ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.4, April 2007, pp.1056−1067
DOI: 10.1360/jos181056

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

操作系统内核程序函数执行上下文的自动检验*

汪 黎+, 杨学军, 王 戟, 罗 宇

(国防科学技术大学 计算机学院,湖南 长沙 410073)

Automatedly Checking Function Execution Context of Kernel Programs in Operation Systems

WANG Li+, YANG Xue-Jun, WANG Ji, LUO Yu

(School of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: Phn: +86-731-4574618, E-mail: dragonfly@linux-vs.org

Wang L, Yang XJ, Wang J, Luo Y. Automatedly checking function execution context of kernel programs in operation systems. *Journal of Software*, 2007,18(4):1056−1067. http://www.jos.org.cn/1000-9825/18/1056.htm

Abstract: Function execution context correctness is one of the most easily violated critical properties by OS (operation system) kernel programs while it is non-trivial to be checked out. The existing solutions suffer some difficulty and limitation. This paper presents a framework PRPF to check the correctness of function execution context, as well as the modeling process and algorithms in detail. The PRPF has the advantages, such as direct checking source code, no need writing formal specifications, low time and space costs, and perfect scalability, etc., over the existing techniques. The technique has been applied in checking the Linux kernel source 2.4.20. The experimental results show that PRPF can check the correctness of function execution context as expected by automatically exploring all paths in the sources. As a result, 23 errors and 5 false positives are found in the 'drivers/net' source directory. The technique is very helpful in improving the quality of OS kernel codes.

Key words: OS (operation system) kernel programs; kernel programming interfaces; program verification; program correctness; verification of Linux kernel

摘 要: 函数执行上下文正确性是操作系统内核程序最容易违反且难以检查的正确性性质.应用传统的技术检查该类错误都有一定的困难和局限性.提出一个验证函数执行上下文正确性的框架 PRPF,详细描述了其建模过程和相关算法.PRPF 相比传统技术的优势有:直接检查源代码、无须编写形式化的验证规约、较低的时空运行开销、良好的可扩展性等等.该技术已应用在 Linux 内核 2.4.20 的网络设备驱动程序检查中.应用表明,PRPF 能够自动探测程序中所有执行路径,有效地检查函数执行上下文的正确性.实验发现了 Linux 内核的 23 处编程错误,另有 5 处误报.该技术对提高内核代码编写的质量可起到重要作用.

关键词: 操作系统内核程序;内核编程接口;程序验证;程序正确性;Linux 内核验证
中图法分类号: TP316 文献标识码: A

* Supported by the National Natural Science Foundation of China under Grant No.60233020 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2002AAIZ2101 (国家高技术研究发展计划(863)); the Program for New Century Excellent Talents in University of China under Grant No.NCET-04-0996 (新世纪优秀人才支持计划)

Received 2006-02-24; Accepted 2006-06-12

当前,国际上操作系统的开发方兴未艾,主要的热点集中于开源操作系统(Linux,FreeBSD 等)开发和嵌入式系统开发(主要是设备驱动开发和移植)两方面.而国内,研发具有自主知识产权的国产操作系统也是一项迫切的任务.但现代操作系统内核的庞大和复杂为开发带来巨大的挑战,Windows 2000 内核的代码量据称有 4 000 万行,而 Linux 内核的代码量也已达到数百万行,即使是顶尖的内核程序员也难以对整个内核有精确的把握,从而对内核开发者的水平提出的要求也越来越高.

我们在开发具有自主知识产权的国产服务器操作系统 Kylin[1]以及关注 Linux 等开源操作系统的代码质量[2]的过程中,发现相当一部分的内核程序错误来自于对内核编程接口(kernel programming interface,简称 KPI)的错误使用.这是由操作系统内核程序不同于一般的用户应用程序的特点所决定的.现代操作系统为了支持众多而复杂的硬件和体系结构,内核结构层次极为复杂,运行时的并行路径非常多,这也决定了 KPI 一般都有严格的使用环境限制.例如,在中断上下文的运行时环境中调用可能阻塞的 KPI,或是在没有进程上下文的运行时环境中调用可能发生进程调度的 KPI(如睡眠 KPI、信号量 KPI),或是在临界区上使用错误的锁原语等,都会导致操作系统运行发生死锁等错误[3−7].文献[8]对操作系统的编程错误进行了分类和统计,发现 Block 类型的错误占大概 36%,是数量最多的一类错误.该类错误是因为在特定的执行上下文环境中错误地调用了 KPI 所造成的.根据文献[8]的研究,该类错误是程序员最容易忽视和难以检查的错误之一.本文中,我们将 KPI 必须运行在允许的执行上下文环境中这类程序的正确性性质定义为函数执行上下文正确性.目前,国际上已针对操作系统的一些领域的正确性验证问题提出了相应的技术和工具,如存储器访问安全[9]、网络协议[10]、文件系统[11]等代码的验证.SLAM 开发了一个工具 SDV(static driver verifier)[12]用于 Microsoft Windows 上基于 WDM(windows driver model)模型的设备驱动程序的验证.但目前的研究缺少针对函数执行上下文正确性的高效解决方法.

本文针对验证操作系统内核程序的函数执行上下文正确性,提出程序着色森林(PRPF)模型,并以此为基础提出一种轻量级的(相对通用程序抽象算法)、高效的框架和算法,通过集成工具的方式对用户屏蔽形式化理论方面的细节.

程序违反函数执行上下文正确性的一般情况是,一个 KPI 函数不能在某个执行上下文中被调用,而待验证程序可能运行在该执行上下文中,又调用了该 KPI 函数,则该程序违反了函数执行上下文正确性.以 Linux 为例,程序违反函数执行上下文正确性的典型情况有:(1) 在没有进程上下文的执行环境中调用进程级原语,如信号量、睡眠、调度原语;(2) 在不能阻塞的执行上下文中调用可能阻塞的函数.不能阻塞的执行上下文,典型的包括硬中断上下文、软中断上下文(softirq,tasklet,timer).可能阻塞的函数,典型的如参数为 GFP_KERNEL 的内存分配函数、访问用户空间的函数(如 copy_from_user(),copy_to_user())等.程序违反函数执行上下文正确性很可能导致系统死锁.一个具体的例子是,在 Linux 中,KPI 函数 vmalloc()不能在中断上下文中调用,因为它可能引起阻塞.因此,如果待验证程序是一个中断处理程序,程序中如果调用 vmalloc()来分配内存,则违反了执行上下文正确性,结果可能导致系统死锁.我们沿用文献[13,14]中对验证算法可靠性(soundness)、完备性(completeness)的定义.算法的可靠性是指程序中的所有错误必被算法报告.算法的完备性是指算法没有误报.于是,PRPF 模型对于检查函数执行上下文正确性是可靠的,不完备的,详见第 3.3 节中的证明.

现代操作系统一般都为内核开发人员提供一套锁原语,用于保证对临界区的互斥操作.锁原语也属于 KPI 函数,也存在前述的执行上下文正确性问题.此外,锁原语的特殊语义决定了锁原语需要满足的最重要的性质是互斥正确性.互斥正确性的概念比较广泛,传统的程序验证技术中验证的互斥正确性包括只加锁而未解锁、重复加锁等.实际上,还存在另一类由执行上下文带来的互斥正确性问题.例如,在 Linux 中,如果临界区可能同时在进程上下文和软中断上下文中运行,应使用 spin_lock_bh/spin_unlock_bh 原语保证互斥.而如果临界区会同时在进程上下文和硬中断上下文中运行,则应使用 spin_lock_irq/spin_unlock_irq 原语才能保证互斥.也就是说,需要详细考虑临界区所可能运行于的执行上下文集.要保证在不同上下文集之间的互斥,需要使用不同的锁原语.由于操作系统内核的复杂性,即使是资深的内核程序员,也难以精细地考虑所有情况,从而导致因使用错误的锁原语而使系统存在潜在的竞态(race)问题.该类问题涉及到特定操作系统中锁原语集的定义和功能.本文将其归结为函数执行上下文正确性.基于本文提出的 PRPF 模型,可以由程序自动并且高效地解决该问题.

本文第 1 节给出 PRPF 模型相关的基本概念.第 2 节描述建模过程并给出相关性质证明.第 3 节描述验证算法.第 4 节给出其应用情况.第 5 节进行相关工作比较.第 6 节总结全文.

1 基本概念

定义 **1**. 基元函数集 *BF*.指待验证程序中调用的 KPI 函数集.

定义 **2**. 自定义函数集 *EF*.指待验证程序中编写的函数集.

定义 **3**. 函数集 TF.$TF=BF\cup EF$.

定义 **4**. 函数调用关系 $Call:TF\times TF\to\{\text{True,false}\}$.

$$Call(f_1,f_2)=\text{True}\Leftrightarrow\text{函数 }f_1\text{ 调用函数 }f_2.$$

定义 **5**. 执行上下文集 *TS*.内核运行时的上下文集合.一种上下文就代表内核在运行时的一种典型的资源状态和执行路径.例如,现代操作系统运行时根据 CPU 所处特权环的不同,可分为用户态上下文和内核态上下文.内核态上下文进一步根据所运行内核代码代表的逻辑实体不同,可细分为用户进程上下文、内核线程上下文、中断上下文等.在 Linux 中,还有 Bottom Half 上下文、Tasklet 上下文[4];FreeBSD 中有中断线程上下文[15];Windows 中有核心服务上下文[16]等.

定义 **6**. 上下文发起函数集 *SF*.产生一个新的执行上下文的函数.这里的产生没有同步性要求,允许通过回调函数的形式异步地注册一个执行上下文.例如在 Linux 中,启动一个内核线程的 KPI 函数 *kernel_thread*(),该函数调用后,系统中将增加一个并行(本文不严格区分并发和并行,统称为并行)运行的内核线程上下文.又如注册中断处理程序的KPI函数*request_irq*(),该函数注册一个异步回调的中断处理程序,在该处理程序运行时,内核将处于中断上下文中.特别地,规定所有的程序入口函数,如 *main*(),*module_init*()等函数,均为上下文发起函数.一般地,除程序入口函数外,有 $SF\subset BF$.

定义 **7**. 色盒函数集 *LF*.一个执行上下文开始执行的第一个函数.一般只关心自定义函数中的色盒函数,即有 $LF\subseteq EF$.特别地,规定所有的程序入口函数,既是上下文发起函数,又是色盒函数.

定义 **8**. 锁集 *KF*.待验证程序中调用的所有锁原语.$KF\subset BF$.

定义 **9**. 颜色集 *CS*.我们要求在执行上下文集与颜色集之间存在双射 $g:TS\to CS$,即

$$CS=\{c:\forall c\in CS,\exists p\in TS,g(p)=c\wedge\forall p\in TS,\exists c\in CS,c=g(p)\wedge\forall p,p\in TS,g(p)=q(p)\Leftrightarrow p=q\}.$$

直观地说,即为每种执行上下文定义一个一一对应的颜色.

定义 **10**. 关系 $Run:TF\times TS\to\{\text{True,false}\}$.

$$Run(f,p)=\text{True}\Leftrightarrow\text{函数 }f\text{ 可能运行于执行上下文 }p\text{ 中.}$$

定义 **11**. 关系 $Color:TF\times CS\to\{\text{True,false}\}$.

$$Color(f,c)=\text{True}\Leftrightarrow\exists p\in TS,g(p)=c\wedge Run(f,p)=\text{True}.$$

直观地说,如果函数 f 可能运行于执行上下文 p 中,则 f 着有颜色 $c=g(p)$.

定义 **12**. 函数 $Run\_Color:TF\to 2^{CS}$.

$$Run\_Color(f)=\{c:c\in CS\wedge Color(f,c)=\text{True}\}.$$

直观地说,函数 *Run_Color* 返回 f 着有的所有颜色,对应 f 可能运行于其中的所有执行上下文.

定义 **13**. 关系 $NRun:BF\times TS\to\{\text{True,false}\}$.

$$NRun(f,p)=\text{True}\Leftrightarrow\text{基元函数 }f\text{ 不能运行于执行上下文 }p\text{ 中.}$$

定义 **14**. 关系 $FColor:BF\times CS\to\{\text{True,false}\}$.

$$FColor(f,c)=\text{True}\Leftrightarrow\exists p\in TS,g(p)=c\wedge NRun(f,p)=\text{True}.$$

直观地说,如果基元函数 f 不能运行于执行上下文 p 中,则 f 着有禁忌色 $c=g(p)$.

定义 **15**. 集合 $Forbidding\_Color:BF\to 2^{CS}$.

$$Forbidding\_Color(f)=\{c:c\in cs\wedge FColor(f,c)=\text{True}\}.$$

直观地说,函数 *Forbidding_Color* 返回 f 着有的所有禁忌色,对应 f 不能运行于其中的所有执行上下文.

定义 **16**. 集合 $Lock\_Color:KF\rightarrow 2^{CS}$.

$$Lock\_Color(f)=\{c:c\in cs,c=g(p),p \text{ 为所有使用 } f \text{ 能保证互斥的执行上下文}\}.$$

直观地说,$Lock_Color(f)$为 f 能锁住的上下文对应的颜色集合.

2 PRPF 模型

2.1 建模过程

模型建立的基本步骤如下:

1. 预处理.
2. 调用算法 Build-PRPF,输出待验证程序的运行时函数调用着色森林图.

预处理的主要工作有:

1. 建立知识库.知识库一旦建成,可反复使用,并且很容易扩充和维护.建立知识库的具体步骤为:
   - 1.1. 集合 *TS* 的生成:通过分析操作系统的技术文档、源代码等,归类总结得到集合 *TS*.*TS* 可以表示为一个符号常量集合,例如一个枚举常量集合.以 Linux 为例,可得到如下 *TS* 集合定义:*enum Linux_TS*{*SOFTIRQ*,*HARDIRQ*,*KERNELTHREAD*,…},其中每个常量代表一种执行上下文类型.
   - 1.2 集合 *BF*,*SF*,*KF* 的生成:通过分析操作系统的技术文档、源代码等可直接得到.主要记录函数名、函数参数描述、类型等域.
   - 1.3 对于每种上下文 $ts\in TS$,给出其对应的上下文发起函数集合,表示为二元组形式:$\langle ts,\{sf_1,sf_2,...\}\rangle$,$sf_i\in SF$.例如,在 Linux 操作系统中,上下文 *HARDIRQ* 对应的上下文发起函数集合的描述为 $\langle HARDIRQ,\{request\_irq()\}\rangle$.
   - 1.4 给出上下文发起函数对应的色盒函数获取方法描述.一般地,上下文发起函数调用或注册色盒函数有两种方式:一种方式是直接以色盒函数的函数指针作为调用参数.例如,在 Linux 中,内核线程启动函数 *kernel_thread* 的函数原型为 *int kernel_thread*(*int*(**fn*)(*void**),*void*arg*,*unsigned long flags*).其中,参数 *fn* 即为新产生线程的执行代码入口.因此,调用上下文发起函数 *kernel_thread* 时 *fn* 参数的值,即为对应的色盒函数.类似地,在 FreeBSD 中,注册协议栈回调函数的函数原型为 *int pfil_add_hook*(*int*(**func*)(),*void*arg*,*int flags*,*struct pfil_head*ph*).参数 *func* 的值即为对应的色盒函数.另一种方式是以一个复合数据结构作为参数,而色盒函数的指针作为该结构中的一个域的值.例如,在 Linux 中,注册网络协议栈回调函数 *nf_register_hook* 的函数原型为 *int nf_register_hook*(*struct nf_hook_ops*reg*),而色盒函数指针作为结构 *nf_hook_ops* 中的一个域.当然,如果对于结构中有多个函数指针的情况,则可以根据特定操作系统 KPI 的知识建立规则库来正确地识别色盒函数.
   - 1.5 给出双射 g 的定义,得到颜色集 *CS*.表示为二元组$\langle ts,g(ts)\rangle$,$ts\in TS$.
   - 1.6 针对 *BF* 中的每个基元函数 *f*,确定其所有不能运行于其中的执行上下文 *p*,由所有 $c=g(p)$构成该基元函数对应的禁忌色集 *forbidding_color*(*f*).如果该基元函数是锁函数,则进一步确定其不能锁住的执行上下文对应的颜色集 *CS′*(理论上,$CS'=CS-lock\_color(f)$),将 *CS′*并入该函数的禁忌色集中.表示为二元组$\langle f,forbidding_color(f)\rangle$.
   - 1.7 将以上集合存放在数据库中.
2. 从知识库中读入上述领域知识到内存中,建立相应的数据结构.利用传统的语法分析程序扫描待验证程序,从符号表[17,18]中得到程序中调用的所有函数名称,可得到集合 *EF*.

算法. Build-PRPF.

Procedure *Build-PRPF*(*P*)

Input: Program to be checked;

Output: A graph represents the runtime function call relation.

```
BEGIN
1.  Entrances=Find-All-Entrances(P);
2.  Trees=∅;
3.  For every element e∈Entrances, do
4.      Build-Function-Tree(e,e);
5.  Return Trees;
END
```

算法以待验证程序作为输入,其输出是一个森林.*Entrances*为一个字符串集合,算法Find-All-Entrances以待验证程序作为输入,找到程序中所有的入口函数,并将其函数名放入集合 *Entrances* 中.这一过程通过调用传统编译器的前端语法分析程序,再结合一定的语义知识便可以实现.事实上,在一般的 C 程序中,只有一个入口函数:*main*()函数.考虑到一些操作系统特殊的编程规定,其入口函数有特定的名称或者可以同时在一个程序中以宏定义等形式定义多个入口函数.例如,在 Linux 中,内核模块入口函数由 *module_init* 宏定义;在 FreeBSD 中,内核模块入口函数由 *DECLARE_MODULE* 宏定义;Windows NT 服务程序的入口函数为 *ServiceMain* 等.因此,对于特定的操作系统,可结合相应的语义知识来获得所有的入口函数.然后,算法对 *Entrances* 集合中每一个元素调用算法 Build-Function-Tree.该算法生成以代表该元素的结点为根的函数调用关系树,根据待验证程序行为,算法也可能同时生成其他调用关系树.*Trees* 为一个顶点集,集合中为森林中所有树的根结点.

算法. Build-Function-Tree.

```
Procedure Build-Function-Tree(s,s′)
Input: Function name string s,s′;
Output: Graph vertex f represents function s.
BEGIN
(1)  Initialization
1.      Generate a vertex f0;
2.      f0.name=s;
3.      f0.type=EF;
4.      f0.father=NULL;
5.      f0.color=Get_Color(s′);
6.      Trees=Trees∪{f0};
(2)  Scan
7.      While not reach the end of the function s, do {
8.          Get the next function call name s″;
9.          Type=Get-Function-Type(s″);
10.         Switch Type is: {
11.             Case a context start function:
12.                 s‴=Get-ColorBox-Function(s″);
13.                 f1=Build-Function-Tree( s‴, s″);
14.             Case a lock function:
15.             Case a base function:
16.                 Generate a vertex f2;
17.                 f2.name=s″;
18.                 Switch (Type) {
19.                     Case context start function: f2.type=SF; break;
```

20. Case lock function: $f_2.type=KF$; break;
21. Case base function: $f_2.type=BF$;}
22. $f_2.father=f_0$;
23. $f_2.color=f_0.color$;
24. Add edge (f_0,f_2) and Let f_2 be the rightest son of f_0;
25. Break;
26. Case a self-definition function:
27. f_2=*Build-Function-Tree*(s'',s');
28. Add edge (f_0,f_2) and Let f_2 be the rightest son of f_0;
29. *Trees*=*Trees*−{f_2};
30. Break;
31. Default:
32. Break;
33. }
34. }

(3) Return

35. Return f_0;

END

算法以函数名为参数,生成对应第 1 个调用参数(函数 s)的调用关系树.算法第 1~5 行首先生成代表该函数的树结点,并以该结点为树的根结点,第 6 行将其加入森林中.第 7~34 行顺序扫描该函数的实现代码,对遇到的函数调用进行处理.如果调用的是上下文发起函数,则通过递归调用在森林中形成一棵新的调用树.如果调用的是基元函数(包括上下文发起函数和锁函数),则生成一个树结点,并将该结点作为外层函数结点的当前最右子结点,颜色与外层结点的颜色相同,并记录下相应的结点类型.如果调用的是自定义函数,则通过递归调用扫描该自定义函数,生成该函数的调用树作为外层函数结点的当前最右子树.算法第 35 行返回生成的树的根结点.算法第 5 行的 Get-Color 过程根据上下文发起函数名查询知识库得到对应的执行上下文,再进一步得到该执行上下文对应的颜色,返回该颜色.第 12 行的 *Get-ColorBox-Function* 过程根据上下文发起函数得到对应的色盒函数,主要通过查询知识库来实现.

2.2 模型性质证明

如前所述,算法运行结果是得到一个森林,该森林具有以下性质:

(1) 森林中每棵树的根结点均为色盒函数.

证明:树的根结点只在调用算法 Build-Function-Tree 时产生,并且为算法的第 1 个调用参数所对应的函数.因此只需考察每次调用该算法时的第 1 个参数.第 1 次调用时,该参数为程序的入口函数,根据色盒函数的定义,满足性质(1)以后,是在第 13 行和第 27 行处的递归调用,首先考察第 13 行处的调用,调用参数是过程 *Get-ColorBox-Function* 的返回值,由前面的讨论可知,该过程返回值必为色盒函数,满足性质(1),再考察算法第 27 行处的调用,第 28~29 行将生成的树作为外层函数结点的右子树,并没有形成新的树.因此,性质(1)得证. □

(2) 完备性 1.森林中每棵树的深度优先的前序遍历为一个执行上下文可能执行的所有函数序列.

证明:根据性质(1),树的根结点必为色盒函数,因而是一个执行上下文的起点.算法总是按函数中的实现代码顺序扫描,考察所有的函数调用.在考察任意的函数 f_1 时,如果 *Call*(f_1,f_2)可能(这里的可能是因为根据算法的执行流程,对于分支结构,如 IF THEN ELSE,SWITCH CASE 结构,忽略其分支条件.即如果 f_1 是在一定的条件下调用 f_2,则仍然视为 f_1 无条件调用 f_2 处理,后文将对该处理进行解释)成立,则增加结点 f_2,增加边(f_1,f_2),f_2 作为 f_1 的子结点.当 f_1 有多个子结点时,最近产生的子结点总是作为当前最右子结点.根据算法 Build-Function-Tree 生成结点和边的方法,性质(2)显然成立. □

(3) 完备性 2.森林中树的棵数与程序运行时的独立并行的执行上下文数相等.

证明:执行上下文总由上下文发起函数发起,因此只需证明树的棵数与程序中上下文发起函数调用次数相等.根据性质(1)的证明,一棵新的树只在调用算法 Build-Function-Tree,且第 1 个参数为色盒函数时产生.因此,只需证明调用算法 Build-Function-Tree 且第 1 个参数为色盒函数当且仅当源程序中调用上下文发起函数.根据算法 Build-PRPF 和 Build-Function-Tree 的相应流程,这显然是成立的,性质(3)得证. □

(4) 完备性 3.森林中每棵树的结点对应的函数如果还运行在另一执行上下文中,则该结点必在另一棵对应的执行上下文树中出现.

证明:任何函数运行在某个执行上下文中,则该函数必在该执行上下文执行的函数序列中出现,结合性质(2)、性质(3),性质(4)显然成立. □

(5) 推论.一个特定程序中,一个基元函数在可能处于的执行上下文中的所有调用与它在该程序对应的 PRPF 模型森林中的所有出现一一对应.一个基元函数可能处于的不同类型的执行上下文与它着有的所有颜色一一对应.

证明:由性质(2)~性质(4)以及 Build-Function-Tree 的着色算法,推论显然成立. □

2.3 建模时空复杂性分析

假设程序的大小为 N,程序中调用上下文发起函数的次数为 m,则建模过程中对同一自定义函数的代码最多进行 m 次扫描.因此,整个程序最坏情况下会进行 m 次扫描.扫描过程中,过程 Build-Function-Tree 第 9 行函数 *Get-Function-Type* 和第 12 行函数 *Get-ColorBox-Function* 都可以通过查找知识库中的相应表格来实现.对于特定操作系统,该表的大小是固定的,因而可以认为查表操作在常数时间内完成.此外,唯一的操作就是生成树结点,该操作的复杂性为常数时间.因此,整个建模过程的复杂性为 $O(mN)$.根据操作系统内核程序的语义,程序中调用上下文发起函数都是为了完成一定的相对独立的功能.因此,可以认为 m 与 N 无关,而具有一个不大的常数上界.故建模算法是关于程序大小的线性算法,时间复杂性是比较低的.

模型中树的存储采用对称序穿线树.设在 PRPF 模型森林中,树的棵数为 m,树中含有的结点数最大为 n,函数名字符串长度最大为 q,则容易证明,存储整个模型所需的存储空间为 $O(qmn)$字节.根据建模算法,式中 m 的值等于程序中调用上下文发起函数的次数,如前分析,可以认为 m 与 n 无关而具有一个不大的常数上界.q 一般也具有一个不超过 20 的常数上界.故模型的空间复杂度为 $O(kn)$,为程序中函数调用次数的线性函数.可见,空间开销是比较低的.

3 PRPF 模型检验

3.1 算法依据

根据前述定义和证明,显然,一般 KPI 函数满足执行上下文正确性的条件是

$$\forall f\in BF, Run\_Color(f)\cap Forbidding\_Color(f)=\varnothing.$$

对于锁原语,还应满足:

$$\forall f\in KF, Run\_Color(f)\subseteq Lock\_Color(f).$$

该式等价于

$$\forall f\in KF, Run\_Color(f)\cap(CS-Lock\_Color(f))=\varnothing.$$

从而可以通过将 $CS-Lock_Color(f)$集合并入 $Forbidding_Color(f)$集合,而与前式统一.实际上,建立知识库时正是这么做的.

综上所述,程序满足函数执行上下文正确性的充分条件是

$$\forall f\in BF, Run\_Color(f)\cap Forbidding\_Color(f)=\varnothing.$$

注意,这里得到的是充分条件.这是因为我们在抽象程序中的分支语句时,采用的是保守的策略,即不考虑分支条件,将可能执行的函数调用作为无条件调用来处理.这样处理的理由是:(1) 操作系统程序要求的首先是

绝对的正确性,采取保守的策略,当一个程序被算法证明无错时,程序是肯定无错的.对于错误,算法打印出错路径,由程序员进一步检查是否是真的错误;(2) 由于只有在调用上下文发起函数时才生成一棵新的树,实际上,通过研究典型操作系统,如 Linux,FreeBSD 等的源代码可以发现,在条件分支中调用上下文发起函数的情况微乎其微.这从语义上也容易解释,通常发起一个执行上下文都是为了完成整个程序功能的一部分,因此很少作为条件调用.所以,算法的误报率是很低的,我们的实际应用也证明了这一点.

3.2 算法描述

下面描述基于 PRPF 模型的函数执行上下文正确性验证算法.

Procedure *Context-Correctness-Check*(*Trees*)

Input: Program runtime painted graph;

Output: Program error violated context correctness.

BEGIN

1. For every element $e \in Trees$
2. Do depth-first preorder traversal of tree *e*, for every node *f* visited
3. if (*f.type*!=*EF*) then
4. if (*f.color*∈*Forbidding_Color*(*f.name*))
5. then print (“Function: %*s* error used in %*s**n*”,*f.name*,*f.father.name*);

END

算法 Context-Correctness-Check 以 PRPF 模型为输入,对森林中的每棵树进行深度优先的前序遍历,对访问到的每个基元函数结点,检查该结点的颜色是否属于该函数的禁忌色集.如果属于,则报告程序违反了函数执行上下文正确性及出错函数和外层函数的名称.

3.3 算法正确性证明

算法 Context-Correctness-Check 对于验证函数执行上下文正确性是可靠的,但不是完备的.非完备性的引入来自建模过程中对程序控制流结构的简化处理,如前文所述.下面我们来证明算法的可靠性.

算法可靠性证明:

用反证法.假设程序中存在某个函数执行上下文正确性违反不被算法报告,则根据定义,必存在某个基元函数 *f*, *f* 可能运行于某个它的禁忌上下文 *p* 中.根据 PRPF 模型推论,一个基元函数在各种执行上下文中的所有调用与它在森林中的所有出现一一对应.因此,*f* 必在 *p* 对应的树中出现.算法对森林中的每棵树的每个基元函数结点进行检查,因此检查是完全的,*f* 在 *p* 中对应的结点必被检查.检查的内容是该函数结点着有的颜色是否属于该函数的禁忌色集,根据 Build-Function-Tree 的着色算法,检查的有效性是显然的.故 *f* 在 *p* 对应树中的出现必被算法报错.矛盾,从而算法可靠性得证. □

3.4 算法复杂性分析

设在 PRPF 模型森林中,树的棵数为 *m*,树中含有的结点数最大为 *n*,一个基元函数的 *Forbidding_Color* 集合的势最大为 *q*,则算法最内层(第 4 行)的基本操作为:判断一个元素是否属于某个集合的算法复杂度为 $O(q)$(如果事先对颜色集定义一个序关系,并将每个基元函数结点的禁忌色集事先排序,则复杂度可以降低为 $O(\lg_q)$).算法第 3 行的判断操作为常数时间.算法第 2 行对一棵树进行深度优先的前序遍历,如前所述,树采用对称序穿线树存储,由于遍历算法不会经过一个结点超过 2 次(遍历算法及证明从略),故算法最坏情况下的复杂度为 $O(2n)$.算法第 1 行对森林中的每棵树进行循环,复杂度为 $O(m)$.故算法总的复杂度为 $O(2mnq)$.一般地,对于目前流行的操作系统,*TS* 集合的势不会超过 50,而显然有:$q \le |TS|$.*m* 为程序中调用上下文发起函数的次数,如前分析,可以认为 *m* 与 *n* 无关而具有一个不大的常数上界.令 $k=2\times m\times q$,则算法复杂度为 $O(kn)$.可见,算法是关于函数调用次数的线性算法,因而具有较低的时间开销和较好的可扩展性.

4 应 用

我们基于 PRPF 模型理论,实现了一个检查函数执行上下文正确性的原型系统 OSChecker.当验证函数执行上下文正确性时,OSChecker 预先从知识库中读入 PRPF 模型需要的领域知识.然后,OSChecker 调用修改过的 C 编译器 CIL(C intermediate language)[19]对源代码进行分析,建立 PRPF 模型,运行前述算法,报告程序的函数执行上下文正确性验证结果.我们利用 OSChecker 对国内流行的操作系统 Red Hat Linux 9.0 所用的内核源代码 Linux Kernel 2.4.20 中的网络设备驱动程序目录 drivers/net 下的代码进行了检查,实验运行的硬件平台为 AMD Athlon(TM) XP 2800+1.25 GHz CPU,512M 内存的 PC 机,操作系统为 Red Hat Linux release 9.检查结果总共发现了 23 处错误,另有误报(false positive)5 处,误报率为 17.85%.相比之下,运用 MOPS(modelchecking programs for security)检查 Linux 应用程序的安全性[20]的误报率为 95.58%,运用编译器扩展的技术检查操作系统错误[21]的平均误报率为 36.36%,PRPF 模型检验算法的误报率是较低的.我们已将发现的错误详细列表发送给 Linux 内核的维护者,并已得到他们的确认.程序的运行结果见表 1.

Table 1

表 1

No.	File-Number of lines	Line number where error at	Wrong used KPI	Time elapsed (ms)	True error
1	aironet4500_core.c-3238	1 591	*DOWN*()	221.996 46	Y
2	appletalk/cops.c-1061	504	*schedule*()	72.081 421	Y
3	appletalk/cops.c-1061	621	*schedule*()	72.081 421	Y
4	appletalk/cops.c-1061	637	*schedule*()	72.081 421	Y
5	appletalk/cops.c-1061	665	*schedule*()	72.081 421	Y
6	bonding.c-2433	1 990	*kmalloc*()	129.307 007	Y
7	ethertap.c-378	266	*skb_clone*()	41.760 498	Y
8	defxx.c-3423	3 309	*dev_kfree_skb*()	125.399 658	Y
9	dl2k.c-1715	739	*dev_kfree_skb*()	131.299 927	N
10	irda/vlsi-ir.c-1337	693	*dev_kfree_skb*()	104.547 974	Y
11	ns83820.c-2103	519	*kfree_skb*()	134.556 518	Y
12	pcmcia/xirc2ps_cs.c-2129	456	*schedule_timeout*()	140.698 364	N
13	sb1000.c-1265	825	*dev_kfree_skb*()	142.855 957	Y
14	sb1000.c-1265	919	*dev_kfree_skb*()	142.855 957	Y
15	sgiseeq.c-727	175	*kmalloc*()	73.799 438	Y
16	sgiseeq.c-727	190	*kmalloc*()	73.799 438	Y
17	shaper.c-752	116	*sleep_on*()	57.827 881	N
18	sk98lin/skge.c-4100	2 261	*DEV_KFREE_SKB*()	218.068 97	Y
19	sk98lin/skge.c-4100	2 298	*DEV_KFREE_SKB*()	218.068 97	Y
20	sonic.c-624	235	*dev_kfree_skb*()	59.026 978	Y
21	sundance.c-1779	1 142	*dev_kfree_skb*()	131.611 938	N
22	wan/comx-hw-mixcom.c-963	340	*kfree_skb*()	93.531 372	Y
23	wan/comx-hw-mixcom.c-963	128	*kfree_skb*()	93.531 372	Y
24	wan/comx-hw-mixcom.c-963	358	*kfree_skb*()	93.531 372	Y
25	wan/comx-hw-mixcom.c-963	369	*kfree_skb*()	93.531 372	Y
26	wan/comx-hw-mixcom.c-963	404	*kfree_skb*()	93.531 372	Y
27	wan/lmc/lmc_main.c-2448	2 196	*dev_kfree_skb*()	148.144 409	Y
28	wireless/airo.c-6324	2 581	*down_interruptible*()	257.088 502	N

分析以上代码中存在错误的原因,一部分是因为驱动程序开发者对 Linux 内核的运行机制理解不够,在网络设备的中断回调处理函数中,调用了可能阻塞的 KPI 函数;还有一部分是因为 Linux 内核升级到 2.4 后,内核结构和运行机理有所变化,而驱动程序开发者没有针对新的内核对驱动程序进行升级.以上错误的纠正相对都较为简单,只需换用调用条件不同的,实现同样功能的 KPI 函数或代码即可.例如上面的 8,10,11,13,14,18,19,20,22,23,24,25,26,27 号错误,只需将对 *dev_kfree_skb*()/*kfree_skb*()的调用替换为对 *dev_kfree_skb_irq*()或 *dev_kfree_skb_any*()的调用即可.

对于误报(false positive)的情况,我们列出一个典型的例子:

```
/*drivers/net/shaper.c*/
static int shaper_lock(struct shaper*sh)
{
```

```
    /*Lock in an interrupt must fail*/
    while (test_and_set_bit(0,& sh→locked))
    {
        if (!in_interrupt())
            sleep_on(& sh→wait_queue);
        else
            return 0;
    }
    return 1;
}
```

程序中对可能阻塞的 KPI 函数 *sleep_on*()的调用是放在条件判断语句的一个执行分支中的,而不是无条件调用.由于 PRPF 模型对程序控制流结构的简化处理而造成误报.但正如理论分析和实践证明,误报率是较低的.

5 相关工作比较

验证程序的函数执行上下文正确性的传统技术有:(1) 程序员的手工检查.其局限性是明显的:现代操作系统的代码量非常庞大,控制流路径非常长,分支也很多,检查一条代码执行路径就需要数分钟甚至数小时的时间,而且手工检查的可靠性也是难以确定的.(2) 程序自动验证技术[14,22],可分为静态方法和运行时检查两大类.静态方法又可细分为静态分析(static analysis)、模型检验(model checking).已知的基于这些技术开发了不少工具用于程序正确性自动验证,比较通用和典型的有 SLAM[13],MAGIC[23],BLAST[24]等.

该类技术的局限性有:

(1) 理论上的局限性.静态方法都是通过一定的技术[25,26]对程序进行抽象,如 SLAM 采用谓词抽象(predicate abstraction)技术将程序自动转化为一个布尔程序抽象(Boolean program abstraction).BLAST 则先将程序表示为控制流自动机(control flow automata,简称 CFA),然后采用懒惰抽象(lazy abstraction)技术将 CFA 转化为一棵可达性树.但由于这些技术都是面向广泛的应用领域,基于的都是一种通用的程序抽象模型,因而通常是不精确的,尤其对于操作系统这样庞大而复杂的系统,很可能过于简化而忽略了关键的性质.另外,针对特定的应用领域,模型中很可能又包含了较多不必要的程序抽象.虽然现在已提出一些技术[27−29],如对抽象进行局部精化、基于反例的抽象精化等,但效果还不是特别令人满意.因此,通用的技术虽然能够解决较大范围的问题,但一般效率都较低.对于大而复杂的程序,容易存在状态空间爆炸的问题,运行的时空开销都很大.对于运行时检查技术,需要实际运行程序,这就排除了在开发过程中排错的可能性,从而不可能较早地发现错误.运行时检查一般会对程序性能产生一定的影响,而性能对于操作系统内核来说是至关重要的指标.另外,运行时检查必须依赖实际运行情况,理论上不可能检查所有的代码路径,因而是不可靠的.

(2) 应用上的局限性:基于程序自动验证技术的工具一般都需要用户使用特定的形式化的描述语言对要验证的性质进行描述,如 SLAM 要求用专门的 SLIC(specification language for interface checking)语言来描述待验证的程序性质,MOPS[30,31]要求用有限状态机(finite state automaton,简称 FSA)描述待验证属性.BLAST 要求对待验证属性写一个安全监视器(safety monitor),再和源程序一起编译重新生成一个新的程序.有的还需要用户在源程序中插入注释,这需要相当多形式化理论和模型检验方面的知识.而且即使是很简单的性质,用形式化的描述语言描述起来也都是比较费时的,稍微复杂一些的性质,其描述的正确性也难以保证.针对函数运行时上下文正确性性质,因为其具有强烈的语义色彩,与 KPI 的功能(领域知识)密切相关.而且该性质还与运行时相关,所以描述起来是相当困难的.MC(metacompilation)[21]虽然不用对待验证性质进行形式化的描述,但要求修改编译器实现元级编译(meta-level compilation),即程序员在编译器中插入对待验证属性的相应检查代码,这需要修改编译器.

另外,从发表的有关 MC 的文章来看,它没有实现对函数执行上下文正确性的检查.基于运行时检查的工具,一般需要修改程序源代码,对于规模庞大的操作系统内核,工作将会很繁琐,而且很容易影响操作系统本身的安全性和稳定性.

(3) 测试上的局限性:表现在操作系统内核由于代码的分支路径非常多,测试需要写很多的测试用例,并且难以确保用例能使代码运行到出错的地方.而且有些错误可能需要测试用例连续运行很长时间才能发现,如本文中提到的函数执行上下文编程错误,导致系统死锁的时间并不是确定的.况且,对于操作系统内核程序,一旦出错则难以有有效的保护措施,一般将直接导致系统的崩溃,从而对查找错误原因带来很大的困难.而且对于在操作系统代码中代码量和出错率都居于首位的设备驱动程序[8]来说,要测试还必须有相应的硬件.因此,传统的技术对于验证操作系统内核程序的函数执行上下文正确性都存在一定的局限性.

我们的解决思路是,针对特定应用领域背景中比较典型和常见的问题,提出一个面向验证特定性质的较精确的程序抽象模型,并以之为基础提出一个轻量级的(相对通用程序抽象算法)、高效的框架和算法,通过集成工具的方式对用户屏蔽形式化理论方面的细节.本文针对验证操作系统内核程序的函数执行上下文正确性,提出程序着色森林(PRPF)模型和相关验证算法.该方法的优点是:(1) 相对于手工检查和测试,该方法具备程序自动验证技术的优点,它采用静态分析的技术,自动检测程序中所有执行路径,无须程序员手工干预;(2) 相对于传统自动验证技术,它的优点是直接以程序源代码作为输入,无须程序员修改源程序,通过集成工具的形式从而也无须程序员用特定的形式化的语言描述待验证的性质,无须实际运行程序.由于算法基于的是专门针对特定问题优化的程序模型,所以运行的时空开销较低,具有良好的可扩展性,并且是可靠的(sound).应用表明,该技术可以以很高的效率和精确性发现操作系统内核程序中潜在的函数执行上下文错误,对提高操作系统内核代码的质量起到重要作用.

6 总 结

本文提出一种操作系统内核程序函数执行上下文正确性的自动检验方法.该方法基于程序着色森林(PRPF)模型,PRPF 模型根据执行上下文建立程序的抽象森林模型,并对其着色.建模算法的时空复杂性均为关于程序规模的线性复杂性.基于 PRPF 模型,我们提出验证函数执行上下文正确性的算法 Context-Correctness-Check.该算法是可靠的,为关于程序中函数调用次数的线性复杂性算法,具有较好的可扩展性.我们已将本文所提出的技术应用于 Linux 内核 2.4.20 的网络设备驱动程序检查中.应用表明,PRPF 能够自动探测程序中所有的执行路径,有效地检查函数执行上下文正确性.实验发现了 Linux 内核的 23 处编程错误,另有 5 处误报.

在将来的工作中,我们将整合更多的领域知识进入知识库,以便将 OSChecker 应用于检查更为广泛领域内的程序.另外,扩充和优化 PRPF 模型以验证更多种类的正确性性质,降低误报率也是我们进一步研究的方向.

致谢 在此,我们向对本文的工作给予支持和建议的同行以及在系统实现中提供帮助的同学,尤其是唐沛蓉同学表示感谢.

References:

[1] Kylin project. 2007. http://www.kylin.org.cn

[2] Linux kernel mailing list archive. 2007. http://www.uwsg.indiana.edu/hypermail/linux/kernel/

[3] Corbet J, Kroah-Hartman G, Rubini A. Linux Device Drivers. 3rd ed., O'Reilly, 2005.

[4] Love R. Linux Kernel Development. 2nd ed., Sams Publishing, 2005.

[5] Bovet DP, Cesati M. Understanding the Linux Kernel. 3rd ed., O'Reilly, 2005.

[6] Russell R. Unreliable guide to locking. 2003. http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html

[7] The Linux kernel API. 2005. http://kernelbook.sourceforge.net/kernel-api.html/

[8] Chou A, Yang J, Chelf B, Hallem S, Engler DR. An empirical study of operating systems errors. In: Proc. of the 18th ACM Symp. Operating Systems Principles. ACM Press, 2001. 73−88.

[9] Beyer D, Henzinger TA, Jhala R, Majumdar R. Checking memory safety with blast. In: Proc. of the FASE 2005. LNCS 3442, Springer-Verlag, 2005. 2−18.

[10] Musuvathi M, Engler DR. Model checking large network protocol implementations. In: Proc. of the 1st Symp. on Networked Systems Design and Implementation. San Francisco: USENIX, 2004.

[11] Yang JF, Twohey P, Engler D, Musuvathi M. Using model checking to find serious file system errors. In: Proc. of the OSDI 2004: the 6th Symp. on Operating Systems Design and Implementation. San Francisco: USENIX, 2004. 273−288.

[12] SDV tool. 2006. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

[13] Ball T, Rajamani SK. The SLAM project: Debugging system software via static analysis. In: Proc. of the POPL 2002. ACM Press, 2002. 1−3.

[14] Rai A. On the role of static analysis in operating system checking and runtime verification. Technical Report, FSL-05-01, 2005.

[15] McKusick MK, Neville-Neil GV. The Design and Implementation of the FreeBSD Operating System. Addison Wesley, 2004.

[16] Solomon DA, Russinovich ME. Inside Microsoft Windows 2000. 3rd ed., Microsoft Press, 2000.

[17] Muchnick SS. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1998.

[18] Aho AV, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[19] CIL project. 2007. http://sourceforge.net/projects/cil

[20] Schwarz B, Chen H, Wagner D, Morrison G, West J, Lin J, Tu W. Model checking an entire Linux distribution for security violations. In: Proc. of the 2005 Annual Computer Security Applications Conf. (ACSAC). Tucson: IEEE, 2005.

[21] Engler DR, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. of the 4th Symp. on Operating Systems Design and Implementation. San Diego: USENIX, 2000. 1−16.

[22] Engler D, Musuvathi M. Static analysis versus software model checking for bug finding. In: Proc. of the 5th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004). LNCS 2937, Springer-Verlag, 2004. 191−210.

[23] MAGIC project. 2004. http://www.cs.cmu.edu/~chaki/magic/

[24] Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with blast. In: Proc. of the SPIN. LNCS 2648, Springer-Verlag, 2003. 235−239.

[25] Graf S, Saidi H. Construction of abstract state graphs with PVS. In: Proc. of the CAV'97: Computer Aided Verification. LNCS 1254, Springer-Verlag, 1997. 72−83.

[26] Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In: Proc. of the POPL 2002. ACM Press, 2002. 58−70.

[27] Rusu V, Singerman E. On proving safety properties by integrating static analysis, theorem proving and abstraction. In: Proc. of the TACAS'99: Tools and Algorithms for Construction and Analysis of Systems. LNCS 1579, Springer-Verlag, 1999. 178−192.

[28] Clarke E, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-Guided abstraction refinement. In: Proc. of the 12th Int'l Conf. Computer Aided Verification (CAV). LNCS 1855, 2000. 154−169.

[29] Saidi H, Shankar N. Abstract and model check while you prove. In: Proc. of the CAV'99: Computer aided Verification. LNCS 1633, Springer-Verlag, 1999. 443−454.

[30] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In: Proc. of the 9th ACM Conf. on Computer and Communications Security (CCS). Washington: ACM Press, 2002. 235−244.

[31] Chen H, Dean D, Wagner D. Model checking one million lines of C code. In: Proc. of the 11th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2004. 171−185.

汪黎(1980－),男,四川南充人,博士生,主要研究领域为操作系统,编译优化.

王戟(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高可信软件技术,Agent 软件方法学.

罗宇(1963－),男,教授,主要研究领域为操作系统,分布式计算.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行体系结构,并行编译,并行操作系统.