软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(2):239−253 [doi: 10.13328/j.cnki.jos.004778]

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

设备驱动程序可靠性和正确性保障方法与技术研究进展*

张一帆[1,2], 黄 超[1,2], 欧建生[1,2], 汤恩义[1,2], 陈 鑫[1,2]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 陈鑫, E-mail: chenxin@nju.edu.cn

摘 要: 随着计算机技术的不断发展,计算机系统在安全攸关领域得到了广泛应用,其中的软件系统正逐渐成为重要的使能部件.在计算机系统中,设备驱动程序扮演了软件与硬件设备之间桥梁的角色.由于与计算机平台、操作系统、设备 3 个方面同时关联所导致的复杂性,设备驱动程序的开发难度大、成本高,程序中所存在的错误和缺陷常常导致系统失效,在安全攸关领域造成不可挽回的损失.以设备驱动程序可靠性和正确性保障为目标,分别从故障的隔离与恢复、正确性分析和验证、设计建模与复杂性控制这 3 个方面对当前相关方法和技术进行分析,为开展进一步深入的研究工作打下基础.

关键词: 安全攸关软件系统;设备驱动程序;可靠性;正确性

中图法分类号: TP311

中文引用格式: 张一帆,黄超,欧建生,汤恩义,陈鑫.设备驱动程序可靠性和正确性保障方法与技术研究进展.软件学报,2015,26(2):239−253. http://www.jos.org.cn/1000-9825/4778.htm

英文引用格式: Zhang YF, Huang C, Ou JS, Tang EY, Chen X. Research on reliability and correctness assurance methods and techniques for device drivers. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):239−253 (in Chinese). http://www.jos.org.cn/1000-9825/4778.htm

Research on Reliability and Correctness Assurance Methods and Techniques for Device Drivers

ZHANG Yi-Fan[1,2], HUANG Chao[1,2], OU Jian-Sheng[1,2], TANG En-Yi[1,2], CHEN Xin[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](State Key Labotary for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: With the rapid development of computer technology, computer systems have been widely used in the safety-critical field where software systems are important enabling components. In computer systems, device drivers act as the bridge between software and devices. Due to the complexity resulted from the fact that device drivers are associated with platforms, operating systems and devices all together, the development of device drivers is very difficult and costly. Errors and faults in device drivers often lead to system failures, causing irreparable damage to the safety-critical applications. Aiming at the assurance of reliability and correctness, the paper presents a survey of related methods and techniques from three aspects: failure isolation and recovery, correctness analysis and verification, model based design and complexity control. The mainstream methods and techniques are evaluated with their pros and cons, which lays the foundation for the further research.

Key words: safety-critical software system; device driver; reliability; correctness

安全攸关系统[1]是一类对可靠性和正确性要求极高的系统,它一旦失效,则将导致生命和财产的重大损失或是周边环境的严重破坏.安全攸关系统的主要功能是控制各种设备安全运行,以完成设定的任务.

* 基金项目: 国家重点基础研究发展计划(973)(2014CB340703); 国家自然科学基金(91318301, 91118002, 61321491, 61402222); 教育部高等学校博士学科点专项科研基金(20110091120058); 江苏省产学研项目(BY2014126-03)

收稿时间: 2014-07-01; 修改时间: 2014-10-31; 定稿时间: 2014-11-26

目前,安全攸关系统典型的应用领域包括:

- 军事,如武器系统作战保障系统空间开发项目等;
- 工业,如生产过程控制机器人控制等;
- 交通,如飞机控制系统空中交通控制火车自动调度与控制车载计算机系统智能交通系统等;
- 通信,如突发事件应急系统、电话系统的紧急呼叫业务等;
- 医疗,如放射性治疗仪医用监视系统医用机器人等;
- 核电站控制等.

在这些应用领域中,安全攸关系统除了依赖与应用相关的定制硬件以外,通常也依赖总线在计算机系统内部交换信息,依赖网络设备以及字符通信设备(例如串口驱动)与外界进行通信,依赖块存储进行数据的存储.因此在安全攸关系统中,总线设备以及相关的控制器、网络设备、字符通信设备、块存储设备的设备驱动程序是较为常见的.

从功能视角来看,设备驱动程序[2]是安全攸关系统中软件系统与设备进行交互的接口,它负责将软件系统对设备的请求转换成具体设备的控制命令,从而驱动设备的实际运行,同时,收集设备的状态信息并传递给软件系统.设备驱动程序在安全攸关软件系统中所处位置如图 1 所示.

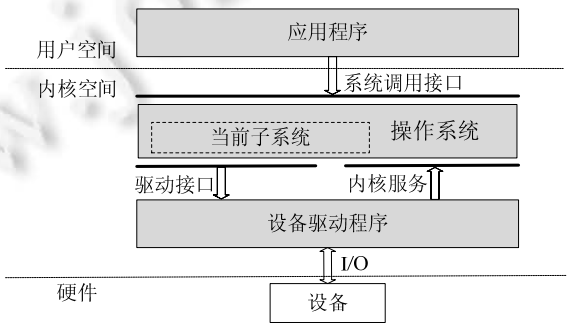

Fig.1 Position of device drivers in safety-critical software systems

图 1 设备驱动程序在安全攸关软件系统中所处的位置

设备驱动程序是安全攸关系统的关键组成部分,它的失效不仅意味着安全攸关系统失去对设备的有效监视和控制,更可能导致整个软件系统的失效甚至崩溃,最终引发重大的事故.设备驱动程序的可靠性和正确性对整个安全攸关系统的可靠性和正确性具有举足轻重的作用.

近年来,由于设备驱动程序的故障导致安全攸关系统失效,进而引发重大事故的事例层出不穷.例如,1986 年 3 月~1987 年 1 月,由加拿大原子能有限公司生产的 Therac-25 放射性治疗机因为设备驱动程序的控制逻辑存在问题,造成两人死亡、数人受伤.1990 年,美国电话系统中新投入使用的软件因设备驱动程序问题发生失效,导致主干远程网大规模崩溃,瘫痪时间长达 9 小时,造成了巨大的经济损失.2006 年,我国中航信离港系统就因为设备驱动程序故障发生了 3 次软件系统故障,造成近百个机场登机系统完全瘫痪.已有的研究表明:目前,设备驱动程序的可靠性和正确性大大低于其他软件.斯坦福大学的一项研究表明:Linux 操作系统中,设备驱动程序中出现 bug 的频率比其他模块要高 3~7 倍[3].因此,如何保障设备驱动程序的可靠性和正确性,是一个重要的技术需求.

设备驱动程序的高复杂性是导致设备驱动程序的可靠性和正确性难以保障的根本原因.为了实现操作系统规定的设备管理和操作规范,设备驱动程序通常需要和操作系统内核进行交互,需要访问设备的接口以监视和控制设备的运行,需要处理与设备所在物理平台相关的配置和资源.因此,设备驱动程序中都会包含与操作系统内核、与设备以及与物理平台上资源相关的 3 个方面的代码.在静态结构上,这些代码通常相互交织在一起;在运行时刻,由于需要响应异步事件如中断,这些代码会交错执行.这种结构上高度复杂的代码,很容易隐藏错

误.同时,由于异步执行的存在,硬件和软件事件交织所引发的驱动程序缺陷很难在测试中准确地重现,使得对设备驱动程序的测试往往是不充分的.因此,相对于其他软件,设备驱动程序的可靠性和正确性更加难以保障.

从开发的角度来看,为开发正确和高可靠的设备驱动程序,开发人员需要掌握关于对应的操作系统、设备以及设备所在硬件平台的完整和准确的知识.在实践中,这是难以完全做到的.开发人员对相关知识掌握得不够完整和准确,将直接导致其开发的设备驱动程序中存在各种类型的错误.这是设备驱动程序中容易存在错误的另一个重要原因.

如何保障设备驱动程序可靠性和正确性,是很多研究工作重点关注的问题.按照方法和技术手段的不同,这些研究工作可分为设备驱动程序故障隔离与恢复、设备驱动程序正确性分析与验证以及设备驱动程序设计建模与复杂性控制这 3 类:

- 设备驱动程序故障隔离与恢复类研究关注驱动程序的可靠性,它使用故障隔离与故障恢复技术,保证驱动程序以及宿主操作系统不会因为驱动程序的故障而停止运行,从而提高整个系统的可靠性.其中:隔离技术用于隔离驱动程序和其他操作系统组件,从而防止驱动程序的故障影响到宿主操作系统,并防止驱动程序执行可能引发其他宿主操作系统组件故障的操作;恢复技术用于在驱动程序故障后,将驱动程序恢复到某个可用状态,从而保证驱动程序始终能够正常运行.
- 设备驱动程序正确性分析与验证类研究关注驱动程序的正确性,它研究如何保证驱动程序正确实现了所需的功能,同时防止驱动程序执行错误或者不安全的操作.这类研究工作主要的技术手段有两类:一是静态分析技术,即通过静态分析驱动代码,在系统投入使用之前检测和消除驱动程序中的缺陷;二是运行时检查技术,即通过在运行时检查驱动程序操作的合法性,以保证驱动程序在运行时刻不会执行不安全的操作.也有一些研究工作结合静态分析技术和运行时检查技术,对那些静态分析无法证明是安全的操作,用运行时检查技技术来保证其在运行时刻的执行是安全的.
- 设备驱动程序设计建模与复杂性控制类研究通过设计建模辅助驱动程序的构造过程,控制驱动程序的复杂性,同时保证驱动程序的可靠性和正确性.这类研究基于形式模型对设备驱动程序进行设计,设计模型完成后,利用形式化分析和验证技术保证设计的正确性,再利用模型转换、程序合成和模块化等技术,从设计模型自动化或半自动化地生成设备驱动程序的代码.形式建模和分析技术可以保证设计的正确性,模型转换和程序合成技术可以减少人工向代码中引入的错误.这一类方法可以从根本上保证设备驱动程序的正确性并提高可靠性.

本文将对这 3 类研究工作中的代表性方法和技术进行分析和简介.

1 故障隔离与恢复

故障的隔离与恢复类研究针对的是设备驱动程序的可靠性问题,它综合使用故障隔离与故障恢复技术,以保证设备驱动程序自身和宿主操作系统不会因为设备驱动程序的故障而停止运行,从而提高设备驱动程序以及宿主操作系统的可靠性.其中,

- 故障隔离技术用于隔离设备驱动程序和其他操作系统组件,防止设备驱动程序执行可能引发其他宿主操作系统组件故障的操作.同时,故障隔离技术可以在运行时刻检测设备驱动程序故障(包括设备驱动程序自身及其依赖服务的程序的故障以及硬件设备的故障),从而防止设备驱动程序的故障影响到宿主操作系统.故障隔离可以通过硬件保护机制或者纯软件技术来实现.
- 故障恢复技术负责进行补偿行动,用于在设备驱动程序发生故障后,将设备驱动程序及其相关联部分恢复到某个可用状态,从而保证设备驱动程序始终能够正常运行.例如,故障恢复技术可以在设备驱动程序发生故障后恢复设备驱动程序的状态到故障发生之前的状态,或者报告故障给所有的已存在的设备驱动程序的客户,并且创建一个新的设备驱动程序实例来提供服务.

从设备驱动程序开发的角度来看,这类研究工作通常通过封装设备驱动程序、或者控制设备驱动程序的执行来实现故障隔离与故障恢复,因此,原有设备驱动程序一般不需要改造、或者仅需进行很少的改造即可用于

这类研究的系统中,这保证了良好的前向和后向兼容性.这类研究工作的主要缺陷在于会引入一定的运行时开销,这导致其在实时系统或是某些资源受限系统中的应用十分有限.同时,这类研究工作只能被动地保障设备驱动程序的可靠性,对设备驱动程序的正确性没有任何保障,既无法保证设备驱动程序正确实现了功能,也无法应对设备驱动程序中存在恶意代码的问题.

从实现技术来看,这类研究工作都以隔离设备驱动程序和宿主操作系统的其他组件为基础.根据所采用的隔离技术的不同,可以将这类研究分为 4 类:基于操作系统体系架构的隔离、基于用户态驱动的隔离、基于虚拟机的隔离和基于软件层面扩展的隔离.其中,

- 第 1 类研究主要通过扩展现有操作系统体系架构,或者引入新的操作系统体系架构,来隔离设备驱动程序和操作系统其他组件,同时监控设备驱动程序的执行,避免设备驱动程序故障的传播.
- 第 2 类和第 3 类研究都在现有操作系统体系架构下,通过复用已有操作系统组件来实现隔离.第 2 类研究主要通过将整个设备驱动程序或者部分设备驱动程序从内核态迁移到用户态执行来隔离设备驱动程序和操作系统其他组件.第 3 类研究在不同的虚拟机中运行不同的设备驱动程序来实现隔离,它能够在设备驱动程序发生故障后,通过简单重启一个对应的虚拟机和相关设备驱动程序完成故障恢复.
- 第 4 类研究只通过软件层面的扩展,在逻辑上隔离设备驱动程序和操作系统其他组件.

下面将根据上面的分类来对故障的隔离与恢复类研究现状进行介绍.

1.1 基于操作系统体系架构的隔离

基于操作系统体系架构的隔离通过扩展现有操作系统体系架构或者引入新的操作系统体系架构来隔离设备驱动程序和操作系统其他组件,这类技术可以在运行时刻监控设备驱动程序的运行状况,自动检测故障、避免设备驱动程序故障的传播,部分技术还可以自动恢复故障的设备驱动程序.

依据使用的具体技术,可以将这类研究工作细分为两个方向:一个方向是以纯软件方式扩展操作系统体系架构,实现对设备驱动程序的封装,从而隔离设备驱动程序和操作系统其他组件;另一个方向则以硬件扩展为核心,从物理上隔离设备驱动程序与操作系统内核的执行环境和涉及资源.

在以纯软件方式扩展操作系统体系架构实现隔离的研究中,华盛顿大学 Nooks 研究组提出的 Nooks 体系结构[4,5]是最具有代表性的研究之一.该体系结构可为设备驱动程序运行提供支持出错隔离和自动恢复机制的环境,其在操作系统内核和设备驱动程序之间添加了一层透明的可靠层来隔离设备驱动程序和操作系统内核,从而防止了设备驱动程序的错误被引入操作系统内核,并提供基于内存快照的故障恢复技术.在 Nook 体系架构中,每个设备驱动程序运行于一个独立的、作为保护域的 nook 中,使得设备驱动程序与内核其他部分实现隔离.以 Nooks 体系架构为基础,Nooks 研究小组又为内核和每个驱动之间添加了一个名为 shadow driver 的代理[6]来监控驱动程序的运行、记录驱动程序的运行状态,从而实现在设备驱动程序故障时的透明恢复.与 Nooks 体系架构类似,Herder 等人的研究[7]通过使用新的操作系统设计实现了运行时故障检测、监控和替换故障操作系统组件以及设备驱动程序重启功能,保证设备驱动程序或者相关操作系统组件出现故障后,操作系统可以自动替换故障组件并重启设备驱动程序.微软研究院也实现了研究用操作系统 Singularity[8−11],其使用 SIP (software-isolated processes)进行封装,每个驱动、系统进程、应用程序都运行在单独的 SIP 中,以隔离故障.Spear 的研究[12]则定义了一类新的操作系统抽象 application abstraction,这类抽象描述应用程序包括了应用程序的配置信息、所需资源在内等丰富的元数据,从而使操作系统可以对设备驱动程序在内的各种应用程序在加载前进行在线和离线推理(online and offline reasoning),保证所加载程序的可靠性.该研究为设备驱动程序实现了这类抽象的一个子集,这个子集可以用来描述 I/O 和 IPC 资源,操作系统可以通过这个子集,在设备驱动程序加载前验证其的可靠性,即保证设备驱动程序对 I/O 和 IPC 资源的使用是安全的.Application abstraction 的一个子集已经在 Singularity 操作系统上实现.David 等人提出的 CuriOS 操作系统[13]则用分布式的思想来实现隔离.CuriOS 操作系统通过在调用方分布式地保存操作系统组件(包括设备驱动程序、操作系统服务等)的状态信息,保证操作系统组件的可重启性.CuriOS 同时限制了设备驱动程序的内存访问权限,进一步提高了可靠性.

以硬件扩展为核心扩展操作系统体系架构实现隔离的研究工作,基本上都是通过硬件来隔离设备驱动程

序使用的内存资源和操作系统其他组件使用的内存资源,从而在物理上实现隔离.其中,

- Forin 等人的研究通过使用新的 I/O 系统[14,15],允许设备驱动程序运行于用户态,并通过硬件防止设备驱动程序发生非法内存访问.
- Chiueh 等人的研究[16]基于 X86 分页和分段硬件实现了一种内部地址空间(intra-address space)保护机制 Palladium,该机制可以高效地保证内核中共享内存地址的不同设备驱动程序不会受到非法内存访问的影响.
- Oliveira 等人的研究[17,18]提出了一种新的操作系统的范式,其依赖软硬件协同来实现逻辑隔离,以保护操作系统不受驱动程序等不安全的操作系统扩展影响.他们设计了实现该范式的一种软硬件体系架构,讨论了相关软硬件的具体设计,同时,通过模拟器进行了实验评估,证明了这种系统有效提高了操作系统的稳定性,并且引入的性能开销在可接受范围之内(约 12%).
- Zhou 等人的研究[19]针对 ARM 架构已成为移动设备主流架构的现状,提出了一种基于 ARM 硬件的错误隔离机制.该机制使用了 ARM 处理器提供的内存域来创建沙箱,并在沙箱中运行不可信代码,从而隔离了不可信模块.特别的,在沙箱中运行的不可信模块不受结构限制(即可以和可信模块一样执行所有的系统调用、接收异常等),因此不需要对已有的设备驱动程序做更改.

1.2 基于用户态驱动的隔离

基于用户态驱动的隔离通过在用户态执行设备驱动程序来实现隔离.这类研究通过将整个设备驱动程序或者部分设备驱动程序(一般是初始化等复杂但不经常执行的代码)从内核态迁移到用户态执行,来实现设备驱动程序和操作系统其他组件的隔离,保证设备驱动程序无法执行一些危及整个宿主操作系统的可靠性操作,同时防止设备驱动程序的故障(主要是非法的内存访问以及类型错误)传递到操作系统其他组件.

根据移入用户态的代码比例,可以将这类研究分为两类:一类是纯用户态驱动,即,采用类似于微内核的思想,将设备驱动程序完整移入用户态执行,从而将设备驱动程序与操作系统其他组件完全隔离,但是这往往会引入较大的运行时开销,因为内核态和用户态的切换会带来较大的延迟和性能开销;另一类是半用户态驱动,即,只将部分设备驱动程序(一般是初始化等复杂、但不经常执行的代码)从内核态迁移到用户态执行,将设备驱动程序中频繁执行、性能攸关的部分仍置于内核态执行,通过牺牲一部分隔离效果来获得高于纯用户态驱动、甚至接近原生驱动的性能.

在纯用户态驱动研究中,比较典型的是 Boyd-Wickizer 等人的工作[20],该工作提出的 SUD 系统在用户线程中模拟 Linux 内核环境,从而可以在用户态运行未经修改的设备驱动程序.特别地,SUD 针对 PCI 设备,基于硬件来阻止恶意驱动通过 DMA、中断等硬件操作来破坏系统.Leslie B 等人的研究工作[21]则以纯用户态驱动的性能为主要关注点,通过应用一系列优化策略,在不引入明显性能损失的情况下将设备驱动程序(包括了高带宽网卡设备的驱动程序)完全移入用户态执行,该工作的原型系统实现在 Linux 系统上.除了以上研究工作以外,微软的 Windows 操作系统了也提供了用户态驱动框架(user-mode driver framework,简称 UMDF)[22,23].它是 Windows 设备驱动程序框架(windows driver foundation)[24]的一部分.通过该框架实现的设备驱动程序运行于用户态,仅能访问用户地址空间.虽然用户态驱动可使用的函数是内核态驱动的子集,但是两者使用相同的状态机和相同的 I/O 模型.基于微内核设计的类 Unix 系统 MINIX 3[25]则将设备驱动程序全部移入了用户空间,从而极大地减少了内核大小,同时提高了操作系统的整体稳定性.

在半用户态驱动研究中,最为典型的是 Ganapathy 等人提出的 Microdrivers[26,27].Microdrivers 试图寻求单内核和微内核的折衷,希望在最大化性能的前提下提高设备驱动程序的可靠性.Microdrivers 将设备驱动程序分为用户态模块和内核态模块两部分,将设备驱动程序中性能攸关或经常使用的部分(如发送接收网络报文和处理 SCSI 命令)放入内核态模块,将设备驱动程序中复杂但不被频繁执行的部分(如设备的初始化)移入用户态执行.用户态模块用单独的进程实现,并仅供内核态模块调用.用户态模块和内核态模块一起提供了完整的设备驱动程序功能.在保持设备驱动程序性能的前提下,Microdrivers 实现了接近纯用户态驱动的隔离效果.

1.3 基于虚拟机的隔离

基于虚拟机的隔离利用虚拟机技术来实现隔离,这类研究通过在不同的虚拟机中运行不同的设备驱动程序来隔离设备驱动程序和操作系统其他组件,保证设备驱动程序的故障不会逸出其所在虚拟机.此外,在设备驱动发生故障时,这类技术可以通过简单重启一个对应的虚拟机和相关设备驱动程序完成故障恢复.

基于虚拟机的隔离提供了非常好的隔离效果,同时,其故障恢复技术也相当简单、有效.同时,从复用角度来看,这类研究工作一般都可以在虚拟机中运行完全未改动的设备驱动程序,因而通常都具有非常好的前向和后向兼容性.但是,严重的性能损失是这类工作的致命问题.由于设备驱动程序被隔离在另一个虚拟机中,操作系统其他组件和设备驱动程序之间大量的 I/O 请求会带来频繁的虚拟机间场景切换,这些切换远比传统操作系统中核内外切换粒度要大,开销也要大得多,因此往往会导致操作系统相关子系统性能的大幅下降.

在这类研究工作中,最为典型的是剑桥大学提出的新一代 I/O 体系结构[28],该研究目标是提高当前操作系统中设备及其驱动程序的可靠性、可维护性和可控性.新一代 I/O 体系结构由 I/O 空间、设备一致化接口和设备管理器这 3 部分组成.I/O 空间的引入是为确保设备在隔离空间里履行职能,这样可通过限制设备故障可能带来的危害来提高系统的可靠性;为一类设备定义一组一致化的接口来提高驱动程序的可移植性,避免了对不同操作系统重新实行特定功能;设备管理器则为所有设备提供统一的控制管理接口,简化了系统的配置、诊断和设备问题处理.剑桥大学研究者们基于他们在 Xen[29−31]上的研究成果,成功实现了新一代 I/O 体系结构框架的原型.该原型将设备驱动程序置入与应用分离的单独资源控制域中,并可配置合适的硬件访问权限,同时依靠虚拟机 Xen 实现了所需的处理器和硬件上下文隔离.他们还在系统控制器中添加了一个设备管理子系统,集成了某些必需的设备控制和设备管理服务.在宿主操作系统与其他域驱动程序之间的通信方面,他们引入了基于共享内存和异步通讯机制的设备通道,设备通道提供了高性能数据传输的统一抽象接口.类似于新一代 I/O 体系结构,Erlingsson 等人的工作[32]把设备驱动程序认为是不安全的操作系统扩展,通过使用虚拟化技术,在 Windows 系统下使用虚拟机运行设备驱动程序,以保证操作系统的可靠性.LeVasseur 等人的工作[33]则将设备驱动程序及其依赖的原生操作系统一起运行在一个单独的虚拟机中,在实现故障隔离和恢复的同时,使得在不同操作系统下复用设备驱动程序成为可能(例如在 Linux 系统下,可以通过该技术复用的 Windows 系统的设备驱动程序).其原型系统已经在 Linux 系统下得到了实现.

1.4 基于软件层面扩展的隔离

这类研究工作通过软件层面的扩展来实现逻辑隔离,即,只是通过软件层面的扩展,在逻辑上隔离设备驱动程序和操作系统其他组件(设备驱动程序仍和操作系统其他组件一起运行于操作系统内核),并实现故障隔离和故障恢复,从而提高设备驱动程序的可靠性.

由于这类研究工作并不从物理上隔离设备驱动程序和操作系统其他组件,只是引入软件层面的扩展来实现逻辑隔离,因此,这类研究不会像基于用户态驱动的隔离以及基于虚拟机的隔离那样引入较大的性能开销,但是其隔离效果也会有一定的损失,通常只能隔离一些特定种类的故障.

在这类研究中,Wahbe 等人的研究[34]通过纯软件机制(software-based mechanism)实现了在同一地址空间内、不同模块之间的故障隔离.该机制将可信模块的代码和数据限定在地址空间内一个逻辑对立的域中,并在沙箱中运行不可信模块的代码.而 Zhou 等人提出的 SafeDrive[35]通过引入类似 Java 中 Annotation 的新语言机制来保证设备驱动程序代码的类型安全.在运行时,SafeDrive 的监控模块监控并记录所有设备驱动程序的运行状态,错误恢复模块则在设备驱动程序出现故障时进行恢复.Herder 等人的研究[36]显示了设备驱动程序不受限制地执行特权操作是设备驱动故障的根源之一,并展示了如何依据最小权限原则(principle of least authority)限制驱动的行为,包括使用细粒度的驱动隔离、运行时内存分配等,来减小设备驱动程序故障带来的影响.Ryzhyk 等人的研究[37]则指出,现有设备驱动程序中的并发访问、控制流混乱问题的起因是设备驱动程序仅被动地被外界环境调用.以此分析为基础,该研究提出了一种新的设备驱动程序架构,该架构为每个设备驱动程序分配单独的线程,该线程用来控制设备驱动程序的执行,设备驱动程序通过消息传递和其他线程交互,以此来解决并发访

问、控制流混乱问题.Kadav 等人的研究[38]则基于事务控制的思想实现了一种细粒度的错误容忍机制(fine-grained fault tolerance),其在驱动的每个函数入口保存驱动状态,并将驱动代码的执行视为事务,提供了基于事务的回滚机制.特别的,其保存的驱动状态也包括了设备状态(即设备寄存器中的内容),设备状态的保存基于现存的电源管理代码实现.实验结果表明:在该机制下,重启设备驱动程序的时间相比现有机制缩短了 79%.

2 正确性分析与验证

正确性分析与验证类研究试图证明设备驱动程序的正确性,即,保证设备驱动程序正确实现了所需的功能,同时防止设备驱动程序执行错误或者不安全的操作.在技术层面,静态分析和运行时检查是进行分析验证的主要技术手段.

虽然这类研究已经成功地找出了 Windows 和 Linux 等操作系统中大量的设备驱动程序缺陷,包括操作系统 API 调用错误[39]、内存泄露和非法访问问题[39−42]、不正确加锁[40−42]、与操作系统的交互错误[43,44]等,但是限于现有静态分析和运行时检查技术的能力,这类研究可以检测的缺陷类型有较大的局限性.目前,这类研究只能对内存安全、并发等一部分缺陷进行检测和排除,同时,这类研究也难以应对部分高复杂度的设备驱动程序.

根据使用的技术,可以将这类研究工作分成 3 类:第 1 类研究使用静态分析技术,即,通过静态分析设备驱动程序代码,在系统投入使用之前检测和排除设备驱动程序中的缺陷;第 2 类研究使用运行时检查技术,通过在运行时检查设备驱动程序操作的合法性来保证设备驱动程序不会执行不安全的操作;第 3 类研究工作结合静态分析技术和运行时检查技术,对那些静态分析无法证明是安全的操作,用运行时检查技术来保证其在运行时刻的执行是安全的.

下面将根据上面的分类来对缺陷的正确性分析与验证类研究现状做介绍.

2.1 静态分析

这类研究工作主要使用静态分析技术,在系统投入使用之前,静态分析驱动程序的源代码来检测和排除设备驱动程序中存在的缺陷.这类研究工作的优点是不会引入运行时开销,缺点是其分析能力受限于使用的抽象模型以及状态空间爆炸问题,故目前仍无法分析竞争等复杂缺陷类型.

在这类研究工作,Engler 等人提出的 Meta-level compilation(MC)[45,46]系统可以允许设备驱动程序实现者依据设备以及操作系统的实际特性,手动定义一系列的系统规则,并且自动检查编译后的代码是否遵循这些自定义的规则,从而保证设备驱动程序的正确性.而微软的 Ball 等人则为 Windows 系统实现了一个静态设备驱动程序验证工具 SDV[39],该工具预定义了操作系统模型以及 60 余条操作系统 API 调用规则,这些规则覆盖了保证设备驱动程序正确性的最基本需求.在此基础上,该工具可以静态分析设备驱动程序,找出误用操作系统 API 相关的缺陷.该工具的特点是高度自动化和很低的误报率.伯克利大学的研究者们则为发现 Linux 系统内核中的内存隐患以及不正确的加锁问题开发了模型检测工具 BLAST[40−42],该工具通过性质驱动的自动构造(automatic property-driven construction)技术创建设备驱动程序的抽象模型,并通过模型检验验证这些抽象模型. Ryzhyk 等人的工作[43]则为设备驱动程序和操作系统的交互协议建立了一套形式化模型,该模型基于状态机模型实现.该研究的特点是,一个形式化交互协议模型可以用来描述一族设备的驱动程序和特定操作系统之间的交互.Ball 等人的研究实现了基于谓词抽象(predicate abstraction)的 C 程序反例制导抽象精化(counterexample guided abstraction refinement,简称 CEGAR)工具[47],其主要特点是大幅降低了静态分析工具的误报率.相关实验结果表明:该工作对基于 Windows Driver Model 实现的 Windows 设备驱动程序误报率低于 4%,而对基于 Kernel- Mode Driver Framework 实现的 Windows 设备驱动程序的误报率则低于 0.05%,并且在性能等方面优于其他现有的基于 CEGAR 的模型检验工具.Amani 等人的研究[44]则指出:现有设备驱动程序的体系架构目标是为了实现操作系统要求的设备接口,这样的体系架构直接导致了驱动的动态行为难以进行自动或手动分析验证.该工作进而通过 Dingo[48]语言定义设备驱动程序协议,显式地规定设备驱动程序与操作系统之间的交互协议,并证明了这些协议有助于改善设备驱动程序架构,可以帮助现有的模型检验工具验证一些在传统设备驱动程序中无法验证的性质,并可以有效地提高验证速度.Kinder 等人则针对现有设备驱动程序的源代码难以获得的现状提出了

一个完整的体系架构,用于对二进制设备驱动程序进行静态分析[49].为了在没有类型信息的 C 程序上准确地跟踪数据和函数指针,他们使用有界地址追踪(bounded address tracking)技术.他们通过实验证明了该架构针对部分 Windows 设备驱动程序可以得到比现有工具更为精确的分析结果,并且已经成功分析了近 300 个闭源的设备驱动程序.

2.2 运行时检查

这类研究工作使用运行时检查技术,通过在运行时检查设备驱动程序操作的合法性来保证设备驱动程序不会执行不安全的操作.运行时检查技术一般都针对系统的动态特性,因此,这类研究工作检测的缺陷类型一般都与设备驱动程序的动态特性相关.对于设备驱动程序而言,其最大的动态特性就是设备驱动程序与操作系统的交互行为以及设备驱动程序与设备硬件的交互行为.

由于运行时检查会引入额外的开销,因此,单纯使用运行时检查技术的研究工作比较少,更多的研究工作都将运行时检查技术作为静态分析技术的一种补充,以最小化运行时检查带来的性能损失.

在单纯使用运行时检查的研究工作中,最典型的是 Williams 等人提出的动态检查框架[50].该框架将设备驱动程序移出可信计算基,在非特权模式下运行设备驱动程序,并在运行时监控设备驱动程序与设备硬件的交互行为,从而保证设备驱动程序与设备硬件的交互符合设备安全规约(device safety specification).该框架的原型系统实现在 Nexus 操作系统下.此外,目前包括 Linux,Windows 和 MacOS 在内的主流操作系统都使用了运行时检查技术来保证设备驱动程序不会进行非法的内存访问.

2.3 静态分析结合运行时检查

这类研究工作综合应用了静态分析技术和运行时检查技术,对那些静态分析无法证明是安全的操作,用运行时检查技术来保证其在运行时刻的执行是安全的.这类研究通过将运行时检查技术作为静态分析技术的一种补充,可以检测静态分析无法检测的一些缺陷,同时也能够最小化运行时检查引入的额外开销.

在这类研究工作中,Kadav 等人实现的工具 Carburizer[51]可以通过静态分析定位设备驱动程序中因为没有考虑硬件故障而引入的缺陷,并通过运行时环境来自动修复一部分缺陷.Condit 等人则针对内存安全的问题实现了 C 程序分析和转换系统 CCured[52],CCured 系统首先静态分析程序,并尝试证明程序的内存安全,对于无法证明的程序片段,则通过插入运行时检查来保证内存安全.Erlingsson 则实现了一个更为泛用的代码执行保护系统系统 XFI[53],其通过结合静态分析和运行时检查来保证代码执行的安全性.该工具主要检查 3 个方面的安全性:内存访问安全、控制流不任意跳转到模块外以及栈数据的完整性(例如,异常栈是否通过指针完整相连).Mao 等人提出的 LXFI[54]则针对权限提升攻击(privilege-escalation attack),通过引入 API integrity 来捕获 API 相关的协议,引入 module principals 来说明共享模块内不同部分的权限级别.LXFI 的编译器插件则可以根据这两方面的内容生成用于保证、检查以及传递权限的代码.其实验结果表明,LXFI 可以有效地帮助 Linux 系统内核抵御权限提升攻击.

3 设计建模与复杂性控制

设计建模与复杂性控制类研究致力于通过设计建模辅助设备驱动程序的构造过程控制设备驱动的复杂性,同时保证设备驱动程序的可靠性和正确性.这类研究一般通过构建即正确(correctness by construction)方法,结合形式化建模、程序合成、模块化、统一接口、使用高级语言等技术为设备驱动程序建立抽象模型,并自动化或半自动化地将抽象模型合成实际的设备驱动程序.这类研究通过设计建模和自动化设备驱动程序的构建,提高了设备驱动程序的抽象层次,降低了设备驱动程序的复杂性.而复杂性是设备驱动程序低可靠性和正确性的根源,因此,这类研究能够从根本上提高设备驱动程序的可靠性和正确性.同时,形式化建模有利于对设备驱动程序进行更好、更全面的分析和验证,而自动化设备驱动程序的构建则有利于减少开发过程中的人为干预,这都有助于保障设备驱动程序的可靠性和正确性.

可以根据使用的技术手段将这类研究细分为 3 类:第 1 类研究工作通过规范化设备驱动程序与外界环境

(主要包括所处硬件平台和操作系统)的接口来控制跨平台、跨操作系统的设备驱动程序的复杂性;第 2 类研究工作通过典型的构建即正确方法驱动合成(driver synthesis)来帮助开发人员为设备驱动程序建立抽象的形式化模型,以提高设备驱动程序的抽象层次,降低设备驱动程序的复杂性,并借助形式化模型的分析和验证与自动化设备驱动程序的构造过程来保设备证驱动程序的可靠性和正确性;第 3 类研究工作通过在设备驱动程序开发中使用高级编程语言或者引入一些高级编程语言的特性,帮助开发人员在语言提供的高抽象层次上构建设备驱动程序,从而控制设备驱动程序的复杂性,并提高设备驱动程序的可靠性和正确性.

3.1 使用统一驱动接口

这类研究工作通常是由工业界主导的,其目标是改善跨平台、跨操作系统设备驱动程序的体系架构,降低开发复杂度,提高设备驱动程序的可靠性和正确性.这类工作通过统一设备驱动程序与外界环境(主要包括设备所处硬件平台和操作系统)的交互接口,使得开发人员可以开发平台无关、操作系统无关的通用设备驱动程序,避免了同时维护多个不同版本设备驱动程序所带来的不一致性等问题,降低了跨平台、跨操作系统设备驱动程序开发的工作量和复杂度.同时,因为统一的交互接口在一定程度上屏蔽了操作系统等外界环境的实现细节,可以帮助设备驱动程序开发人员在更高的抽象层次上进行开发,并为设备驱动程序的调试等提供了更有力的支持,有利于提高设备驱动程序的可靠性和正确性.这类研究已经在 Linux 和 Windows 两个主流操作系统上得到了应用.

在这类研究工作中,最典型的是网络驱动程式接口规范(network driver interface specification,简称 NDIS)[55].NDIS 是由 Microsoft 和 3Com 共同提出的、针对网卡设备的统一接口规范,遵循 NDIS 规范的设备驱动程序可以在 Windows 下运行,也可以在有 NDIS 适配层的 Linux 系统下运行.Apple 和 Novell 提出的开放数据链路接口(open data-link interface,简称 ODI)[56]也提供了与 NDIS 同等的功能.HP,Intel,IBM 等厂商共同提出的统一驱动接口(uniform driver interface,简称 UDI)[57]则可适用于更多的设备,包括网卡设备、USB 设备等.UDI 是一个统一的、与平台无关、与操作系统无关的设备驱动程序接口规范,其旨在消除设备驱动程序对平台以及操作系统实现的依赖(包括 I/O 总线拓扑、CPU 字节序、多处理器、中断处理实现等).

虽然这类研究工作可以改善跨平台、跨操作系统设备驱动程序的体系架构,但是基于这类规范开发设备驱动程序,将无法利用大量的现存设备驱动程序(因为现存设备驱动程序针对特定平台和特定操作系统),必须完全重写设备驱动程序,这在很多场合可能是无法接受的;同时,某些实时操作系统、嵌入式操作系统由于自身的特殊限制,可能难以实现对应规范的适配层,也就无法使用这类设备驱动程序;最后,由于操作系统厂商的互相制衡等原因,目前真正在多个操作系统上广泛使用的统一驱动接口并不多.因此,目前实际采用这类技术开发的跨平台、跨操作系统设备驱动程序并不多.

3.2 驱动合成

驱动合成类研究工作通过典型的构建即正确方法驱动合成,综合使用形式化建模、程序合成、模块化等技术来帮助开发人员为设备驱动程序建立抽象的形式化模型,以提高设备驱动程序的抽象层次,降低设备驱动程序的复杂性,并借助形式化模型的分析和验证以及自动化设备驱动程序的构造过程来保证驱动程序的可靠性和正确性.

早期的驱动合成研究工作集中于为设备驱动程序定义领域特定语言(domain-specific languages,简称 DSL),然后直接由领域特定语言合成设备驱动程序代码.这类研究工作的缺点在于:只能通过领域特定语言描述设备驱动程序的部分简单特性和行为(通常集中于设备资源、设备访问操作),缺乏泛用性,特别是难以描述设备驱动程序与外界环境复杂的交互行为.产生该缺点的根源在于,这类领域特定语言只是为底层的硬件操作代码进行了一层更加可靠、方便、易读的包装.

在早期的领域特定语言类研究中,Mérillon 等人的研究定义了一个比较简单的描述硬件设备的接口定义语言(interface definition language,简称 IDL)Devil[58,59],其被用来抽象定义设备资源(主要是设备 I/O 端口和设备寄存器)以及设备 I/O 接口(即,设备 I/O 访问操作).Devil 的编译器可以生成对应的 C 语言代码,并检查生成的设备

访问函数是否和抽象设备定义一致.Conway 等人提出的 NDL[60]语言是 Devil 的一种改进,其采用与 Devil 类似的语法,但在 Devil 的基础上提供了对设备状态的抽象描述方式,并解决了 Devil 中存在的诸如没有明确描述设备所使用的协议、没有提供类型安全措施等问题.HAIL[61]是另外一种领域特定的设备驱动程序描述语言,该语言包括针对设备数据手册中的设备寄存器和寄存器位的描述、地址空间描述、设备实例的描述以及限制设备访问的不变式的描述.根据这些规范化的描述,HAIL 编译器可以生成对应的设备 I/O 访问操作以及可选的前、后置条件检测代码.Wittie 提出的 Laddie[62]是对 HAIL 的一种改进,其主要的改进内容是保证了类型安全,从而大幅降低了设备 I/O 接口被误用的可能性.Schüpbach 等人的工作[63]从传统的设备寄存器访问中分离了设备参数配置逻辑,为传统的 C 语言设备驱动程序提供了一种声明式的语言扩展,以简化设备参数配置逻辑的实现.

后期的驱动合成研究工作集中于模型驱动的设备驱动程序生成,这类研究工作常通过模块化技术来分解设备驱动程序,然后采用形式化语言来为不同模块建立形式化模型,最后通过程序合成技术来生成完整的设备驱动程序.用模块化技术对设备驱动程序进行分解,可以分离设备驱动程序中与操作系统、硬件设备、物理平台相关的部分,在一定程度上解决设备驱动程序非结构化的问题,降低单个模块的复杂度;用形式化模型抽象描述模块,提高了单个模块的抽象层次.这些都可以有效地降低设备驱动程序的复杂性,而复杂性正是设备驱动程序的低可靠性和正确性的根源,因此,这类技术有助于从根本上提高设备驱动程序的可靠性和正确性.同时,形式化模型的分析和验证以及程序合成技术的引入,也有利于保障设备驱动程序的可靠性和正确性.此外,这类研究工作可以帮助开发人员在开发过程中较早地发现设计或者实现上的缺陷,这对于减少缺陷修复代价具有重要的实践意义,因为开发阶段修复缺陷的代价往往远远低于在系统实现后修复缺陷的代价.当然,这类研究工作也存在一定的缺点,主要是这类研究通常很难全面地考虑实际设备驱动程序的特性、复杂性和复用性需求,因此这类研究工作往往只能适用于一些较为简单的设备或者一族特定的设备,其通用性仍有待提高,同时也难以满足实际设备驱动程序对复用性的基本需求.此外,目前形式化建模的表达能力仍有一定的限制,难以描述设备驱动程序中的部分复杂的特性和交互行为,可能造成形式化模型与代码之间的信息不对等.

在后期的模型驱动的设备驱动程序生成类研究中,Wang 等人的研究工作[64]通过事件驱动的有穷状态机来为设备的行为建模,然后从这样的形式化模型生成独立于具体平台(包括处理器、操作系统和其他硬件环境)的虚拟环境中的设备驱动程序,最后将虚拟环境映射到具体的平台,以生成用于具体平台的实际设备驱动程序.该方法的主要问题是虚拟环境到具体平台的映射过程过于简单,缺乏泛用性.但是,该方法通过对形式化模型进行模型检验,保证了设备驱动程序的执行时间是确定的,同时可以避免死锁等问题.Ryzhyk 等人的研究工作[48]首先定义了一种形式化的、基于状态机的语言 Dingo 来描述设备协议,以避免协议混乱和产生歧义等问题.Dingo 语言要解决的主要问题是设备驱动程序中的并发问题,它可以辅助用户将设备协议映射到事件驱动的代码(大部分映射仍需要手动完成),并可以自动生成检查代码,在运行时检验代码的执行过程是否遵循设备协议.以 Dingo 语言为基础,Ryzhyk 等人进一步提出了一个完整的设备驱动程序合成框架 Termite[65].Termite 定义了基于 Dingo 语言的规范化语言 Tingu 来编写设备接口和操作系统接口的规约,并可以通过这些形式化规约自动生成对应的设备驱动程序代码.该研究工作主要存在 3 个问题:一是需要用复杂的形式化语言编写对应的规范,这对于一般开发人员是难以接受的;二是对于复杂的设备驱动程序,可能存在状态空间爆炸问题;三是生成的设备驱动程序只能运行在 Dingo 运行时环境中,该环境只能通过将所有的方法调用串行化来解决并发问题. Ryzhyk 等人之后实现的 Termite2[66]则对 Termite 做出了近一步改进,其主要的改进点在 3 个方面:可以混合自动化的驱动合成和传统的手工驱动开发过程;能够从抽象精化(abstraction refinement)合成实际设备驱动程序;可以根据设备驱动程序的规约进行自动化的除错.Chen 等人的研究提出了一种设备驱动程序合成技术 Me3D,该技术为设备驱动程序建立 3 个形式化模型:设备功能模型(device features model)、硬件规约(specifications of hardware)和内核接口(in-kernel interfaces),然后根据这 3 种模型加上相关类库以及驱动程序配置参数生成对应的实际设备驱动程序.该研究的目的是通过自动化技术来减少开发过程中人为因素引起的错误.该研究的主要缺陷在于:它是一个针对单个设备驱动程序的解决方案,并不涉及设备驱动代码复用等问题.O'Nils 等人的研究[67]则关注嵌入式系统和软硬件的交互协议,提出了一种为具体的嵌入式操作系统合成设备驱动程序的方法.该方法通过设

备的协议规约、操作系统相关信息、处理器相关信息这 3 部分内容来自动合成设备驱动程序.该研究的主要缺陷在于,其涉及的设备都只有较为简单的功能和控制逻辑.

3.3 使用高级编程语言或者引入相关特性

这类研究工作通过在设备驱动程序开发中引入高级编程语言或者引入一些高级编程语言的特性,帮助开发人员在语言提供的高抽象层次上构建设备驱动程序,从而控制设备驱动程序的复杂性,并提高设备驱动程序的可靠性和正确性.这类研究工作普遍存在的一个缺点是:难以利用大量现存设备驱动程序,需要重新开发设备驱动程序.

Hsieh 等人在 1996 年的研究[68]就展示了编程语言的一些特性对操作系统可扩展性的重大影响,这些特性包括类型安全的指针转换、隔离不安全的代码以及用接口、模块等概念描述程序结构.随后,随着 Java 语言的流行,很多研究者都尝试将 Java 语言或者 Java 语言包含的高级语言特性引入到设备驱动程序开发中.其中,Von Eicken 等人的研究[69]尝试在操作系统中使用类似 Java 的高级语言,并通过限制对对象引用的直接访问来实现操作系统的保护域等在高级语言中缺失的特性.Yamauchi 等人的研究[70]则实现了一个可以在 Solaris 操作系统内核运行的 Java 虚拟机,该虚拟机能运行由 Java 编写的设备驱动程序,从而使设备驱动程序开发人员可以使用 Java 语言编写设备驱动程序.该研究还具体分析了由 Java 语言编写的设备驱动程序相对于由 C 编写的设备驱动程序的优越性.之后,Okumura 等人的工作[71]则成功地将 Java 虚拟机嵌入到 FreeBSD 操作系统内核,从而允许操作系统执行用 Java 语言编写的操作系统扩展(包括设备驱动程序),降低了设备驱动程序等操作系统组件的编写难度,同时提高了这些组件的可靠性和安全性;Renzelmann 的研究则没有直接在操作系统中嵌入 Java 虚拟机,而是结合用户态驱动技术实现了半自动代码转换系统 Decaf Drivers[72].该系统可以通过程序分析工具识别出性能敏感代码,并帮助程序员增量化地将其他非性能敏感代码自动转换成用户态 Java 程序,这些用户态的 Java 程序可以降低设备驱动程序的复杂度,提高设备驱动程序的可靠性和正确性.也有一些研究者针对设备驱动程序的固有特性(例如异步特性、数据驱动特性),引入了一些高级语言来简化相关代码的编写.例如,Chandrasekaran 等人的研究[73]通过引入新的语言 CLARITY 来简化事件驱动的异步操作代码的编写,并可以有效地检查异步操作代码是否安全.

4 结束语

设备驱动程序是安全攸关软件系统中最重要的组成部分,其可靠性和正确性和对整个安全攸关系统的可靠性和正确性具有决定性的影响.因此,设备驱动程序可靠性和正确性保障方法和技术是近年来的研究热点.

目前,设备驱动程序可靠性和正确性保障方法和技术研究主要集中在 3 个方向上:故障隔离与恢复、正确性分析与验证以及设计建模与复杂性控制.对比 3 个方向的研究,故障隔离与恢复类方法在运行时刻对发生故障的设备驱动程序进行隔离,或者帮助设备驱动程序从故障中恢复,从而保证设备驱动程序的故障不会危及整个系统的可靠性.其主要缺点是只能被动地提高设备驱动程序的可靠性,但是对正确性没有多大的帮助.同时,故障检测、隔离和恢复机制都会引入运行时开销,在实时系统或资源受限系统中,这类方法的使用受到很大的限制.正确性分析与验证类方法则尝试主动对设备驱动程序的正确性进行分析和验证,帮助开发人员检测和排除设备驱动程序中的错误.受到驱动程序固有的高复杂性以及分析技术能力的局限,目前这类方法只能对内存安全、部分并发行为错误进行有效的检测,对于设备驱动程序中其他类型的错误还缺乏有效的分析和检测方法.前两类研究工作共同的技术特征是,它们提供的方法和技术都是应用在已经构造完成的设备驱动程序上.与前两类不同,设计建模与复杂性控制类方法提供了一系列在设备驱动程序构造过程中保证其正确性的技术手段,保证了按照这样的方法构造出来的驱动程序一定不存在某些类型的缺陷.它们可以从根本上提高可靠性并保证正确性.同时,设备驱动程序固有的高复杂性是其相对较低的可靠性和正确性的根源,设计建模与复杂性控制类方法直接针对这一根本问题进行研究,我们认为,这是最值得进一步探索的研究方向.

展望未来的研究,我们认为未来的研究工作将重点关注以下几个方面的问题:

1) 故障隔离与恢复类研究工作将重点关注如何提高隔离技术和恢复技术的效率和透明性,以及开发针

对嵌入式系统、实时系统、分布式系统等有特殊技术需求的系统的隔离技术和恢复技术.其中,提高隔离与恢复技术的效率需要依靠与底层硬件的紧密协同;高透明性技术使得设备驱动程序被隔离后,上层应用程序感觉不到与隔离之前的差别,从而使具备隔离与恢复功能的驱动能够更好地与已有系统进行融合.

2) 正确性分析与验证类研究工作将开发能力更强的分析验证技术来增加可处理的错误和缺陷的种类,提高分析和验证技术的可扩展性,以处理更大规模的程序,以及综合应用静态分析技术和运行时检查技术以有效结合两种技术手段的能力.其中,针对特定类型的错误,如何从代码中抽象出有效的模型供分析和验证技术进行处理,是这类研究工作的关键点.

3) 设计建模与复杂性控制类研究工作应该是未来关注的主要方向.除了研究引入新的模型对设计进行抽象以降低复杂度之外,我们还认为:从体系结构的角度出发,为设备驱动程序设计更好的内部结构也是一个值得探索的途径.文献[65,74]已经在尝试将设备驱动程序中与操作系统内核、设备以及物理平台相关的代码解耦,分解为功能独立、依赖明确的若干模块,以降低由于设备驱动程序中与操作系统内核、设备以及物理平台相关的代码相互交织而造成的高复杂性.同时,一个良好的体系结构也可以显著降低模型的复杂度,方便基于模型的设计、分析和验证工作.此外,领域化的建模语言以及针对这类语言的从模型到代码的自动生成技术,也应是主要的研究方向.

4.1 一种模块化的驱动开发方法

以对相关研究的分析为基础,针对设备驱动程序的功能和结构特点,我们从体系结构入手,为设备驱动程序设计了一种模块化的体系架构和开发方法,将设备驱动程序解耦为功能独立、依赖明确的系统平台模块、设备模块、操作系统模块.我们认为:运用模块化的技术封装耦合度高的代码到独立的模块,再通过组装这些模块生成实际的驱动,可以有效地降低驱动代码相互交织带来的高复杂性,提高驱动的可复用性和质量.在以上述方法学为基础,并对大量实际设备驱动程序进行详尽分析之后,我们针对嵌入式 Linux,WinCE 以及 VxWorks 这 3 个嵌入式操作系统和以太网卡类外部设备,设计和实现了一种模块化的设备驱动程序开发框架,并基于此框架开发了 20 多个设备驱动程序,初步验证了为设备驱动程序内部设计新的体系结构的可行性.

在上述研究的基础上,我们下一步的工作将集中于通过形式化建模、驱动合成等技术来简化设备驱动中单个模块的开发过程,以进一步降低设备驱动程序的复杂度,并提高设备驱动程序的质量.目前,我们根据对大量设备驱动代码的深入分析,针对设备驱动程序并发、数据驱动的特点,拟以数据流模型为基础,为设备驱动程序中的设备模块建模,并实现自动化的代码生成工具.

References:

[1] Safety-Critical system. 1986. http://en.wikipedia.org/wiki/Life-critical_system

[2] Device driver. 1968. http://en.wikipedia.org/wiki/Device_driver

[3] Chou A, Yang J, Chelf B, Hallem S, Engler D. An empirical study of operating systems errors. In: Proc. of the ACM Symp. on Operating Systems Principles (SOSP 2001). New York: ACM Press, 2001. 73−88. [doi: 10.1145/502059.502042]

[4] Swift MM, Martin S, Levy HM, Eggers SJ. Nooks: An architecture for reliable device drivers. In: Proc. of the ACM SIGOPS European Workshop. New York: ACM Press, 2002. 102−107. [doi: 10.1145/1133373.1133393]

[5] Swift MM, Bershad BN, Levy HM. Improving the reliability of commodity operating systems. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 2003. 207−222. [doi: 10.1145/1165389.945466]

[6] Swift MM, Annamalai M, Bershad BN, Levy HM. Recovering device drivers. ACM Trans. on Computer Systems, 2006,24(4):333−360. [doi: 10.1145/1189256.1189257]

[7] Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS. Failure resilience for device drivers. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE Press, 2007. 41−50. [doi: 10.1109/DSN.2007.46]

[8] Hunt G, Larus J, Abadi M, Aiken M, Barham P, Fähndrich M, Hawblitzel C, Hodson O, Levi S, Murphy N, Steensgaard B, Tarditi D, Wobber D, Zill B. An overview of the singularity project. Technical Report, MSR-TR-2005-135, Washington: Microsoft Research, 2005.

[9] Schulte W. From dependable multi-user to dependable multi-application operating systems: Invited talk. In: Proc. of the ACM Workshop on Secure Execution of Untrusted Code. New York: ACM Press, 2009. 1−2. [doi: 10.1145/1655077.1655079]

[10] Hunt G, Aiken M, Fähndrich M, Mann T. Sealing OS processes to improve dependability and safety. In: Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems. New York: ACM Press, 2007. 341−354. [doi: 10.1145/1272998.1273032]

[11] Hunt GC, Larus JR. Singularity: Rethinking the software stack. ACM SIGOPS Operating Systems Review, 2007,41(2):37−49. [doi: 10.1145/1243418.1243424]

[12] Spear MF, Roeder T, Hodson O, Hunt GC, Levi S. Solving the starting problem: Device drivers as self-describing artifacts. In: Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems. New York: ACM Press, 2006. 45−57. [doi: 10.1145/1217935.1217941]

[13] David FM, Chan E, Carlyle JC, Campbell RH. CuriOS: Improving reliability through operating system structure. In: Proc. of the USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2008. 59−72.

[14] Rashid RF, Tokuda H. Mach: A system software kernel. Computing Systems in Engineering, 1990,1(2):163−169.

[15] Forin A, Golub D, Bershad BN. An I/O system for Mach. In: Proc. of the USENIX Mach Symp. USENIX Association, 1991. 163−176.

[16] Chiueh T, Venkitachalam G, Pradhan P. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 1999. 140−153. [doi: 10.1145/346152.346187]

[17] Oliveira D, Wetzel N, Bucci M, Navarro J, Sullivan D, Jin Y. Hardware-Software collaboration for secure coexistence with kernel extensions. ACM SIGAPP Applied Computing Review, 2014,14(3):22−35. [doi: 10.1145/2670967.2670969]

[18] Oliveira D, Navarro J, Wetzel N, Bucci M. Ianus: Secure and holistic coexistence with kernel extensions—A immune system-inspired approach. In: Proc. of the 29th ACM Symp. on Applied Computing. New York: ACM Press, 2014. 1672−1679. [doi: 10.1145/2554850.2554923]

[19] Zhou Y, Wang X, Chen Y, Wang Z. ARMlock: Hardware-Based fault isolation for ARM. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2014. 558−569. [doi: 10.1145/2660267.2660344]

[20] Boyd-Wickizer S, Zeldovich N. Tolerating malicious device drivers in Linux. In: Proc. of the 2010 USENIX Annual Technical Conf. USENIX Association, 2010.

[21] Leslie B, Chubb P, Fitzroy-Dale N, Götz S, Gray C, Macpherson L, Potts D, Shen YT, Elphinstone K, Heiser G. User-Level device drivers: Achieved performance. Journal of Computer Science and Technology, 2005,20(5):654−664. [doi: 10.1007/s11390-005-0654-4]

[22] User-Mode driver framework. 2004. http://en.wikipedia.org/wiki/User-Mode_Driver_Framework

[23] Microsoft Corporation. Architecture of the user-mode driver framework. Version 0.7. 2006. http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx

[24] Windows driver foundation. 2005. http://en.wikipedia.org/wiki/Windows_Driver_Foundation

[25] MINIX 3. 1987. http://www.minix3.org/

[26] Ganapathy V, Balakrishnan A, Swift MM, Jha S. Microdrivers: A new architecture for device drivers. In: Proc. of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI). 2007.

[27] Ganapathy V, Renzelmann MJ, Balakrishnan A, Swift MM, Jha S. The design and implementation of microdrivers. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2008. 168−178. [doi: 10.1145/1346281.1346303]

[28] Fraser K, Hand S, Neugebauer R, Pratt I, Warfield A, Williamson M. Safe hardware access with the Xen virtual machine monitor. In: Proc. of the Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS). 2004.

[29] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 2003. 164−177. [doi: 10.1145/945445.945462]

[30] Pratt I, Fraser K, Hand S, Limpach C, Warfield A. Xen 3.0 and the art of virtualization. In: Proc. of the Linux Symp. 2005. 65.

[31] Matthews JN, Dow EM, Deshane T, Hu W, Bongio J, Wilbur PF, Johnson B. Running Xen: A Hands-on Guide to the Art of Virtualization. Prentice Hall PTR, 2008.

[32] Erlingsson U, Roeder T, Wobber T. Virtual environments for unreliable extensions. Microsoft Research Techical Report, MSR-TR-2005-82, Microsoft Corporation, 2005.

[33] LeVasseur J, Uhlig V, Stoess J, Götz S. Unmodified device driver reuse and improved system dependability via virtual machines. In: Proc. of the Symp. on Operating Systems Design and Implementation. USENIX Association, 2004. 17−30.

[34] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 1994. 203−216. [doi: 10.1145/173668.168635]

[35] Zhou F, Condit J, Anderson Z, Bagrak I, Ennals R, Harren M, Necula G, Brewer E. SafeDrive: Safe and recoverable extensions using language-based techniques. In: Proc. of the Symp. on Operating Systems Design and Implementation. USENIX Association, 2006. 45−60.

[36] Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS. Fault isolation for device drivers. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE Press, 2009. 33−42. [doi: 10.1109/DSN.2009.5270357]

[37] Ryzhyk L, Zhu Y, Heiser G. The case for active device drivers. In: Proc. of the Asia-Pacific Workshop on Systems. New York: ACM Press, 2010. 25−30. [doi: 10.1145/1851276.1851283]

[38] Kadav A, Renzelmann MJ, Swift MM. Fine-Grained fault tolerance using device checkpoints. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2013. 473−484. [doi: 10.1145/2451116.2451168]

[39] Ball T, Bounimova E, Cook B, Levin V, Lichtenberg J, McGarvey CM, Ondrusek B, Rajamani SK, Ustuner A. Thorough static analysis of device drivers. In: Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems. New York: ACM Press, 2006. 73−85. [doi: 10.1145/1218063.1217943]

[40] Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with BLAST. In: Model Checking Software. Berlin, Heidelberg: Springer-Verlag, 2003. 235−239. [doi: 10.1007/3-540-44829-2_17]

[41] Beyer D, Chlipala AJ, Henzinger TA, Jhala R, Majumdar R. The Blast query language for software verification. In: Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2004. 2−18. [doi: 10.1007/978-3-540-27864-1_2]

[42] Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker Blast. Int'l Journal on Software Tools for Technology Transfer, 2007,9(5-6):505−525. [doi: 10.1007/s10009-007-0044-z]

[43] Ryzhyk L, Kuz I, Heiser G. Formalising device driver interfaces. In: Proc. of the Workshop on Programming Languages and Operating Systems. New York: ACM Press, 2007. [doi: 10.1145/1376789.1376803]

[44] Amani S, Ryzhyk L, Donaldson AF, Heiser Gernot, Legg A, Zhu Y. Static analysis of device drivers: We can do better! In: Proc. of the Asia-Pacific Workshop on Systems. New York: ACM Press, 2011. [doi: 10.1145/2103799.2103809]

[45] Engler D, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. of the Conf. on Symp. on Operating System Design & Implementation, Vol.4. USENIX Association, 2000. 1−1.

[46] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE Press, 2002. 143−159. [doi: 10.1109/SECPRI.2002.1004368]

[47] Ball T, Bounimova E, Kumar R, Levin V. SLAM2: Static driver verification with under 4% false alarms. In: Proc. of the Conf. on Formal Methods in Computer-Aided Design. FMCAD Inc., 2010. 35−42.

[48] Ryzhyk L, Chubb P, Kuz I, Heiser G. Dingo: Taming device drivers. In: Proc. of the 4th ACM European Conf. on Computer Systems. New York: ACM Press, 2009. 275−288. [doi: 10.1145/1519065.1519095]

[49] Kinder J, Veith H. Precise static analysis of untrusted driver binaries. In: Proc. of the Conf. on Formal Methods in Computer-Aided Design. FMCAD Inc., 2010. 43−50.

[50] Williams D, Reynolds P, Walsh K, Sirer EG, Schneider FB. Device driver safety through a reference validation mechanism. In: Proc. of the USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2008. 241−254.

[51] Kadav A, Renzelmann MJ, Swift MM. Tolerating hardware device failures in software. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 2009. 59−72. [doi: 10.1145/1629575.1629582]

[52] Condit J, Harren M, McPeak S, Necula GC, Weimer W. CCured in the real world. ACM SIGPLAN Notices, 2003,38(5):232−244. [doi: 10.1145/780822.781157]

[53] Erlingsson U, Abadi M, Vrable M, Budiu M, Necula GC. XFI: Software guards for system address spaces. In: Proc. of the Symp. on Operating Systems Design and Implementation. USENIX Association, 2006. 75−88.

[54] Mao Y, Chen H, Zhou D, Wang X, Zeldovich N, Kaashoek MF. Software fault isolation with API integrity and multi-principal modules. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 2011. 115−128. [doi: 10.1145/2043556.2043568]

[55] Microsoft and 3Com Corporation. Network Driver Interface Specification. 1989. http://www.ndis.com/

[56] Apple and Novell Corporation. Open data-link interface. 1992. http://support.novell.com/techcenter/articles/ana19921103.html

[57] Uniform driver interface. 1999. http://www.projectudi.org/

[58] Mérillon F, Réveillère L, Consel C, Marlet R, Muller G. Devil: An IDL for hardware programming. In: Proc. of the Conf. on Symp. on Operating System Design & Implementation, Vol.4. USENIX Association, 2000. 2−2.

[59] Réveillère L, Muller G. Improving driver robustness: An evaluation of the Devil approach. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE Press, 2001. 131−140. [doi: 10.1109/DSN.2001.941399]

[60] Conway CL, Edwards SA. NDL: A domain-specific language for device drivers. ACM Sigplan Notices, 2004,39(7):30−36. [doi: 10.1145/998300.997169]
[61] Sun J, Yuan W, Kallahalla M, Islam N. HAIL: A language for easy and correct device access. In: Proc. of the 5th ACM Int'l Conf. on Embedded Software. New York: ACM Press, 2005. 1−9. [doi: 10.1145/1086228.1086230]
[62] Wittie L. Laddie: The language for automated device drivers (ver 1). Technical Report, 08-2, Bucknell CS-TR, 2008.
[63] Schüpbach A, Baumann A, Roscoe T, Peter S. A declarative language approach to device configuration. ACM Trans. on Computer Systems, 2012,30(1):5. [doi: 10.1145/1950365.1950382]
[64] Wang S, Malik S, Bergamaschi RA. Modeling and integration of peripheral devices in embedded systems. In: Proc. of the Conf. on Design, Automation and Test in Europe, Vol.1. IEEE Press, 2003. 136−141. [doi: 10.1109/DATE.2003.1253599]
[65] Ryzhyk L, Chubb P, Kuz I, Sueur EL, Heiser G. Automatic device driver synthesis with termite. In: Proc. of the ACM Symp. on Operating Systems Principles. New York: ACM Press, 2009. 73−86. [doi: 10.1145/1629575.1629583]
[66] Ryzhyk L, Walker A, Keys J, Legg A, Raghunath A, Stumm M, Vij M. User-Guided device driver synthesis. In: Proc. of the USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2014. 661−676.
[67] O'Nils M, Jantsch A. Device driver and DMA controller synthesis from HW/SW communication protocol specifications. Design Automation for Embedded Systems, 2001,6(2):177−205. [doi: 10.1023/A:1011246731756]
[68] Hsieh WC, Fiuczynski ME, Garrett C, Savage S, Becker D, Bershad BN. Language support for extensible operating systems. In: Proc. of the Workshop on Compiler Support for System Software. 1996. 127.
[69] Von Eicken T, Chang CC, Czajkowski G, Hawblitzel C, Hu D, Spoonhower D. J-Kernel: A capability-based operating system for Java. In: Proc. of the Secure Internet Programming. Berlin, Heidelberg: Springer-Verlag, 1999. 369−393. [doi: 10.1007/3-540-48749-2_17]
[70] Yamauchi H, Wolczko M. Writing Solaris device drivers in Java. In: Proc. of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems. New York: ACM Press, 2006. [doi: 10.1145/1215995.1215998]
[71] Okumura T, Childers BR, Mosse D. Running a Java VM inside an operating system kernel. In: Proc. of the 4th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: ACM Press, 2008. 161−170. [doi: 10.1145/1346256.1346279]
[72] Renzelmann MJ, Swift MM. Decaf: Moving device drivers to a modern language. In: Proc. of the USENIX Annual Technical Conf. USENIX Association, 2009.
[73] Chandrasekaran P, Conway CL, Joy JM, Rajamani SK. Programming asynchronous layers with CLARITY. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. New York: ACM Press, 2007. 65−74. [doi: 10.1145/1287624.1287636]
[74] Chen H, Godet-Bar G, Rousseau F, Petrot F. Me3D: A model-driven methodology expediting embedded device driver development. In: Proc. of the 22nd IEEE Int'l Symp. on Rapid System Prototyping (RSP). IEEE Press, 2011. 171−177. [doi: 10.1109/RSP.2011.5929992]

张一帆(1989－),男,浙江杭州人,博士生,主要研究领域为软件工程,软件开发,设备驱动程序.

汤恩义(1982－),男,博士,主要研究领域为软件工程,新型软件测试方法与程序分析方法.

黄超(1988－),男,博士生,主要研究领域为信息物理融合系统的建模、验证、仿真、控制.

陈鑫(1975－),男,博士,讲师,主要研究领域为软件工程,软件测试,验证技术.

欧建生(1989－),男,硕士,主要研究领域为软件工程,自动化单元测试.