应的反例;反之,则表明在界限 k 之内不存在整数溢出错误.

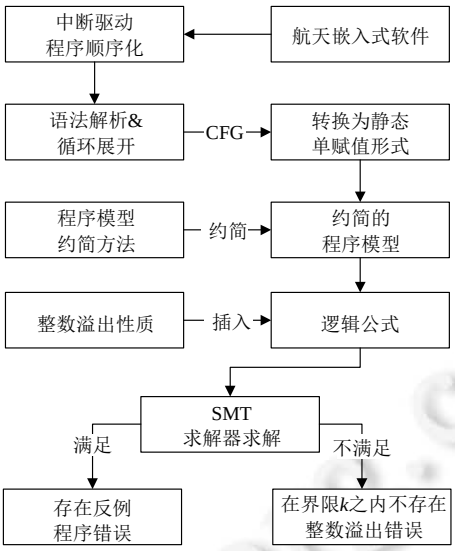


Fig.6 Basic framework of the bounded model checking for integer overflow

图 6 整数溢出有界模型检测基本框架

2.2 基于整数溢出变量依赖的程序模型约简技术

在有界模型检测中,通过对源程序进行语法分析、循环和递归按 k 边界展开、控制流分析,最终产生一种中间表示,称为静态单赋值形式 SSA.在 SSA 形式中,任何变量只在一个语句中被定义,且对变量的任何使用都只有一个到达定值.在构建逻辑公式的过程中,会将 SSA 形式下的所有语句进行合取操作,得到程序的约束条件 C,将待验证的整数溢出断言语句转换为性质集合 P,用于 SMT 求解器求解.随着程序规模的扩大,约束条件 C 将变得异常复杂,使得求解效率大幅降低.

本文根据整数溢出变量的特点对程序约束条件进行约简,使得最终得到的约束条件 C_{simp} 中不包含与整数溢出性质无关的语句.下面给出相应的程序模型约简算法.

算法. 程序模型约简算法.

输入:SSA 中间表示语句集合 S、整数溢出断言语句 *assert*、整数溢出断言语句位置 *locs*;

输出:程序约束条件 C_{simp}.

```
Worklist:={·}
C_simp:=∅
foreach variable v used in assert do
    Worklist:=Worklist∪{v}
line:=locs
repeat
    line:=line−1
    if variable v defined in S[line]∈Worklist then
        delete v from Worklist
        C_simp:=C_simp∪{S[line]}
        if v is defined by a constant then
            continue
        foreach variable u used in S[line] do
            Worklist:=Worklist∪{u}
```

18 **while** *Worklist*≠∅ and *line*>0

该算法的目的是找出所有与指定的断言相关语句组成的约束条件.为了证明本文算法的正确性,首先给出“断言相关语句”定义.

定义 1(断言相关语句). 给定断言 *assert*(*p*),与该断言相关语句 *s* 应满足下列条件之一.

1) *s* 的左值出现在 *p* 中;

2) 存在语句 s_1 位于 *s* 和 *assert*(*p*)之间,s_1 与断言 *assert*(*p*)相关,且 *s* 的左值出现在 s_1 的右值中.

采用反证法给出上述算法的正确性证明,即,上述算法既不会遗漏约束条件,也不会输出多余的约束条件.

1) 假设本文算法遗漏了条件 *S*[*n*],则:

- ∀*assert*(*p*),假设 $S[n]\notin C_{simp}$,且 *S*[*n*]左值 *v* 出现在 *p* 中.由算法定义 6、定义 7⇒$v\in Worklist$,由算法定义 11~定义 13⇒如果 $v\in Worklist$,则 $S[n]\in C_{simp}$,与假设相反;
- ∀*assert*(*p*),假设 $S[n]\notin C_{simp}$,且∃$S[m]\in C_{simp}$ 满足 *S*[*m*]的右值中含有 *S*[*n*]的左值 *v*.由算法定义 16、定义 17⇒$v\in Worklist$,由算法定义 11~定义 13⇒如果 $v\in Worklist$,则 $S[n]\in C_{simp}$,与假设相反.

综上,可得证.

2) 假设本文算法输出了多余的条件 *S*[*n*],则:

∀*assert*(*p*),假设 $S[n]\in C_{simp}$ 与 *assert*(*p*)不相关,C_{simp} 其余条件都与 *assert*(*p*)相关,记 *S*[*n*]的左值为 *v*.

由算法定义 11~定义 13⇒$v\in Worklist$.

由算法定义 6、定义 7 和定义 16、定义 17⇒必存在以下两种情况之一.

a) *v* 定义在 *assert*(*p*)中;

b) ∃$S[m]\in C_{simp}$ 满足 *v* 在 *S*[*m*]的右值中出现,且 *S*[*m*]与 *assert*(*p*)相关.

对于情况 a),由断言相关语句定义 a)⇒*S*[*n*]与 *assert*(*p*)相关;对于情况 b),由断言相关语句的定义 b)⇒*S*[*n*]与 *assert*(*p*)相关.

与假设矛盾,故得证. □

本文算法通过对 SSA 中间表示语句集合进行遍历,根据整数溢出变量传递性和无后效性选取与整数溢出相关的语句加入约束条件 C_{simp} 中.因此,约简后的验证条件为 $C_{simp}\wedge\neg P$,该约简条件将作为模型检测的输入.图 7 给出了一个从 *C* 程序到约简后 SSA 形式的转换过程示例,可以看出:约简后不仅降低了程序约束条件 *C* 的规模,也不会影响程序执行到断言语句处的状态和性质.

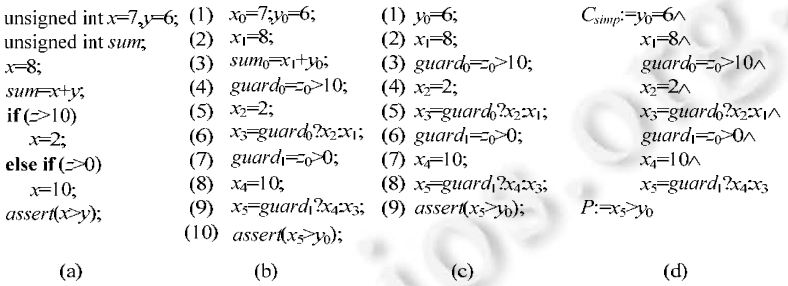
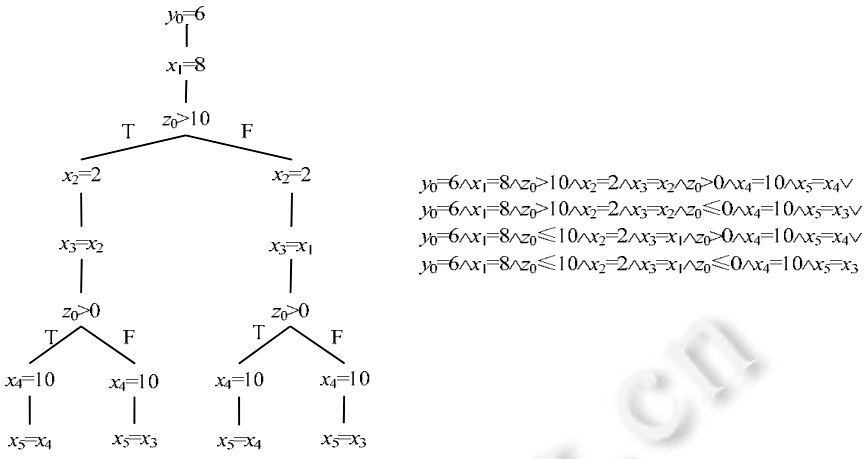


Fig.7 Translation from a *C* program to its reduction SSA form

图 7 *C* 程序到约简后 SSA 形式的转换

本文使用 Z3 作为 SMT 求解器[11].图 7(d)所展示的 C_{simp} 还包括 $x_3=guard_0?x_2:x_1$ 的形式,因此需要对公式加以展开,以便调用 SMT 求解器,如图 8 所示.

图 8(b)中展开得到的析取范式包括具有蕴含关系的单元子句($z_0>10\wedge z_0>0$)及矛盾的子句($z_0>10\wedge z_0\leqslant 0$),本文暂不考虑具有单元子句之间的约简,直接利用 SMT 求解器完成判定.



(a) (b)

Fig.8 Translation from a non-pure linear arithmetic logic to a DNF

图 8 非标准线性算术逻辑展开为析取范式

2.3 基于干扰变量的中断驱动程序顺序化

中断驱动是嵌入式软件设计常用的模式,由主程序和多个中断服务程序共同构成.在航天嵌入式软件中,通常利用中断服务程序完成遥测数据采集、信息通信、遥控指令处理等功能.在中断驱动型程序中:中断机制具有不确定性,即中断服务程序能够发生在主程序执行过程中的任一时刻,当中断发生后,会打断主程序正在执行的操作转而执行中断服务程序;同时,中断机制还具有并发性,即主程序与中断服务程序之间交替执行.

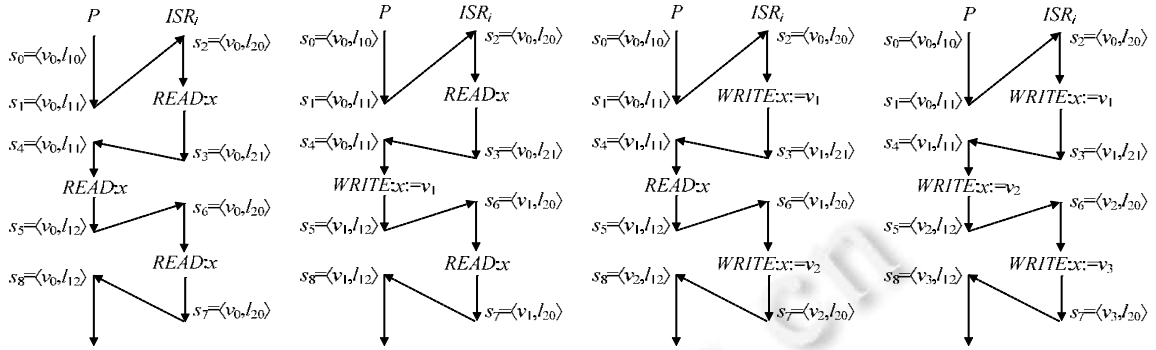
针对第 1.2 节中断使用分析中提到的一类中断驱动程序,本文提出一种基于干扰变量的顺序化方法,该方法的关键是采用抽象解释、摘要机制对中断函数特征表示,并基于干扰变量分析保证顺序化后的程序包含尽可能少的中断函数.

2.3.1 中断驱动程序模型

中断驱动机制引起整数溢出错误主要是由于在中断服务程序中,对变量的操作影响了主程序中该变量的取值.本文将中断驱动程序抽象为 4 类中断驱动程序模型,分别为 R-R 模型、W-R 模型、R-W 模型、W-W 模型(如图 9 所示),并基于中断驱动程序模型分析中断服务程序对变量取值的影响.其中,x 表示程序 P 和中断向量号为 i 的中断 ISR_i 之间的共享数据单元,初值为 v_0;s_i=⟨v_k,l_j⟩表示变量 x 在当前程序下的状态,即变量 x 在位置 l_j 的取值为 v_k;*READ*:x 表示当前程序位置下对变量 x 执行读操作;*WRITE*:x:=v_i 表示当前程序位置下对变量 x 执行写操作,将 v_i 写入 x.

- R-R 模型:表示在 P 中对变量 x 执行读操作,在 ISR_i 中对变量 x 执行读操作.此时,P 和 ISR_i 执行路径中每个状态上变量 x 的取值均为 v_0,说明在 R-R 模型中 ISR_i 的执行并不影响 P 中变量 x 的取值;
- W-R 模型:表示在 P 中对变量 x 执行写操作,在 ISR_i 中对变量 x 执行读操作.此时,P 执行路径上在 *WRITE* 前变量 x 的取值为 v_0,在 *WRITE* 后变量 x 的取值为 v_1.ISR_i 中并不改变变量 x 的取值,说明在 W-R 模型中 ISR_i 的执行并不影响 P 中变量 x 的取值;
- R-W 模型:表示在 P 中对变量 x 执行读操作,在 ISR_i 中对变量 x 执行写操作.此时,由于在 ISR_i 中对变量 x 执行 *WRITE* 操作,使得 P 执行路径中状态 s_1↦s_4、s_5↦s_8 上变量 x 的取值发生改变.说明在 R-W 模型中,ISR_i 的执行影响 P 中变量 x 的取值.进一步来说,对于 R-W 模型中的共享数据单元 x,P 中每次 *READ* 操作都要考虑 ISR_i 中 *WRITE* 操作的影响;
- W-W 模型:表示在 P 中对变量 x 执行写操作,在 ISR_i 中对变量 x 执行写操作.此时,虽然在 ISR_i 中对变量 x 执行 *WRITE* 操作,使得 P 执行路径中状态 s_1↦s_4 上变量 x 的取值发生改变,但是 P 中通过 *WRITE*

操作将状态 s_5 上变量 x 赋值为 v_2,即:对于 W-W 模型中的共享数据单元 x,ISR_i 对变量 x 的作用已被 P 中 *WRITE* 操作清除.



(a) R-R 模型 (b) W-R 模型 (c) R-W 模型 (d) W-W 模型

Fig.9 Model of an interrupt-driven program

图 9 中断驱动程序模型

对于程序 P 和中断 ISR_i,v 是 P 中语句 St 上的一个全局变量.基于对中断驱动程序模型的分析,我们能够得出以下推论.

(1) 若 ISR_i 不存在对 v 的写操作,则语句 St 的执行不受中断影响;

(2) 若 St 中存在对 v 的写操作且 ISR_i 中存在对 v 的写操作,则语句 St 的执行不受中断影响;

(3) 若 St 中存在对 v 的读操作且 ISR_i 中存在对 v 的写操作,则语句 St 的执行受中断影响.

2.3.2 基于函数摘要的中断函数特征分析

首先,基于上一节提出的中断驱动程序模型推论,给出干扰变量的定义.

定义 2(干扰变量). 对于程序 P 和中断向量号为 i 的中断 ISR_i,$Vars(P)$表示程序 P 上的全局变量集合,$Vars(ISR_i)$表示中断 ISR_i 上的全局变量集合,$assess(proc,t)$表示变量 t 在程序 $proc$ 上的访问方式,St 表示程序 P 上的任意一个语句.v 是语句 St 上的一个全局变量,若满足以下两个条件:

(1) 变量 v 属于集合 $Vars(P)$且变量 v 属于集合 $Vars(ISR_i)$;

(2) $assess(P,v)$为读访问方式且 $assess(ISR_i,v)$为写访问方式,

则称变量 v 为中断 ISR_i 对程序 P 上语句 St 的干扰变量,记为 v^{st}.其中,中断 ISR_i 称为干扰中断.

由于在主程序执行过程中,中断 ISR_i 可能在主程序的任何一个语句处发生;而且随着主程序循环的执行,中断发生次数是不确定的;同时,中断 ISR_i 的触发本质上也属于函数调用.因此,我们可以将中断函数 ISR_i 理解为被调用函数,记为函数 ISR_iProc,其形式可以描述为

$$j:=0;\ \text{while}\ (j<nondetnum(\cdot))\{ISR_i(\cdot);j{+}{+}\},$$

其中,函数 $nondetnum(\cdot)$随机返回 1 个正整数.

更一般地,在航天嵌入式软件中断设计中,不同中断完成各自相对独立的功能,其干扰变量集合各不相同,因此我们可以将不同中断上的干扰变量分别单独考虑.在中断函数 ISR_i 中,对于干扰变量 v^{st} 写访问,v^{st} 取值范围不随中断发生次数的增加而改变;对于干扰变量 v^{st} 读写访问,v^{st} 的取值范围随着中断发生次数的增加而扩大,在整数溢出有界模型检测验证时仅考虑 v^{st} 最大取值范围即可.因此,函数 ISR_iProc 可进一步描述为以下形式:

$$\text{while (true)}\{ISR_i(\cdot);\}.$$

对于中断驱动型程序的整数溢出检测,我们所关心的仅是中断函数调用对干扰变量的更新和修改.因此,本文采用函数摘要机制将中断函数特征近似抽象为三元组$\langle VecNum,VarInterfSet,ValRange\rangle$,简称 $summary(ISR_i)$.其中,$VecNum$ 表示中断向量号,$VarInterfSet$ 为中断 ISR_i 对程序 P 的干扰变量集合,$ValRange$ 为干扰变量值区间不

变式.

在进行中断驱动程序顺序化之前,基于抽象解释采用数值区间抽象域对函数 ISR_iProc 进行迭代分析,计算出 ISR_i 上各干扰变量值区间不变式后,生成中断函数摘要.当遇到中断函数触发时,用计算得到的中断函数摘要替代原中断函数.这样不仅能降低分析复杂度,而且能避免对中断函数调用进行多次迭代分析,提高分析性能.

2.3.3 中断驱动程序顺序化方法

中断驱动程序顺序化[12]是指将中断驱动程序转换为可靠的非确定的顺序化程序,即保证转化后的程序行为是原中断驱动程序在实际执行时的一个上近似.

本文基于中断函数摘要机制给出面向整数溢出检测的中断驱动程序顺序化方法,具体步骤如下.

- 步骤 1:对程序 P 以及中断向量号为 i 的中断 $ISR_i(i<N)$进行干扰变量分析,在分析过程中,记录干扰变量信息$\langle v^{st},st,loc^{st},vec\rangle$,简称 $interf^{st}$,其中,v^{st} 为语句 st 上的干扰变量,st 为 P 上使用 v 的语句,loc^{st} 为 P 中的语句 st 位置,vec 为 ISR_i 对应的中断向量号;
- 步骤 2:对中断 ISR_i 分别进行抽象解释迭代分析,求解干扰变量值区间不变式,生成函数摘要.其中,函数摘要中的中断向量号为 i,中断 ISR_i 上的干扰变量集合已在步骤 1 中求得;
- 步骤 3:将相应的干扰变量信息插入到程序的语句块中.本文引入函数 $Trigger(st,interf^{st})$用于将中断函数摘要插入到对应语句之前;同时,引入干扰位置变量 Loc_i 和中断触发标志变量 $Rand_i$.其中,变量 Loc_i 和 $Rand_i$ 的个数等于程序中干扰中断的数量,下标 i 表示干扰中断对应的中断向量号.Loc_i 表示受中断影响的语句位置集合,取值范围为$\{interf^{st}.loc^{st}|interf^{st}.ver==i\}$,用于保证同一中断在主程序一次循环过程中只执行一次;$Rand_i$ 表示中断触发标志集合,取值范围为{0,1},用于表征中断在干扰位置前是否触发.在中断驱动程序顺序化过程中,对 $interf^{st}$ 中的每条语句 st 前显式地加入 $Trigger(st,interf^{st})$函数,假设程序 P 由语句 $st_1,st_2,st_i,\ldots,st_n$ 组成,语句 st_i 为 $interf^{st}$ 中的一条语句,则顺序化后的程序为 $st_1,st_2,st_i',\ldots,st_n$,其中, st_i' 定义如下: $st_i'\Rightarrow Trigger(st,interf^{st});st$.

下面给出 $Trigger$ 函数执行框架,输入为待插入的语句 st 以及经步骤 1 干扰变量分析后得到的干扰变量信息 $interf^{st}$.

```
Trigger(st,interf^st)
1   begin
2   flag_vec:=true
3   for each ⟨v^st,st,loc^st,vec⟩ in interf^st do
4       if Loc_i=interf^st.loc^st and Rand_i=1 and flag_vec=true then
5           flag_vec:=false
6           insert summary(ISR_vec) before st
7       endif
8   endfor
9   end
```

上述执行框架首先对语句 st 中的干扰变量进行遍历,并根据干扰变量信息 $interf^{st}$ 找到该干扰变量对应的干扰中断 ISR_{vec};然后,将干扰中断函数摘要插入到语句 st 之前,完成顺序化操作.在主程序一次循环执行过程中,该执行框架考虑了同一干扰中断最多触发一次以及不同干扰中断可以同时触发的情况,能够确保转化后程序行为的可靠性.

图 10 给出了一个中断驱动程序顺序化示例.图 10(a)中的中断驱动程序包含 1 个主程序和 1 个中断 ISR_0,以及 1 个全局变量 $interfval$,主程序在第 7 行~第 11 行间禁止中断 ISR_0 的触发.若不考虑中断函数触发,则主程序运行至第 9 行时,被减数 $firstval$ 取值为 2,减数 $interfval$ 取值为 1,因此,两个无符号整数相减后未引起整数溢出;若考虑中断触发,即在主程序第 6 行、第 7 行之间发生中断 ISR_0,则减数 $interfval$ 被重新赋值为 3,主程序运行至第 9 行时发生无符号整数减溢出错误.综上,图 10(a)所示中断驱动程序存在整数溢出错误.

主程序在第 8 行赋值语句中对全局变量 *interfval* 进行读访问,中断 ISR_0 对 *interfval* 进行写访问,因此该程序干扰变量信息为⟨*interfval*,*tmpval=interfval*,8,0⟩,中断 ISR_0 函数摘要表示为⟨0,*interfval*,[3,3]⟩.图 10(b)给出了顺序化后的程序,该程序引入了干扰位置变量 loc_0 和中断触发标志变量 $rand_0$,并在赋值语句 *tmpval=interfval* 前插入了中断 ISR_0 函数摘要.经有界模型检测后,给出了发生整数溢出的路径,即:当 loc_0 为 8、$rand_0$ 为 1、*interfval* 为 3 时,会导致整数溢出错误.

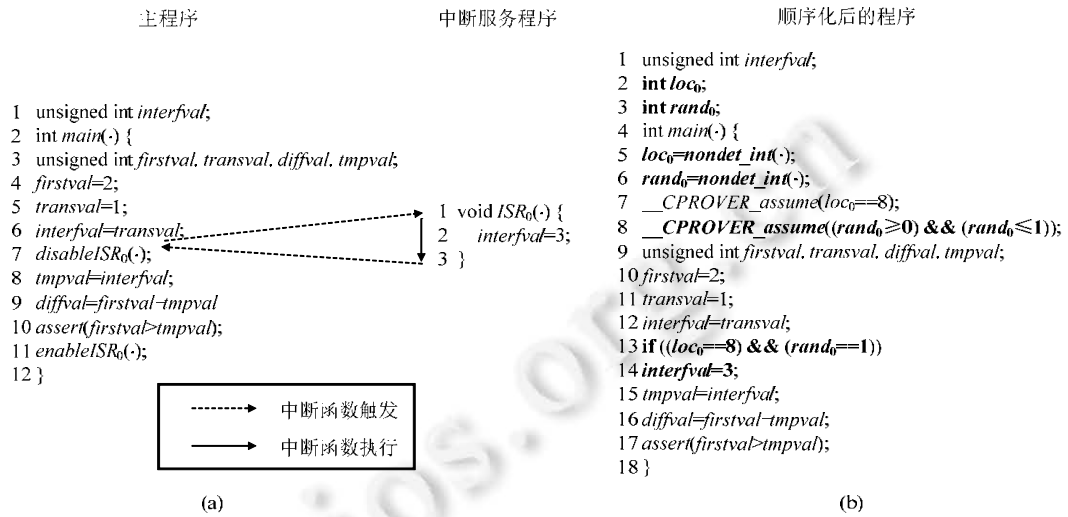


Fig.10 Sequentialization of an interrupt-driven program

图 10 中断驱动程序顺序化示例

3 实验评估

在本节的实验评估中,我们采用“样例+实例”的验证思路对提出的整数溢出检测方法的性能和效果进行评估.在样例验证中,使用基准测试程序来验证基于整数溢出变量依赖的程序模型约简技术的有效性.基准测试程序来源于 SIR(software-artifact infrastructure repository)程序集[13],该程序集是软件研究领域广泛认可的标准验证程序;在实例验证中,以文献[14]介绍的基准测试程序以及从典型整数溢出实例中选取的两个真实航天领域嵌入式软件为对象,重点验证中断驱动程序顺序化方法的有效性.实验的硬件环境均配置为 Windows 10 Intel Core i7- 8550@1.80GHz 8.00GB RAM.

3.1 样例验证

样例验证阶段,选择 SIR 标准验证程序套件中的 4 个软件作为被测程序.被测程序均为顺序化程序,由于其内置故障并不包含整数溢出,因此需要通过变异注入的方式构造整数溢出错误.被测程序的基本信息以及变异注入各类整数溢出错误的数量见表 1.

Table 1 Basic information of the program

表 1 被测程序基本信息

被测程序	无符号整数上溢	无符号整数下溢	有符号整数上溢	有符号整数下溢	问题总数	*LOC*	功能
tcas	1	1	1	1	4	173	飞行器高度隔离
schedule	1	1	2	1	5	412	优先级规划
schedule2	1	1	1	1	4	374	信息度量
printtokens	1	2	2	1	6	726	词法分析

以 tcas 程序为例,表 2 给出了 4 种整数溢出实例,其中,每一个错误包含两行:第 1 行表示变异注入的语句,第 2 行表示整数溢出语句.

Table 2 Fault code in tcas program
表 2 tcas 程序中的故障代码

序号	故障代码	溢出类型	代码行号
1	unsigned int *Positive_RA_Alt_Thresh*[4];	无符号整数上溢	27
	return *Positive_RA_Alt_Thresh*[*Alt_Layer_Value*]*5804010;		58
2	unsigned int *alt_sep*=0;	无符号整数下溢	116
	alt_sep=*alt_sep*−1;		122
3	*Up_Separation*=*nondet_int*(·);	有符号整数上溢	165
	return (*Climb_Inhibit*?*Up_Separation*+*NOZCROSS*:*Up_Separation*)		63
4	Up_Separation = *nondet_int*();	有符号整数下溢	165
	return (*Climb_Inhibit*?*Up_Separation*+*NOZCROSS*:*Up_Separation*−*NOZCROSS*)		63

在样例验证过程中,设置界限 k 为 3,对以上被测程序进行整数溢出检测,并对程序模型约简前后的结果进行对比,实验结果见表 3.

Table 3 Analysis results of sample verification
表 3 样例验证实验结果

被测程序	分析结果							
	布尔变量数		子句数		检测时间(s)		问题命中数	
	约简前	约简后	约简前	约简后	约简前	约简后	约简前	约简后
tcas	3 083	1 803	6 967	5 271	0.06	0.03	4	4
schedule	176 915	120 605	310 125	223 980	2.32	1.86	5	5
schedule2	2 047 412	1 294 648	3 883 800	2 548 033	24.26	14.16	4	4
printtokens	445 260	192 138	3 244 408	2 053 365	8.89	5.64	6	6

从表 3 的实验结果可以看出:在保证问题检出效果的同时,约简后的布尔变量数、子句数以及检测时间相对于约简前大幅度降低.以 tcas 程序为例,其整数溢出检测时间下降 50%.这表明,本文提出的约简技术是行之有效的.

3.2 实例验证

实例验证阶段,由于文献[14]介绍的基准测试程序内置故障不包含整数溢出,因此本文通过变异注入的方式构造整数溢出错误,而选取的真实航天领域嵌入式软件均属于安全关键的中断驱动型程序.被测软件描述信息见表 4.

Table 4 Description of programs used in our experimental evaluation
表 4 被测软件描述信息

被测程序	描述信息	来源
logger	温度记录固件建模程序,包括两个中断处理函数,分别用于测量和通信	基准测试程序
blink	LED 灯控制程序,通过检查计时器值控制 LED 灯闪烁	
brake	由 Matlab/Simulink 仿真模型生成的刹车系统程序,中断处理函数主要用于根据每个轮子的速度计算制动扭矩	
ic2_pca_isa	Linux 内核驱动程序,用于支持 ISA 板功能	
i8xx_tco	Linux 内核驱动程序,用于支持 i8xx 芯片组的 TCO 定时器功能	
wdt_pci	Linux 内核驱动程序,用于支持看门狗功能	
ALTU	用于动量轮脉冲信号采集以及动量轮、磁力矩器、自锁阀等的电源切换	航天嵌入式软件
SIFU	用于完成星上天线伺服控制功能	

本文选取的两个真实航天嵌入式软件分别存在一个受中断使用引发的整数溢出错误,错误信息见表 5.

在实例验证过程中设置界限 k 为 3,使用本文方法对以上被测程序进行整数溢出检测,并分别与两款商业工具 polyspace 2016b 以及 SpecChecker 2.3.2 的结果进行对比,实验结果见表 6.从表 6 的实验结果可以看出:

(1) 本文方法能够有效地检测整数溢出错误.对比其他两款工具,本文方法命中总数为 15 个,并且正确检出了真实航天嵌入式软件所包含的全部整数溢出错误,这表明,本文方法针对中断引起的整数溢出检测是非常有效的;

(2) 与其他两款工具相比,本文方法误报数略高.对所有被测程序检测误报总数进行分析,Polyspace 和 SpecChecker 的误报总数分别为 7 个和 12 个;而本文方法的误报总数为 13 个,误报数略高,误报分析将在后文中给出详细描述;

(3) 与 SpecChecker 相比,本文方法检测时间较长.对所有被测程序检测总时间进行分析,使用 SpecChecker 工具的检测时间为 13.69s,而本文方法的检测时间为 45.69s.这是由于,经顺序化操作后程序规模变大,导致有界模型检测时间增加,并且检测时间增长量与干扰变量的个数以及插入中断的次数密切相关,因此,如何进一步缩短中断驱动程序整数溢出检测时间、提高方法可用性,是未来的研究方向之一.

Table 5 Fault information of aerospace embedded software

表 5 航天嵌入式软件错误信息

被测程序	整数溢出总数	整数溢出类型	出错原因
ALTU	1	无符号整数下溢	中断中将共享变量清零,退出中断后在主程序中对该变量进行减 1 操作,导致整数溢出
SIFU	1	无符号整数下溢	中断中对共享变量 *count* 进行累加操作, 主程序中循环执行如下语句: *mwcur=count*; *mwsend=mwcur−mwlast*; *mwlast=mwcur*; 若共享变量 *count* 累加后溢出(属于正常设计), 则主程序中会发生小数减大数的整数溢出错误

Table 6 Analysis results of instance verification

表 6 实例验证实验结果

软件		logger	blink	brake	ic2_pca_isa	i8xx_tco	wdt_pci	ALTU	SIFU	合计
规模/LOC		161	164	819	321	757	1 339	904	1 985	6 450
中断数		2	3	2	4	3	8	4	3	29
整数溢出错误		3	2	3	2	2	4	1	1	18
Polyspace	命中数	1	1	1	0	1	2	0	0	6
	误报数	0	0	0	0	0	0	1	6	**7**
	检出时间(s)	4.00	6.00	10.00	6.00	9.00	13.00	11.00	18.00	77.00
SpecChecker	命中数	2	1	1	0	2	3	0	0	9
	误报数	0	0	0	0	0	0	4	8	12
	检出时间(s)	0.34	0.33	0.62	0.45	0.78	4.15	0.54	6.48	**13.69**
本文方法	命中数	3	2	2	1	2	3	1	1	**15**
	误报数	0	0	0	0	0	0	2	11	13
	检出时间(s)	0.97	0.73	2.33	1.17	0.80	5.82	0.37	33.50	45.69

本文对所有误报进行了分析,发现本文方法误报的主要原因是:某些干扰变量取值范围仅在软件需求中进行约束,而在程序中缺少相关条件判定,经中断函数特征抽象后,该干扰变量取值区间大于需求中规定的取值范围,经有界模型检测后引起误报.图 11 给出了此类误报的典型样例,样例中,数组 *uartSend* 类型为无符号字符型.本文方法报告 S_1 处发生无符号整数上溢,即:在循环累加过程中,累加值超出了无符号字符型整数取值范围.在实际程序运行中,从地址 XBYTE[0x002D],XBYTE[0x002E]取值内容受需求规范约束,使得 S_2 和 S_3 处的变量 *uartSend*[2]和 *uartSend*[3]被赋值后,参与 S_1 处的累加操作时不会产生整数溢出,因此该整数溢出为误报.

本文方法通过对中断 *ex*0*_ISR* 中的干扰变量 *uartSend*[2]和 *uartSend*[3]值区间不变式进行迭代分析,计算出上述变量取值范围为[0,0xFF],经顺序化后进行有界模型检测,报告 S_1 处发生整数溢出.虽然仅从代码分析角度来看确实存在发生整数溢出的风险,但是从软件功能分析角度来看,该处不可能发生整数溢出.因此,下一步我们将探索更精确的中断特征抽象技术来消除该类误报.

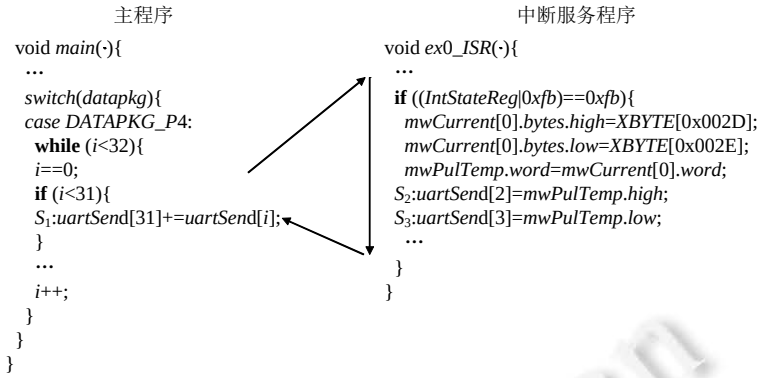


Fig.11 A typical example of false alarms

图 11 误报典型样例

4 相关工作

针对源码级整数溢出,已有的检测方法主要包括变量区间运算、污点分析和类型规约、动态分析.

(1) 变量区间运算.Sarkar 等人[15]提出了流不敏感的整数溢出分析方法,通过条件判断语句约束变量取值范围,在遍历抽象语法树的过程中更新子分支下的变量约束条件,通过获得的变量数值范围信息来判断是否发生整数溢出.Pereira 等人[16]提出了分阶段提取变量之间关系操作的约束,在拓扑顺序中分析强连通分支,以加快变量约束求解的速度;

(2) 污点分析和类型规约.以 IntPatch、Kint、DRIVER 等工具[6,17,18]为代表的污点分析及类型规约技术,通过提取程序中的危险路径,并利用代码插装或约束求解等方法来判定该路径上是否发生整数溢出.其中,IntPatch 和 Kint 主要检测流入内存分配操作的整数溢出错误;DRIVER 尝试提取程序中所有危险路径,并采用代码插装的方式来动态判定是否会触发整数溢出;

(3) 动态分析.以 ARCHERR 和 RICH 等工具[19,20]为代表的动态分析技术,在程序执行过程中对整数操作插装变量取值范围的判定来检测整数溢出.其中,ARCHERR 考虑了运行时环境信息对整数算术运算的影响;RICH 为整数操作语义定义形式化的安全规则,根据类型规则约束插装动态验证代码.

由于航天嵌入式软件控制算法涉及大量的数值型数据以及数学运算,而中断触发的不确定性导致软件可执行路径分析变得异常复杂,因此上述这些技术具有明显的应用局限性,如存在运行开销大、误报率高等问题.本文与这些研究的不同在于:根据整数溢出变量特征和中断驱动程序模型提出了针对性的检测方法,能够更有效地捕获整数溢出错误.

在中断驱动型程序分析与验证基础研究方面,Kroening 等人[21]将中断驱动程序转换为对内存的原子读写访问事件,并将这些访问事件的所有并发交叠状态直接编码为 SAT/SMT 公式,然后使用有界模型检测工具对生成的公式进行验证.Schlich 等人[22]在模型检测时采用偏序约简技术,降低了中断驱动程序验证所要遍历的状态空间规模.Kidd 等人[23]通过线程调度策略将多线程程序转换为顺序化程序,该方法只适用于中断优先级固定的情况.Wu 等人[12]基于数据流依赖对中断驱动程序顺序化,并在抽象解释框架的基础上设计标志量抽象域和语法等价抽象域来提高顺序化后中断驱动型程序的分析精度.

文献[24]的研究思路与本文较为接近,主要提出将中断并发程序自动序列化为非确定性的顺序程序,然后将竞争发生的路径条件转换为 SMT 公式,并通过有界模型检测进行验证.但当中断程序的数目和规模较大时,会产生状态空间爆炸,造成分析无法完成.与该项研究相比,本文作了如下关键技术改进:提出了一种基于干扰变量的中断驱动程序顺序化方法,引入抽象解释、摘要机制对中断函数特征表示,在保证中断驱动型程序正确性的同时也有效降低了顺序化后程序规模,并使得目前的模型检测技术能够适用于中断驱动型程序整数溢出检测.

5 总 结

在航天嵌入式软件整数溢出检测方面,模型检测技术少有工程应用.一方面,随着程序规模的扩大,存在着状态空间爆炸问题;另一方面,已有的模型检测技术不能有效支持中断驱动型程序的整数溢出检测.鉴于此,本文以航天嵌入式软件整数溢出真实案例为基础,系统地分析了整数溢出分布趋势以及溢出特征.在有界模型检测技术的基础上,结合整数溢出特征,提出了基于整数溢出变量依赖的程序模型约简技术,有效地降低了程序约束条件C的规模.同时,面向中断驱动型程序,提出了基于干扰变量的中断驱动程序顺序化方法,该方法采用抽象解释、摘要机制对中断函数特征表示,并基于干扰变量分析保证顺序化后的程序包含尽可能少的中断函数.通过实验,结果表明:本文所提出的方法不仅能够有效地提高分析效率,还使得已有的模型检测技术能够适用于中断驱动型程序整数溢出检测.由实验分析可知,整数溢出有界模型检测的精度和效率与中断特征抽象程度、干扰变量个数以及插入中断次数密切相关.因此,结合更精确的中断特征抽象以及程序顺序化优化技术来缩短检测时间、提升分析精度,将是本文下一步研究的重点.另外,针对带中断的多线程程序整数溢出模型检测,也将是未来的研究方向.

References:

[1] Sercord RC. Secure Coding in C and C++: Of Strings and Integers. Addison Wesley, 2006. [doi: 10.1109/MSP.2006.22]

[2] Christey S, Martin RA. Vulnerability type distributions in CVE. 2007. https://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

[3] Christey S, Martin RA, Brown M, Paller A, Kirby D. 2011 CWE/SANS top 25 most dangerous software errors. Technical Report, MITRE Corporation, 2011. http://cwe.mitre.org/top25

[4] Sun H, Zeng QK. Research on integer-based vulnerabilities: Security model, detecting methods and real-world cases. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):413−426 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4793.htm [doi: 10.13328/j.cnki.jos.004793]

[5] Dietz W, Li P, Regehr J, Adve VS. Understanding integer overflow in C/C++. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Los Alamitos: IEEE, 2012. 760−770. [doi: 10.1109/ICSE.2012.6227142]

[6] Wang X, Chen HG, Jia ZH, Zeldovich N, Kaashoek MF. Improving integer security for systems with KINT. In: Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2012. 163−177.

[7] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Jensen K, Podelski A, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Berlin: Springer-Verlag, 2004. 168−176. [doi: 10.1007/978-3-540-24730-2_15]

[8] Biere A, Cimatti A, Clarke E, Zhu YS. Symbolic model checking without BDDs. In: Cleaveland WR, ed. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99). Berlin: Springer-Verlag, 1999. 193−207. [doi: 10.1007/3-540-49059-0_14]

[9] Clarke E, Kroening D, Yorav K. Behavioral consistency of C and verilog programs using bounded model checking. In: Getreu I, Fix L, Lavagno L, eds. Proc. of the 40th Annual Design Automation Conf. (DAC 2003). New York: ACM, 2003. 368−371. [doi: 10.1145/775832.775928]

[10] Cordeiro LC, Fischer B, Marques-Silva J. SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans. on Software Engineering, 2012,38(4):957−974. [doi: 10.1109/TSE.2011.59]

[11] de Moura L, Bjørner N. Z3: An efficient SMT solver. In: Ramakrishnan CR, Rehof J, eds. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Berlin: Springer-Verlag, 2008. 337−340. [doi: 10.1007/978-3-540-78800-3_24]

[12] Wu XG, Chen L, Mine L, Dong W, Wang J. Static analysis of runtime errors in interrupt-driven programs via sequentialization. ACM Trans. on Embedded Computing Systems (TECS), 2016,15(4):70. [doi: 10.1145/2914789]

[13] Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 2005,10(4):405−435. [doi: 10.1007/s10664-005-3861-2]

[14] Sung CH, Kusano M, Wang C. Modular verification of interrupt-driven software. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Los Alamitos: IEEE, 2017. 206−216. [doi: 10.1109/ase.2017.8115634]

[15] Sarkar D, Jagannathan M, Thiagarajan J, Venkatapathy R. Flow-insensitive static analysis for detecting integer anomalies in programs. In: Hasselbring W, ed. Proc. of the 25th Conf. on IASTED Int'l Multi-Conf.: Software Engineering. ACTA, 2007. 334−340. [doi: 10.5555/1332044.1332098]

[16] Pereira FMQ, Rodrigues RE, Campos VHS. A fast and low-overhead technique to secure programs against integer overflows. In: Proc. of the 2013 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). Los Alamitos: IEEE, 2013. 1−11. [doi: 10.1109/CGO.2013.6494996]

[17] Zhang C, Wang TL, Wei T, Chen Y, Zou W. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Gritzalis D, Preneel B, Theoharidou M, eds. Proc. of the 15th European Conf. on Research in Computer Security (ESORICS). Berlin: Springer-Verlag, 2010. 71−86. [doi: 10.1007/978-3-642-15497-3_5]

[18] Sun H, Li HP, Zeng QK. Statically detect and run-time check integer-based vulnerabilities with information flow. Ruan Jian Xue Bao/Journal of Software, 2013,24(12):2767−2781 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4385.htm [doi: 10.3724/SP.J.1001.2013.04385]

[19] Chinchani R, Iyer A, Jayaraman B, Upadhyaya SJ. ARCHERR: Runtime environment driven program safety. In: Samarati P, Ryan P, Gollmann D, Molva R, eds. Proc. of the 9th European Symp. on Research in Computer Security (ESORICS 2004). Berlin: Springer-Verlag, 2004. 385−406. [doi: 10.1007/978-3-540-30108-0_24]

[20] Brumley D, Song DX, Chiueh T, Johnson R, Lin H. RICH: Automatically protecting against integer-based vulnerabilities. In: Proc. of the 14th Annual Network and Distributed System Security Symp (NDSS 2007). 2007.

[21] Kroening D, Liang L, Melham T, Schrammel P, Tautschnig M. Effective verification of low-level software with nested interrupts. In: Nebel W, Atienza D, eds. Proc. of the Design, Automation & Test in Europe Conf. & Exhibition (DATE 2015). CA: EDA Consortium, 2015. 229−234. [doi: 10.7873/DATE.2015.0360]

[22] Schlich B, Noll T, Brauer J, Brutschy L. Reduction of interrupt handler executions for model checking embedded software. In: Namjoshi K, Zeller A, Ziv A, eds. Proc. of the 5th Int'l Haifa Verification Conf. on Hardware and Software: Verification and Testing (HVC 2009). Berlin: Springer-Verlag, 2009. 5−20. [doi: 10.1007/978-3-642-19237-1_5]

[23] Kidd N, Jagannathan S, Vitek J. One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In: van de Pol J, Weber M, eds. Proc. of the 17th Int'l SPIN Conf. on Model Checking Software (SPIN 2010). Berlin: Springer-Verlag, 2010. 245−261. [doi: 10.1007/978-3-642-16164-3_18]

[24] Wu XG, Wen YJ, Chen LQ, Dong W, Wang J. Data race detection for interrupt-driven programs via bounded model checking. In: Proc. of the 7th Int'l Conf. on Software Security and Reliability Companion (SERE 2013). Los Alamitos: IEEE, 2013. 204−210. [doi: 10.1109/SERE-C.2013.33]

附中文参考文献:

[4] 孙浩,曾庆凯.整数漏洞研究:安全模型、检测方法和实例.软件学报,2015,26(2):413−426. http://www.jos.org.cn/1000-9825/4793.htm [doi:10.13328/j.cnki.jos.004793]

[18] 孙浩,李会朋,曾庆凯.基于信息流的整数漏洞插装和验证.软件学报,2013,24(12):2767−2781. http://www.jos.org.cn/1000-9825/4385.htm [doi: 10.3724/SP.J.1001.2013.04385]

高猛(1982－),男,高级工程师,CCF 专业会员,主要研究领域为软件可靠性,嵌入式软件测试,模型检测.

王政(1986－),男,博士,高级工程师,主要研究领域为软件形式化建模,分析与验证,基于模型的设计.

滕俊元(1985－),男,高级工程师,主要研究领域为软件形式化验证,嵌入式软件测试.