软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006265]

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

操作系统内核并发错误检测研究进展*

石剑君, 计卫星, 石峰

(北京理工大学 计算机学院,北京 100081)

通讯作者: 计卫星, E-mail: jwx@bit.edu.cn

摘 要: 并发错误是程序设计语言和软件工程领域的研究热点之一.近年来,针对应用程序并发错误检测的研究已取得了很大进展.但由于操作系统内核的并发和同步机制复杂、代码规模庞大,与应用程序级并发错误检测相比,操作系统内核的并发错误检测研究仍面临巨大挑战.对此,国内外学者提出了各种用于操作系统内核并发错误检测的方法.首先介绍了并发错误的基本类型、检测方法和评价指标,讨论了现有的并发错误检测方法和工具的局限性;接着,从形式化验证、静态分析、动态分析和静态动态相结合四个方面,对现有的操作系统内核并发错误检测的研究工作进行了分类阐述,并作了系统总结和对比分析;最后,探讨了操作系统内核并发错误检测研究面临的挑战,并对该领域未来的研究趋势进行了展望.

关键词: 操作系统;多线程并行;并发错误;缺陷检测

中图法分类号: TP311

中文引用格式: 石剑君,计卫星,石峰.操作系统内核并发错误检测研究进展.软件学报. http://www.jos.org.cn/1000-9825/6265.htm

英文引用格式: Shi JJ, Ji WX, Shi F. Recent progress of concurrency bug detection in operating system kernels. Ruan Jian Xue Bao/Journal of Software, 2021 (in Chinese). http://www.jos.org.cn/1000-9825/6265.htm

Recent Progress of Concurrency Bug Detection in Operating System Kernels

SHI Jian-Jun, JI Wei-Xing, SHI Feng

(School of Computer Science and Technology, Beijing Institute of Technology, Beijing 10081, China)

Abstract: Concurrency bug detection is a hot research topic in the area of programming language and software engineering. In recent years, researchers have made great progress in concurrency bug detection of applications. However, as operating system(OS) kernels always have high concurrency, complex synchronization mechanisms and large scale of source code, researches on concurrency bug detection of OS kernels are more challengeful than applications. To address this problem, domestic and foreign researchers have proposed various approaches to detect concurrency bugs in OS kernels. In this paper, we first introduce the basic types, detection techniques and evaluation indicators of concurrency bug detection, and the limitations of existing concurrecny bug detection tools in OS kernels are discussed. Then, researches on concurrent bug detection in OS kernels are classified and described from four aspects: formal verification, static analysis, dynamic analysis, combination of both static and dynamic analysis. Some typical researches are systematically organized and compared. Finally, the challenges of concurrency bug detection in OS kernel are discussed, and the future research trends in this field are prospected.

Key words: operating system; multithreading parallel; concurrency bug; defect detection

随着计算机软件技术的发展,多线程与并发编程已经广泛应用于软件系统的开发过程中.多个线程的并发交叉执行可能引发各种各样的并发错误.并发错误通常会使程序或系统陷入一种不确定的运行状态,甚至造成系统崩溃等问题,带来巨大的经济损失.例如,从 1985 年到 1987 年期间,导致多名患者丧生的 Therac-25 医疗事故[1],2003 年美国东北部地区的大停电事件[2],2012 年 Facebook 公司的 IPO 故障带来的数亿美元的经济损失等[3].由于多线程并发执行的不确定性,并发错误产生的原因往往难以追踪,并发错误的检测和修复也比串行程序错误更加困难[4].

研究人员对并发错误进行了大量研究,并提出了一些应对并发错误的措施,包括并发错误检测、并发错误避免、并发错误预测和诊断等,本文仅关注并发错误检测相关的研究.目前,大部分针对并发错误检测的研究工作主要集中在应用程序级[5−15].然而,操作系统内核作为软件栈中最底层的软件,一旦发生并发错误,将导致服务中断、系统崩溃等更加严重的后果.由于操作系统内核中采用复杂的同步机制、各种软硬件中断处理机制,以及错综复杂的线程调度机制,操作系统内核的并发错误的研究变得更加困难.此外,操作系统内核的代码规模庞大,尤其是开源操作系统内核,如 Linux 内核自 1991 年发布至今,由分布在全球各地的开发者共同开发和维护,且经过了多个版本的迭代更新,如 Linux kernel v5.0 的代码量已经多达 2600 多万行.复杂的代码结构和庞大的规模,使得操作系统内核级并发错误检测的研究面临巨大的挑战.因此,很多用于应用程序级并发错误检测的方法,并不能直接用于操作系统内核的并发错误检测.随着静态分析、动态分析、形式化验证等程序分析技术的发展,研究人员提出了很多用于操作系统内核并发错误检测的方法.

* 基金项目: 2018 年工业互联网创新发展工程-工业微服务与工业 APP 部署应用工业互联网平台测试床

Foundation item: 2018 Industrial Internet Innovation and Development Project - The Project of Industrial Internet Platform Test Bed for Industrial Microservice and Application Deployment

收稿时间: 2020-09-14; 修改时间: 2020-10-26; 采用时间: 2020-12-14; jos 在线出版时间:2021-01-22

本文对操作系统内核级并发错误检测相关的研究进行了广泛调研,首先在 IEEE、ACM、Web of Science、Springer、Elsevier、DBLP、Google Scholar 和 CNKI 等论文数据库和搜索引擎中进行检索,检索采用的英文关键字包括“kernel/os concurrency”,“kernel/os deadlock”,“kernel/os race”,“kernel/os atomicity violation”和“kernel/os order violation”等,然后对检索到的论文进行逐一筛选,去除掉那些与操作系统内核和并发错误检测不相关的研究论文,并通过这些论文的相关工作部分对文献进行补充,最终得到与操作系统内核并发错误检测相关的研究论文.图 1 中展示了主要相关研究论文的分布情况,自 2003 年至今,关于操作系统内核的并发错误研究整体呈增长趋势,尤其是 2016 年以来发表在 ASE、OSDI、SOSP、ATC、S&P 等顶级会议上的论文数目多达 12 篇.

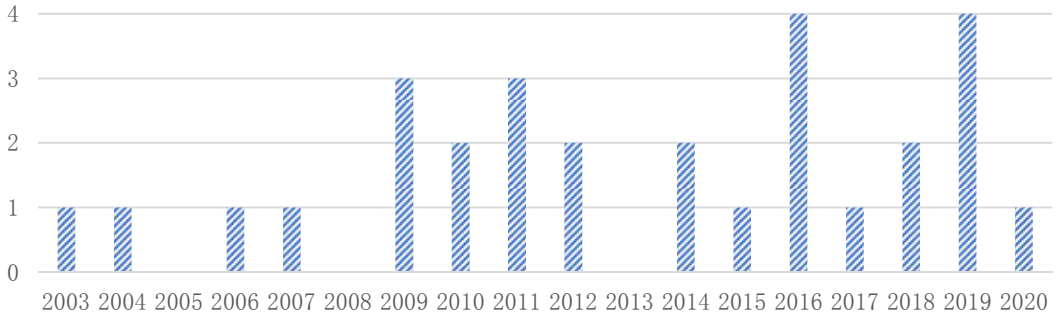

Fig.1 Literature distribution of concurrency bug detection in OS kernels

图 1 操作系统内核并发错误检测研究相关文献分布情况

本文第 1 节介绍了并发错误的基本类型、相关的并发错误检测算法和并发错误检测性能评价指标.第 2 节讨论了已存在的应用程序级并发错误检测方法无法直接用于操作系统内核的原因,以及操作系统内核中已有的并发错误检测工具的局限性.第 3 节从形式化验证、静态分析、动态分析和静态与动态相结合四个方面分类阐述了近年来针对操作系统内核并发错误检测相关的研究方法,并从技术手段、检测效果和性能等方面对相关的代表性研究工作进行了系统总结和对比分析.第 4 节对操作系统内核并发错误检测研究面临的挑战进行了探讨,并对该领域未来可能的研究趋势进行了展望.第 5 节总结全文.

1 并发错误简介

1.1 并发错误的基本类型

常见的并发错误包括死锁、数据竞争、原子性违例和顺序性违例等.下文以 Linux 内核中出现的并发错误为例,介绍死锁、数据竞争、原子性违例和顺序性违例的触发机制.

死锁是指两个或者多个线程互相等待对方释放系统资源而陷入无限等待的状态[16].在操作系统内核中,为了支持并发,开发人员采用了各种各样的同步机制.例如,在 Linux 内核中,广泛采用自旋锁(spin_lock)、互斥锁(mutex_lock)和顺序锁(seqlock)等,这些同步原语帮助开发人员保证了内核代码的正确执行,但同时也大大增加了死锁发生的可能性.图 2 展示了一个 Linux kernel v3.0.32 文件系统 jffs2 中的死锁实例.从图 2 中可以看出,有两个线程 *Thread1* 和 *Thread2* 并发执行函数 jffs2_garbage_collect_pass,其中 *Thread1* 先获取了自旋锁 c->erase_completion_lock,然后请求互斥锁 c->alloc_sem,而 *Thread2* 先获取了自旋锁 c->alloc_sem,又请求自旋锁 c->erase_completion_lock,从而造成了死锁.

Thread 1	***Thread 2***
int jffs2_garbage_collect_pass(…) {	int jffs2_garbage_collect_pass(…) {
…	…
spin_lock(&c->erase_completion_lock);	
	mutex_lock(&c->alloc_sem);
mutex_lock(&c->alloc_sem);	spin_lock(&c->erase_completion_lock);
…	…
}	}

Fig.2 An example of deadlock in Linux kernel v3.0.32

图 2 一个 Linux kernel v3.0.32 中的死锁实例

数据竞争是一种重要的非死锁并发错误[17].数据竞争的发生通常需具备 3 个必要条件:(1)两个或者多个线程并发地访问同一个内存位置,(2)至少有一个写操作,(3)没有采取适当的同步操作.数据竞争可分为良性数据竞争和恶性数据竞争.良性数据竞争是开发人员期望或者有意设计的一种数据竞争.例如,在操作性能计数器时允许数据竞争,这样就可以容忍计数值的小误差.恶性数据竞争是会对程序的运行时行为产生负面影响的数据竞争.程序非确定性的运行时行为,会导致系统出现故障.图 3 展示了一个

Linux kernel v3.10.66 中的数据竞争实例,该数据竞争发生在主线程 *Thread1* 和子线程 *Thread2* 之间,主线程调用 nlmclnt_init 函数创建了一个并发的子线程 *Thread2*, *Thread2* 调用了函数 lockd,并对共享变量 nlmsvc_timeout 进行初始化.由于 *Thread1* 和 *Thread2* 之间没有适当的同步操作, *Thread1* 和 *Thread2* 可以并发地访问共享变量 nlmsvc_timeout,可能造成未初始化变量访问的错误.

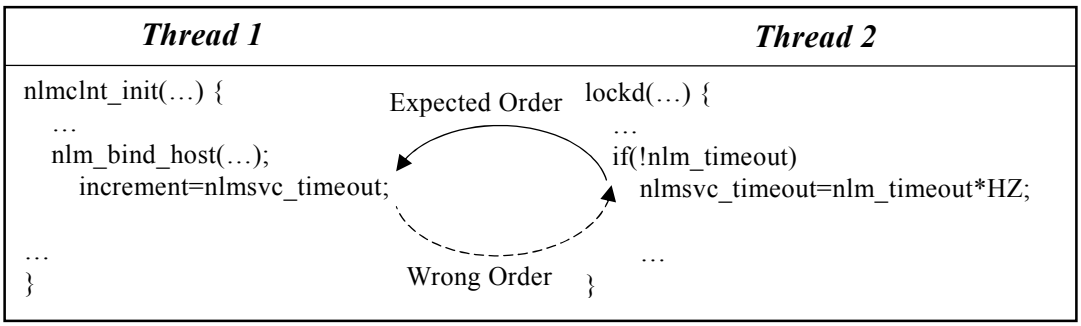

Fig.3 An example of data race in Linux kernel v3.10.66

图 3 一个 Linux kernel v3.10.66 中的数据竞争实例

原子性违例发生在开发人员对原子性操作的范围和粒度做出了错误的判断,而没有对那些应该出现在临界区的访存操作进行原子性地封装[18].图 4 展示了一个 Linux kernel v3.12.36 中的原子性违例实例, *Thread1* 和 *Thread2* 并发地访问函数 vmpressure_work_fn,其中 *Thread1* 检查变量 vmpr->scanned 是否为空,此时 *Thread2* 调度执行,并对变量 vmpr->scanned 赋值为 0,然后 *Thread1* 再次被调度执行,将 vmpr->scanned 的值赋给了 scanned,并在后续代码中进行了除法操作,从而引发除零错误.虽然线程在访问变量 vmpr->scanned 时有自旋锁 vmpr->sr_lock 的保护,但是由于加锁的粒度太小而造成了原子性违例错误.

顺序性违例是指程序中的两个或者多个访存操作必须按照某种顺序执行,而开发人员在程序编写过程中却没有保证它的顺序性而导致的并发错误[19].图 5 展示了一个 Linux kernel v3.11.10 中的顺序性违例实例. *Thread1* 调用函数_request_firmware_load,执行完函数 kobject_uevent 时发生线程调度, *Thread2* 开始执行并调用函数 firmware_loading_store,同时释放了 buf->pending_list.然后, *Thread1* 再次被调度执行,调用 list_add 函数执行 buf->pending_list 添加操作而报错.这是由于虽然对共享变量 buf->pending_list 的访问都加了锁保护,但没有保证 buf->pending_list 的释放必须在 buf->pending_list 添加之前.

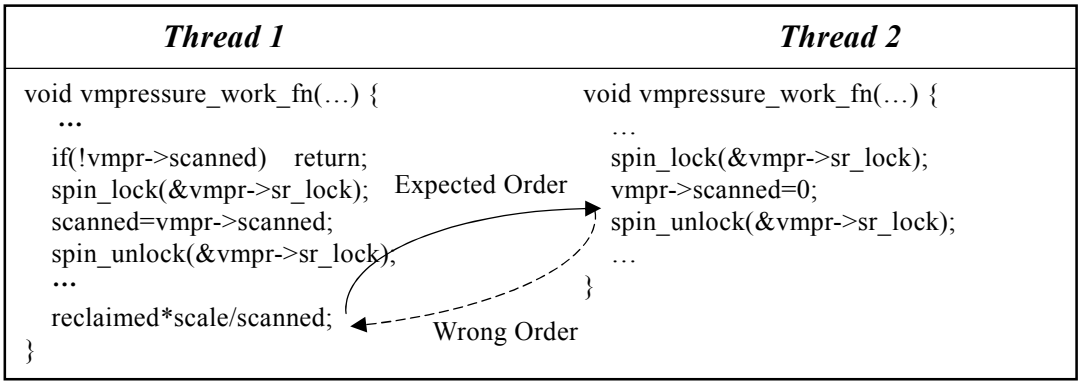

Fig.4 An example of atomicity violation of Linux kernel v3.12.36

图 4 一个 Linux kernel v3.12.36 中的原子性违例实例

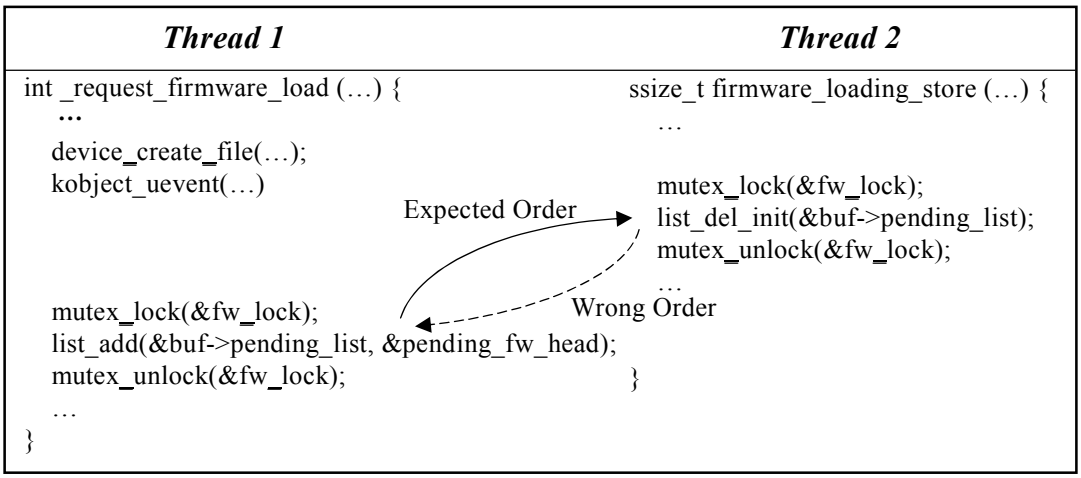

Fig.5 An example of order violation in Linux kernel v3.11.10

图 5 一个 Linux kernel v3.11.10 中的顺序性违例实例

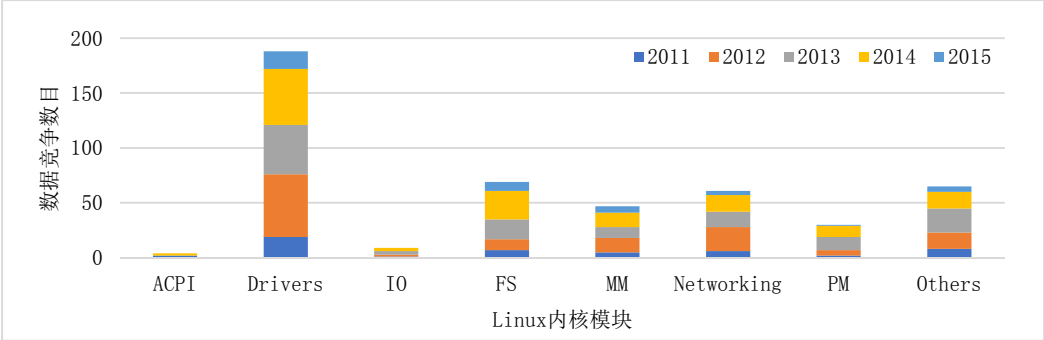

Fig.6 Distribution of data races in Linux kernel modules(2011-2015)

图 6 Linux 内核各模块中数据竞争的分布情况(2011-2015)

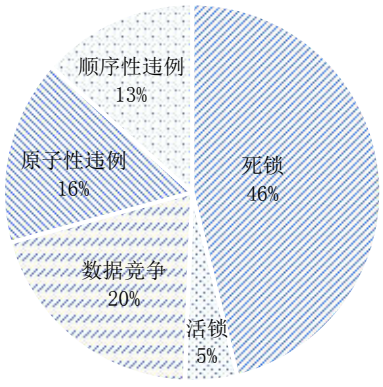

Fig.7 Distribution of concurrency bugs over categories in Linux kernel(2011-2019)

图 7 Linux 内核各种类型并发错误的分布情况(2011-2019)

研究发现,并发错误的这四种基本类型中,死锁与其他三类并发错误并不相关.非死锁的并发错误之间存在着关联关系,如很大一部分原子性违例或顺序性违例发生的本质原因是数据竞争[20].如图 3 中的数据竞争实例也可以被归为顺序性违例.此外,并发错误的类型不限于上述四种基本类型,除此之外还有活锁、Concurrency-use-after-free(CUAF)、Sleep-in-atomic-context(SAC)等.CUAF 是由于多线程并发而导致的 UAF 错误,相比于普通的 UAF 错误,线程的并发和交叉执行使其检测难度大大增加.SAC 错误通常出现在内核级,在原子性上下文中出现了睡眠操作[21],如在自旋锁加锁过程中或者中断处理程序中出现了睡眠操作,导致 CPU 长期处于阻塞状态而造成系统挂起或者死锁.研究人员对实际应用中的并发错误进行了调研分析.Shan Lu 等人对 MySQL、Firefox、

OpenOffice 和 Apache 中的并发错误进行研究,发现 97%的非死锁并发错误是原子性违例或者顺序性违例[19].Pedro Fonseca 等人[22]对 MySQL 中从 2003 年到 2009 年之间的并发错误进行了统计分析,发现在修复的错误中并发错误占的比例在 6 年中增长了两倍多,而且非死锁的并发错误(63%)比死锁(40%)要多.Sara Abbaspour Asadollah 等人[23]对 Apache、ZooKeeper、Spark 等开源软件中的并发错误进行了调研分析,发现并发错误平均修复时间(82 天)要比非并发错误(66 天)时间久.

我们之前的研究工作对 Linux 内核代码中的数据竞争进行了调研[24],通过分析 Linux 内核代码 5 年的修改日志发现,2011 年到 2015 年有将近 500 个数据竞争被确认和修复,Linux 内核各模块中数据竞争的分布情况如图 6 所示.其中,文件系统和驱动程序模块的数据竞争占比最大.Jiaju Bai 等人[25]对 Linux 内核中的 use-after-free(UAF)错误进行了调研分析,发现从 2016 到 2018 年之间,49%的 UAF 相关补丁来自于 Linux 内核驱动程序.而且 42%的 UAF 错误都与多线程并发相关.事实上,任何一个 Linux 内核驱动程序的设计和实现都需要依赖于 Linux 内核驱动程序接口和 Linux 内核中断处理接口两个模块,而各接口函数之间存在并发关系,因此驱动程序模块更有可能触发并发错误.此外,我们对 2011 到 2019 年 Linux 内核中文件系统相关的并发错误进行了调研分析,各种并发错误类型的占比如图 7 所示,从图 7 中可以看出将近一半的并发错误是死锁,数据竞争的数量也达到 20%.有研究表明,当前发布的操作系统代码中 39%的并发错误补丁都是不正确的,比内存错误补丁高 4~6 倍[4].因此,并发错误特别是操作系统内核并发错误的研究显得尤为重要.

1.2 并发错误检测方法

针对不同的并发错误类型,研究人员从并发错误的触发机制出发,结合形式化验证、静态和动态代码分析等技术提出了一系列用于并发错误检测的方法.

死锁的发生通常需要满足四个基本条件:互斥、非抢占、持有并等待和循环等待.死锁的检测需要构建出程序的资源依赖关系图,通过检查是否存在资源的循环依赖来进行检测.如 Amy Williams 等人[26]基于流敏感和上下文敏感的静态代码分析技术,构建出 Java 库代码的 lock-order 图,进而检查 lock-order 图中是否存在环,若存在环,则可能存在死锁.在此基础之上,很多研究人员采取了一系列的措施以减低死锁检测的误报率和漏报率.如 DeadlockFuzzer[27]和 MagicFuzzer[28]方法通过动态调度线程的方法以确认检测到的锁依赖循环是否是真正的死锁错误,降低了死锁检测的误报率.

数据竞争的检测需要对并发线程及其访存操作进行建模,分析是否满足数据竞争发生的 3 个基本条件.研究人员提出了 Happens-before(HB)关系和 Lokset 锁集算法用于数据竞争检测.HB 关系是一个最早由 Lamport 于 1978 年提出[29]的偏序关系,它表示在程序运行过程中两个事件之间的关系,即这两个事件在现实中的执行顺序.在同一线程内,事件的执行顺序是由它们发生的先后顺序决定的.而在不同的线程中,事件的执行顺序是由它们对线程中同步对象的访问权决定的.例如若存在两个并发线程 T1 和 T2,线程 T1 中的事件 a 先于线程 T2 中的事件 b 执行,那么必有事件 a 先于事件 b 获得同步对象的访问权限.因此,具有 HB 关系的事件是有先后关系的,反之,如果两个事件不具有 HB 关系,则被称为并发事件.如果存在两个并发事件同时访问了同一个共享变量,且其中至少一个是写操作,则认为对共享变量的访问时可能存在数据竞争.基于 HB 关系的数据竞争检测工具就是基于上述原理实现.如 RACEZ[30]以动态插桩的方式监控程序的运行时行为,并利用 HB 关系进行数据竞争检测,该方法采用采样机制降低跟踪同步和访存操作的运行时开销.Lockset 锁集分析是一个轻量级的用于并发错误检测的算法,它最早于 1997 年被用于一个动态的数据竞争检测工具 Eraser[31].Lockset 锁集算法的核心思想是跟踪程序执行过程中保护每一个访存位置的锁.锁集为空表明可能有两个或多个线程并发地访问了一个内存位置,即可能存在数据竞争.基于 HB 关系和基于 Lockset 算法的一个重要区别在于,前者可以用于分析除锁之外的其他同步机制之间的关系,而后者相对来说更加轻量级.虽然基本的 HB 关系算法和 Lockset 锁集算法可以帮助开发人员进行并发错误的检测,但考虑到具体的使用场景,研究人员还对这些算法进行了进一步的拓展和优化,使其更好地用于并发错误研究中.如 Robert O'Callahan 等人[32]提出将 HB 关系算法与 Lockset 锁集算法相结合进行并发错误检测,与单纯采用 Lockset 锁集算法相比有更低的误报率.鉴于传统的 Lockset 锁集算法受限于动态分析技术代码覆盖率较低,漏报率较高的问题,RELAY[33]和 Locksmith[34]方法将 Lockset 锁集算法用于静态代码分析技术中,进行并发错误检测,提高了并发错误检测效率.Whoop[35]则利用符号对 Lockset 锁集分析算法,检测设备驱动程序中的并发错误,使得 Lockset 锁集算法具有更好的可扩展性,检测到了更多的并发错误.

原子性违例的检测需要定位程序中的原子性区域,并对原子性区域的操作进行检查,进而判断这些操作是否违反原子性.研究人员常通过模式匹配的方法找到程序中的原子性区域,再利用动态分析、静态分析等方法检测程序中存在的原子性违例错误.如 CTrigger 通过动态控制线程调度,触发尽可能多的原子性违例错误[36].EPAJ 方法通过静态分析并发线程的所有可能执行序列,检查其是否符合预定义的原子性类型[37].

1.3 并发错误检测方法的评价指标

判断一个并发错误检测方法好坏的指标主要有:误报率、漏报率和检测速度.误报率是指检测到的非并发错误报告数占总的检测报告数的比率.误报率越低,表明检测效果越好.漏报率是指实际存在而没有被检测到的并发错误数占总的检测报告的比率.漏报率越低,检测效果越好.检测速度是指整个并发错误检测过程所花费的时间.花费的时间越短,检测方法更容易应用于实际的生

产过程之中.

研究人员在设计并发错误检测工具时,常常需要在误报率、漏报率和检测速度之间进行权衡.如采用静态分析方法往往能对代码进行较为全面的分析,具有较低的漏报率.但由于代码中存在的函数指针、宏定义等而无法进行精确分析导致误报率较高.很多研究人员提出通过过程间分析、流敏感分析、上下文分析和指向分析等方法降低并发错误的误报率,然而,这又增加了静态分析的时间和空间开销,检测速度会变慢.动态分析方法可以检测出实际运行过程中存在的并发错误,误报率较低.但由于很难探测到所有的执行路径,漏报率较高.另一方面,动态插桩、动态调度等方法的运行时开销较大.因此,研究人员一直在寻求一种低误报率、低漏报率,且运行速度快的并发错误检测方法.

2 现有并发错误检测方法局限性

作为软件系统领域的一个研究热点,并发错误检测方法的研究已经取得了很大进展.然而,很多研究工作针对应用程序级的并发错误,而由于操作系统内核本身的特殊性和复杂性,这些方法通常不能直接用于操作系统内核级的并发错误检测.另一方面,目前很多内核开发人员仍在广泛使用的内核级并发错误检测工具,如 Linux 内核死锁检测工具 Lockdep[38]、内存检查工具 KASAN[39]、模糊测试工具 Syzkaller[40]和数据竞争检测工具 KTSAN[41]等都存在很大的不足.因此,操作系统内核并发错误检测的研究仍面临巨大的挑战.

2.1 应用程序级并发错误检测方法局限性

操作系统内核作为软件栈最底层的部分,区别于上层的普通应用程序,其并发错误检测的难度也更大.很多可用于应用程序级并发错误的检测方法不能直接用于操作系统内核之中.主要原因有以下几个方面:

1) 操作系统内核代码规模大,复杂度高:以 Linux 内核代码为例,其代码量从最初发布时的几千行增加到现在的两千多万行.庞大的代码量会造成静态检测方法分析时间长,而动态检测方法 Trace 收集和分析的时间空间开销大大增加.另一方面,Linux 内核代码的结构相比于普通应用程序更为复杂,包含大量的宏定义和预处理指令,goto 语句及配置项繁多.如 Linux kernel v4.15 的代码函数达到 2500 多万行,而其中的 goto 语句就有超过 15 万行,宏定义数目更是多达 210 多万条.
2) 操作系统内核位置特殊性:操作系统内核作为应用程序与硬件设备交互的平台,处于软件栈最底层,很多可用于应用程序级动态二进制插桩的工具如 Pin[42]和 Valgrind[43]等无法对操作系统内核进行插桩和分析.而且,由于操作系统内核通常支持多种硬件架构,如 Linux 内核中支持的硬件架构包含 ARM,X86,Alpha 等超过 30 多种.因此,为了实现对不同架构下操作系统内核的插桩和动态监控,还需要虚拟化技术的支持.
3) 操作系统内核的同步原语复杂多样:以 Linux 内核为例,其中的同步原语除了各种锁机制(自旋锁、互斥锁、读写锁和顺序锁等),还包括内存屏障、原子操作和信号量等同步机制.此外,还有 Linux 内核特有的大内核锁(big kernel lock,简称 BKL)和读-复制-更新(read-copy-update,简称 RCU)等机制. BKL 可以锁定整个内核,确保没有 CPU 运行于内核态中.RCU 机制的设计主要针对指针类型的数据,它允许多个读者和写者访问临界区,提高了并发访问的效率.操作系统内核的很多同步原语,尤其是 RCU 和 BKL 机制,在应用程序中使用较少.因此,很多与这些同步原语相关的并发错误则无法被应用程序级的并发错误检测工具所识别.
4) 操作系统内核中断机制:中断是操作系统内核的特有构成部分,然而,中断机制的加入,也使操作系统内核中多线程并发执行过程变得更为复杂.一方面,由于中断处理程序的线程可能会与正常的内核线程交叉执行,从而造成并发错误;另一方面,中断相关的线程之间可能存在并发.因此,内核开发人员很难推断内核并发错误的产生是否与中断相关.
5) 线程调度带来的不确定性:在抢占式内核中,操作系统内核线程的调度可以分为抢占式调度和主动式调度.主动式调度中,线程的时间片结束或者线程需要等待资源就会自动挂起线程;而抢占式调度中,高优先级的线程会抢占正处于内核态执行的低优先级线程.因此,相比于应用程序,操作系统内核线程的调度的不确定性更增加了并发错误的复杂性.

2.2 操作系统内核中并发错误检测工具局限性

目前,在操作系统内核中集成了很多用于对内核进行动态调试以及并发错误检测的工具.例如,Linux 内核中的死锁检测工具 Lockdep[38]、Linux 内核内存错误检测工具 KASAN[39], Linux 内核模糊测试工具 Syzkaller[40],以及 Linux 内核数据竞争检查工具 KTSAN[41]和 KCSAN[44].这些工具被内核开发人员广泛使用,但仍存在很多的不足和缺点.

Lockdep 可以构建 Linux 内核锁类的统一抽象,并通过跟踪内核中锁的状态和不同锁之间的相互依赖关系,进行死锁检测.Lockdep 被证明可以达到几乎 100%的准确率. 然而,Lockdep 只能检测到 Linux 内核运行过程中发现的死锁问题,无法找出潜在的死锁问题,漏报率较高.我们之前针对 Linux 内核文件系统的死锁问题进行调研发现,Linux 内核中同步原语种类多样,除了 mutex_lock,spin_lock,down_write 等常见的同步函数,还包含各文件系统中为了同步而专门设计的同步函数,如 btrfs 文件系统中的 btrfs_tree_lock 函数用于实现对全局结构 extent_buffer 变量的同步访问.

KASAN 是 Linux 内核的一个内存错误检测工具.KASAN 利用影子内存机制实现编译时的内存检查,可以检测出堆栈溢出,全局变量访问越界等内存错误,如 UAF 错误和 out-of-bounds(OOB)错误等.KASAN 在 Linux 内核中易于使用,只需要在内核配置选

项中简单设置即可.KASAN 通过将代码编译为内联模式,大大提高了检测速度,但 KASAN 无法检测未初始化内存的错误.Kmemcheck 是一个与 KASAN 类似的 Linux 内核内存检测工具.Kmemcheck 可以跟踪内存的初始化、分配和释放过程,检测出未初始化或未分配内存,以及已释放内存的非法访问问题.但 Kmemcheck 无法检测出栈溢出或者全局变量访问越界等问题.而且,这些工具并不是针对操作系统内核并发错误检测而设计的.

Syzkaller 是 Google 公司开发的一款用于 Linux 内核模糊测试的工具.Syzkaller 通过输入一系列系统调用序列,并利用模拟器 QEMU 监控 Linux 的执行状态,并对模糊测试的覆盖面和测试结果予以反馈,以提高模糊测试的效率.利用 Syzkaller 进行并发错误检测常通过随机输入系统调用序列实现,因此,触发 Linux 内核并发错误的概率比较低,漏报率也比较高.

KTSAN 一款用于 Linux 内核数据竞争的检测工具. KTSAN 的实现基于动态二进制插桩工具 Valgrind,通过 Valgrind 获取内核执行过程中的访存和同步操作等信息,再利用 HB 关系分析内核运行过程中是否存在数据竞争. KTSAN 的不足之处在于它必须构建起所有同步机制的先后顺序,任何疏漏都将导致大量的误报.KCSAN 的实现则依赖于编译时插桩,并采用基于观察点采样的方法进行数据竞争检测.但 KTSAN 和 KCSAN 的检测结果都依赖于内核线程的动态执行过程,内核中没有执行的部分则无法进行检测.另外, KTSAN 也没有考虑内核的中断特性,漏报率较高.

尽管这些针对操作系统内核错误检测的工具得到内核开发人员的广泛使用,但这些工具本身存在漏报率和误报率较高,检测效果差等问题.而且针对操作系统内核并发错误的检测,开发人员往往需要结合多种工具的分析结果才能得出结论.因此,对操作系统内核并发错误的研究仍有待进一步提高.

3 操作系统内核并发错误检测研究

为了提高操作系统内核并发错误检测的效率,很多研究人员从并发错误检测方法、检测类型、针对的内核检测模块和检测效果上都进行了大量的研究,并取得了一定的研究成果.图 8 总结了最近几年研究人员针对操作系统内核并发错误检测的方法,这些方法包括形式化验证方法、静态检测方法、动态检测方法和静态动态相结合的检测方法.其中, 形式化验证方法包括定理证明和模型检验等,静态检测方法包括流分析、符号执行、代码注释等,动态检测包括动态二进制插桩、系统化测试等方法.为了提高检测效率,研究人员还提出了动态静态相结合的检测方法,如 Razzer[45]结合了指向分析和模糊测试技术,以检测操作系统内核的数据竞争错误.

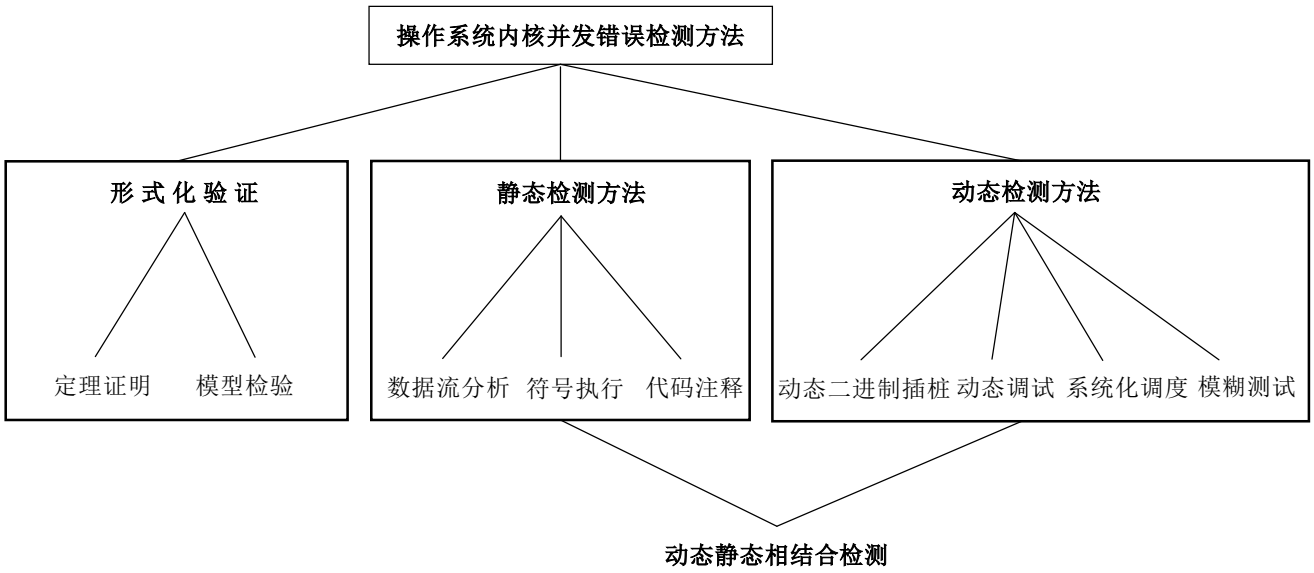

Fig.8 Approaches for concurrency bug detection in OS kernels

图 8 操作系统内核并发错误检测方法

静态检测是指在不运行程序的情况下,对程序的源代码、编译过程中生成的中间代码等进行分析,报告所有潜在的并发错误的检测技术.静态检测方法能覆盖到尽可能多的并发错误,但由于不能获取代码的动态运行时信息,可能造成较高的误报率.动态检测是指在程序运行的过程中进行并发错误检测的方法.一般情况下,动态检测方法只能检测出程序运行过程中暴露出来的并发错误,而无法检测未执行的代码路径中可能存在的并发错误.因此,动态检测方法的漏报率较高.为了提高并发错误检测的效率,很多研究人员提出了静态与动态相结合的并发错误检测方法.首先用静态检测方法定位到尽可能多的并发错误,再用动态检测方法进一步提升检测结果的准确性.

3.1 形式化验证方法

形式化验证作为形式化方法的一个重要应用,最近几年在软件程序的安全性验证方面取得了很大进展.基于形式化验证的操作系统内核并发错误的研究主要包含两个方面:基于定理证明的操作系统验证和基于模型检验的并发错误检测[46].

1) 基于定理证明的操作系统验证

基于定理证明的形式化验证方法通常利用从系统生成的一组规则并进行推理证明,以验证系统是否满足指定的规则.例如,Owicki 和 Gries 提出的并发程序验证方法[47],对单个线程的执行进行单独验证,以保证每个线程的执行过程不受其他线程的影响,并要求所有共享数据的访问都必须在互斥锁的保护下进行,确保并发访问的正确性.

研究人员还提出一些用于操作系统内核验证的方法.如 Jean Yang 等人[48]利用基于 Hoare 逻辑的自动定理证明器,并将 C#编写的内核代码编译成中间语言 Boogie[49]进行验证.华盛顿大学的 Luke Nelson 等人[50]设计的 Hyperkernel 是针对 Unix 内核的验证方法,该方法的实现基于 LLVM 中间代码,并利用 Z3 求解器进行验证,最终证明 Hyperkernel 可以实现一键完成验证,并能避免 bug.Yggdrasil[51]是一个自动验证文件系统的工具.该工具的设计基于一个称为"崩溃改进(crash refinement)"的新的文件系统正确性的定义,通过在文件系统实现过程中的一些硬盘状态(包括崩溃状态)来进行规则定义,从而使得开发人员实现一个可用于自动验证的文件系统.除此之外,还有一些操作系统内核验证的工作,包括 Fengwei Xu 等人[52]对抢占式内核 μC/OS-II 验证, Gerwin Klein 等人对微内核 seL4 的验证[53],以及 Ronghui Gu 等人对并发操作系统内核的验证 CertiKOS[42]等.

2) 基于模型检验的检测

模型检验是一种对有限状态的并发软件系统进行自动化验证的技术.该方法最早由 Clarke 和 Emerson,以及 Queille 和 Sifakis 于 20 世纪 80 年代分别开发出来[54].模型检验技术通过代码的简化描述方法,对所有可能的输入进行测试,以实现对程序的状态空间进行有效测试[55].基于模型检验的分析方法通常具有较高的准确率,但分析的开销也比较大.

MOKERT[56]是一个基于模型检验的内核测试框架.该框架可以在真实的内核代码中重复执行反例以帮助开发人员进行内核调试、错误检测以及补丁验证.相比于传统的手动构建测试模型的方法,MOKERT 框架设计了半自动化的模型构建方法,采用 Modex[57]从 C 代码中提取形式化模型,并将原始 C 代码与相应的 Promela 模型[58]进行映射,从而便于根据生成的反例对目标代码进行插桩.MOKERT 允许用户重复对一个反例进行回放,从而方便用户识别出并发错误发生的原因.该框架还支持模型的不断优化,如果在内核代码测试过程中发现一个反例是一个假反例,那么可以对模型进行不断的优化以保证模型的有效性.此外,该框架还可以帮助用户验证生成的补丁是否可以真正地消除内核中的错误.MOKERT 对 Linux 内核 2.6 中出现的两个并发错误进行了回放测试,以验证该框架的有效性,并检测出了 Linux 内核中的一个数据竞争错误.但模型检验方法的有效性取决于构建的测试模型,由于内核的线程交叉复杂,配置项繁多,很难构建一个精确的模型触发内核中的并发错误,例如 MOKERT 无法处理中断相关的并发错误,因此,该检测方法的漏报率较高.

基于形式化验证方法的并发错误检测目前也成为一个研究热点.如何对内核的特性进行抽取,构建更有利于操作系统内核并发验证的规则和模型是利用形式化验证实现操作系统内核并发错误检测的重要挑战.

3.2 静态检测方法

目前常用的静态并发错误检测方法包括基于流分析的检测、基于符号执行的检测和针对操作系统内核的特点而设计的检测方法等.主要通过构建并发错误触发的模式规范,并分析程序的控制流图或数据流图,以检查和分析程序是否满足给定的并发错误模型[34,59−61].

1) 基于数据流分析的检测

数据流分析最早由 Gary 和 Naval[62]提出,通过构建程序的控制流图(control flow graph,简称 CFG),对 CFG 图中的每条语句或者每个基本块进行计算和分析,求解相关的数据流方程,并进行反复迭代计算,在出口点求得最终结果.数据流分析最常见的应用包括可达定义分析和污点数据分析等.基于流分析的检测方法,通常采用编译技术获取源代码的 CFG 或者抽象语法树(abstract syntax tree,简称 AST)等,再利用数据流分析、上下文分析等技术,结合相应的并发错误检测算法进行检测.

RacerX[63]采用流敏感的过程间分析检测 Linux 内核和 FreeBSD 中的死锁和数据竞争错误.该方法首先针对程序中特有的锁操作(加锁、解锁以及是否允许或者禁止中断等)进行标注,然后提取程序的 CFG,并在 CFG 上运行死锁检测器和数据竞争检测器以检测出程序中存在的并发错误.RacerX 成功运行在 Linux kernel 2.5.62 版本(1.8MLOC)和一个代码行数将近 500K 的商业操作系统 System X 上,并最终检测出了 16 个并发错误.为了能进行大规模代码的分析,RacerX 没有采用指向分析等精确的分析技术,只进行了简单的数据流分析,并根据经验对分析结果进行排序,因此找到的并发错误数目较少.Jiaju Bai 等[64]采用混合的流分析方法检测并发错误中的 SAC 错误.SAC 错误通常出现在操作系统内核级别,在原子性上下文(如自旋锁加锁或者执行中断处理程序)中出现了睡眠操作而导致.论文称 SAC 错误在之前的研究中没有得到足够的重视,主要由于这类错误在实际的执行过程中很少发生,因此比较难以发现.事实上,由于 SAC 会引起 CPU 的阻塞,甚至导致系统挂起,因此,在原子性上下文中的睡眠操作是禁止的.然而,在操作系统内核中,SAC 错误仍然存在,尤其是在设备驱动程序和文件系统模块.主要有以下几方面的原因:(1)一个操作是否可以睡眠往往需要特定的系统知识或经验;(2)测试操作系统内核模块是比较困难的;(3)SAC 错误在每次实际执行过程中并不总是出现问题,而且它们通常比较难以复现.因此,作者提出 DSAC 方法,一个用于检测操作系统内核中的 SAC 错误的静态检测方法.该方

法首先采用过程间流敏感和流不敏感两者相结合的混合型流分析方法收集那些在原子性上下文中可能被调用的函数;然后通过启发式的方法,结合程序的控制流图和内核代码注释,提取到在程序运行时可能睡眠的函数接口;再利用混合数据流分析的方法检测可能存在的 SAC 错误,最后通过路径检查的方法过滤掉重复报告和误报的 SAC 错误.除此之外,论文还设计了一个基于模式匹配的自动化生成 SAC 错误修复补丁的工具.最终,该方法检测到了 401 个 SAC 错误,其中 272 个被开发人员确认.只有 26 误报,误报率为 6.3%.生成了 61 个补丁,其中 43 个被用在了内核代码中.DSAC 方法可以有效检测出 Linux 内核中的 SAC 错误,但仍有一些不足,如 DSAC 方法没有解决函数指针的问题,因此,与函数指针相关的很多 SAC 错误无法被检测到.另外,路径检查的方法并不能过滤掉所有的误报,还需要更有效的误报处理方法.

2) 基于符号执行的检测

符号执行最早由 King[65]等人提出,用于提高程序分析中的代码覆盖率,从而挖掘更多的错误.基于符号执行的静态分析方法,通过符号输入、路径探测和约束求解等方法尽可能扩大代码的检测范围.RELAY[33]设计出用符号执行和数据流分析相结合的方法检测大规模复杂代码(如 Linux 内核包含 450 万行代码)中的数据竞争问题.该方法提出相对锁集的分析方法,即一个锁集只描述相对于函数入口点开始的锁变化.相比于传统的锁集算法,该方法支持模块化的分析,使得函数的分析相对独立.而且该方法方便实现并行化分析,从而大大提高了分析的效率.例如,分析 Linux 内核代码的时间从没有并行化的 72 小时降低到 5 个小时.该方法首先利用符号执行的方法提取出函数中与调用上下文无关的函数行为的摘要信息,将函数内部本地变量的信息映射到全局形式化符号上,然后执行一个自底向上的基于数据流分析的相对锁集分析,最后基于从每个程序入口点生成的相对锁集信息,分析所有访存操作受保护的情况,生成数据竞争的检测报告.为了降低误报率,论文还采用了简单的语法过滤机制.例如,过滤掉那些不可能并发的函数报告.最终,该方法报告了 149 个数据竞争的警告,其中 53 个被确认为真正的数据竞争错误.然而在进行过滤之前,RELAY 的准确率只有 11%,误报率较高[20].

3) 基于操作系统内核特性的检测

基于操作系统内核中断、同步机制,以及并发接口设计等特性,研究人员也提出了一系列检测并发错误的方法.aComment[66]提出了中断相关的内核代码注释方法,用以检测内核中存在的并发错误.相比于传统的通过代码注释进行代码漏洞的检测方法,该方法重点检测操作系统内核中由于中断导致的并发错误.aComment 首先采用启发式的方法提取内核代码中所有关于与中断有关的注释,并对相应的函数进行注释的标记;根据已经收集的代码注释,通过函数调用图,采用自底向上的方法,推断并对所有函数进行注释标记.最后,根据函数的注释标记,如果该函数的中断前置条件和后置条件相互冲突,就报告该函数中可能存在错误.最终,该方法报告了 12 个并发错误,其中有 9 个是真正的并发错误.aComment 是一个专门针对操作系统内核中断相关的并发错误检测方法,可以在短时间内分析大量的代码,但 aComment 只能用于特定情况下的中断代码注释和分析,如只能分析无前置或者后置条件的中断,而且在中断提取阶段需要大量的人工分析和确认中断相关的注释,一旦代码注释发生改变或者失效,该方法则无法得出正确的分析结果. Peter T. Breuer 等[67]提出了一个针对 Linux 内核中 spinlock 死锁相关的并发错误检测的静态分析方法.该方法主要针对 Linux 内核中自 spinlock 加锁之后,由于线程调度而再次调用睡眠函数而导致的死锁问题.通过扫描源代码,设计针对程序语句的符号分析规则,检测出 Linux 内核中存在的死锁问题.论文中还给出了一个 Linux 内核网络适配器模块中检测出来的死锁错误.该方法只是一个简单的原型设计,并没有对实际的 Linux 内核代码进行检测,而且,该方法只能用于 spin_lock 相关的死锁检测,无法检测出其他类型的并发错误.Jiaju Bai 等[25]提出了一个静态的用于检测 Linux 驱动程序中的 CUAF 错误的方法 DCUAF.论文称经过统计分析,Linux 内核提交的用于修复 UAF 错误有 42%涉及到驱动程序的并发,而且几乎所有的错误是经过手动检查或者运行时发现的.因此,论文提出了一种有效的静态分析方法,可以覆盖到尽可能多的代码,检测出尽可能多的 CUAF 错误.根据 Linux 驱动程序模块的特点,所有的驱动程序代码实现都要继承统一的驱动程序接口,论文提出了从局部到全局的并发函数对提取的方法.首先通过局部分析得到本地并发函数对,然后通过全局分析得到整个驱动程序模块的并发函数接口对.再利用基于摘要的锁集算法检测 Linux 内核驱动程序中的 UAF 错误.最终发现了 640 个真正的 UAF 错误,作者随机选取了 130 个错误提交给了 Linux 内核开发人员,其中有 95 个得到最终确认.虽然 DCUAF 在检测驱动程序的 UAF 错误上取得了不错的效果,但仍有一些不足之处.如 DCUAF 无法处理中断相关的 UAF 错误,且只能检测到锁相关的 UAF 错误,因此漏报率较高.此外,DCUAF 没有处理函数指针的问题导致无法构建所有驱动程序代码的函数调用图造成一定程度的漏报.

4) 其他

其他的静态检测方法还包括基于模式匹配的检测方法和基于操作系统内存模型的检测方法等.文献[68]提出基于模式匹配的操作系统并发错误静态检测框架 COBET.该框架需要用户自己设定并发错误的模式,并采用语义分析引擎,通过调用路径分析、锁集分析以及别名分析等方法,帮助用户分析和检测出符合模式规则的并发错误.论文将该框架分别用于文件系统、设备驱动程序和网络模块的并发错误检测,并检测出了 10 个新的并发错误.该方法的不足在于用户需要根据代码的具体语义,同步机制等设计出检测模式,可扩展性较差.另外,用户只能检测出预定义匹配模式相关的并发错误,导致漏报率也较高.Pavel Andrianov 等[69]设计了一个轻量级的 Linux 内核数据竞争检测方法.该方法的实现基于可配置程序分析(CPA)的思想,因此称为 CPALockator.该方法采用了一个简单的内存模型,分析程序中的共享变量,得到一个共享变量访问的集合;然后,进行锁集分析、线程分析和预测分析得

到潜在的数据竞争报告,再利用优化分析的方法对每个潜在的数据竞争执行进行分析,排除掉那些不可达路径上的伪数据竞争.该方法被用于 Linux 内核 4.5-rc1 中的 drivers/net/wireless 模块,最终检测出 32 个数据竞争错误,其中有部分被开发人员确认.虽然该方法可以在分析效率和准确率之间做灵活调整,但限于内存模型的设计,无法处理 RCU 等内核同步机制相关的数据竞争,因此,漏报率和误报率仍较高.

综上,静态检测方法的优点是在不运行程序的程序的情况可以实现对代码进行分析和检查,同时可以尽可能覆盖到所有的程序调用路径,大大降低了漏报率.但采用静态分析的方法对操作系统内核进行并发错误检测仍面临以下挑战:

- 操作系统内核代码结构复杂,误报率较高.在操作系统内核代码中,各种宏定义、inline 函数、函数指针,以及汇编代码等层出不穷,使得传统的静态代码分析方法效果不佳.研究人员需通过结合流敏感分析、路径分析、符号执行,以及指向分析等多种静态分析技术才能降低并发检测的误报率.如 DSAC 方法采用的混合流分析方法,RELAY 中采用符号执行和数据流分析相结合的检测方法,都比单纯使用数据流分析方法的 RacerX 具有更好的检测效果.如何将传统的静态代码分析技术更好地用于操作系统内核将是一个重要挑战.
- 操作系统内核代码规模大,分析开销大.由于操作系统内核的代码量庞大,如 Linux 内核代码多达两千多万行,采用静态分析时花费的时间和空间开销都较大.如何降低静态分析的开销也将是研究人员关注的重点问题.RELAY 方法采用并行化的思想,使得静态分析的时间的时间从 72 小时缩短至 5 小时.
- 操作系统内核特有的并发错误研究有待加强.相比于应用程序,操作系统内核有其特殊性,如内核的中断特性可能导致中断处理线程与其他内核线程并发执行,进而引发更多的并发错误.aComment 方法基于代码注释的方法检测中断相关的并发错误,但人工分析的工作量较大.如何设计更有效的检测方法也是一个重要挑战.此外,操作系统内核特有的同步机制如 RCU、信号量、原子性操作等相关的并发错误也占有相当的比例,但相关的静态检测方法研究需要引起更多的关注.

3.3 动态检测方法

动态检测方法包括基于动态二进制插桩的检测、基于动态调试的检测、基于系统化调度的检测[71,72]和基于模糊测试的检测.通过跟踪操作系统内核的动态运行时行为或借助于动态调试技术、系统化调度和模糊测试等策略,检测内核运行过程中存在的并发错误.

1) 基于动态二进制插桩的检测

基于动态二进制插桩的检测方法常通过跟踪内核代码的执行过程,获取内核运行时 Trace,进而通过 HB 关系算法或者 Lockset 锁集算法进行在线或离线分析.

Redflag[72]就是一个基于动态二进制插桩进行内核级并发错误检测的工具,主要针对内核级的数据竞争和原子性违背错误.Redflag 通过动态二进制插桩的方法,记录内核代码执行的运行时信息,包括获取共享变量的访存操作、同步函数的使用、内存分配以及系统调用边界等,并将这些信息写入日志文件,再利用 Lockset 锁集分析算法和基于块的原子性违例分析算法进行数据竞争和原子性违例的分析.该方法还将内核中特有的同步机制,如 RCU 机制[73]以及 LOA 分析等手段,引入到这些分析算法中以提升检测的有效性.作者将 Redflag 用于 3 个内核组件:Btrfs、Wrapfs 和 Noveau,并分别检测出了数据竞争和原子性违例错误.为了降低二进制插桩的开销,Redflag 采用离线分析的方法进行并发错误检测.而且 Redflag 支持内核的特殊同步机制如 RCU、BKL 等,降低了误报率.但是 Redflag 只能检测到在内核运行过程中出现的并发错误,而无法检测到未执行代码中的并发错误,漏报率较高.DILP[74]是一个将动态二进制插桩用于 Linux 内核数据竞争检测更高效的研究方法.DILP 主要用于检测 Linux 内核驱动程序中由于不一致的锁保护而导致的数据竞争问题.论文发现在 Linux 内核驱动程序的修复补丁中,有 38%的数据竞争补丁涉及到不一致的锁保护问题.DILP 通过 LLVM[75]插桩的方法获得内核驱动程序的运行时踪迹,通过记录共享变量的访存信息、加锁解锁信息、以及并发的驱动函数信息,再利用锁集算法分析程序中存在的数据竞争问题.DILP 被用于 Linux kernel v3.3.1 和 v4.16.9 中进行并发错误检测,分别发现了 13 和 25 个数据竞争.最终,11 个被开发人员确认为真正的数据竞争.与类似工具 KernelStrider[76]相比,DILP 具有更好的检测效果. DILP 对并发的分析是函数级, 检测准确率不如更细粒度的指令级,但运行时开销却比指令级分析大大降低.但由于 DILP 是基于 Lockset 锁集的数据竞争检测,DILP 只能检测到锁相关的数据竞争而无法检测到内核中其他同步机制相关的数据竞争.此外,DILP 具有动态检测方法的缺点,只能检测到执行路径相关的数据竞争,代码覆盖率低.

不同于 Redflag 和 DILP, 为了帮助开发人员更好理解内核同步机制,进而避免并发错误的发生,LockDoc 提出用动态二进制分析的方法,通过分析内核同步机制生成内核同步机制使用规则并根据是否违反规则进行并发错误检测.LockDoc 的实现包含三个阶段:Trace 收集和分析阶段、锁规则生成阶段和文档生成及规则违背分析阶段.Trace 的收集采用动态二进制插桩方法,通过 Linux Test Project 中的测试用例获取访存信息.因此,LockDoc 的代码覆盖率取决于测试用例的有效性.锁规则生成是根据 Linux 内核源代码提取的同步信息,以便生成更多的锁规则.LockDoc 的一个不足之处在于其只关注单个变量相关的同步操作,而无法处理包含多个变量同步的并发错误.另外,LockDoc 只分析内核中的锁机制而不能检测其他同步机制导致的并发错误.

2) 基于动态调试的检测

基于动态调试的检测方法依赖于内核代码调试中的代码断点和数据断点技术.该方法需要借助于硬件调试技术,而且目前仅

仅基于 x86 架构有相关的实现.DataCollider[77]就是利用动态调试技术实现的动态数据竞争检测方法.相比于传统的数据竞争检测方法针对特定的同步协议(如锁机制等)保护的共享访存,该方法对于像底层内核代码这样采用多种复杂同步机制的程序代码更为重要.为了提高运行时效率,该方法采用随机采样内存的方法构建目标访存集合.利用硬件代码中断和数据中断检测 Windows 内核代码中存在的数据竞争错误.为了去除掉良性数据竞争,该方法通过检查是否共享变量的值一直增加,是否涉及到修改不同标志,或者其他的一些特殊变量等手段,降低 DataCollider 的误报率.最终,该方法检测出了 38 个数据竞争错误,其中 25 个被确认为真正的数据竞争错误,12 个已经被修复.DRDDR[78]采用与 DataCollider 类似的方法实现了 Linux 内核上的数据竞争检测,该方法首次将调试寄存器用于 Linux 内核的数据竞争检测之中,并成功复现了两个 Linux 内核中已有的数据竞争错误,发现了一些新的数据竞争错误.为了降低动态分析的运行时开销,这两种检测方法都采用了动态采样机制,因此,该方法的准确率都依赖于最初访存采样算法的有效性.

3) 基于系统化调度的检测

基于动态二进制插桩和基于动态调试的检测方法,很难检测到没有执行而潜在的并发错误.因此,研究人员提出了基于系统化调度的检测方法.该方法通过探究内核线程可能的交叉执行路径,检测到尽可能多的并发错误.Ben Blum 等[79]提出了系统化调度的方法来检测内核中存在的并发错误.通过确定性地执行每一种可能的线程交叉,识别出触发并发错误线程模式.Landslide 的实现基于仿真器 Simics[80],该仿真器是一个 x86 的全系统仿真器,从而实现对内核的动态监控.Landslide 的基本组件包含内核插桩模块、调度模块、内存跟踪模块、以及调度探索等.通过时钟中断的方式控制调度器的线程调度,并跟踪线程的生命周期(线程创建、睡眠和消亡等)和内存的读写操作,检查内核中是否发生 Panic,内存访问是否合法以及线程之间是否存在阻塞的环路等,找出内核中存在的并发错误.Landslide 的实现主要用于课程教学中的操作系统内核原型,并没有真正用于实际的操作系统内核如 Linux 内核等,但它提供了一种重要的用于动态检测内核并发错误的思路.SKI[81]的设计思路类似于 Landslide 提出的系统化探索方法,通过系统化地测试内核线程的交叉执行,动态地检测内核中的并发错误.相比于 Landslide,SKI 可以用于真实的操作系统内核中.SKI 的实现基于全系统仿真器 QEMU[82],可以实现在避免修改内核代码的情况下对内核进行调度控制.SKI 将内核线程与仿真器的虚拟 CPU 进行映射,并通过加入内核中断机制、扩展已有的 PCT 算法[27],从而实现对内核线程的调度.且利用内核的特殊指令,如 halt 指令、pause 指令、loop 指令等推断内核线程的生命周期.通过跟踪内核的运行时信息,并将访存信息、指令执行顺序,以及执行上下文信息记录到日志文件,再利用相应的 Bug 检测器检测出并发错误.SKI 用于 Linux 内核的文件系统中,并检测到 11 个并发错误,其中有 6 个已经被修复.SKI 还可用于内核并发错误的复现,通过与传统的压力测试方法相比较,SKI 可以大大降低复现并发错误的运行时开销.由于 SKI 的实现基于 VMM,操作系统内核运行于一个被虚拟化了的硬件环境中,因此,很多需要硬件支持的并发错误是无法检测和复现的.另外,由于 SKI 利用用户线程事件驱动线程调度以发现系统调用中存在的并发错误问题,忽略了很多其他内核线程并发导致的错误.对操作系统内核而言,还有大量的空间没有被探测到,从而造成漏报率较高.

4) 基于模糊测试的检测

模糊测试是一种常用的软件测试方法,主要通过生成随机的输入数据到程序中,以触发程序中可能存在的错误[83].研究人员将模糊测试和系统化调度方法相结合,提出了一些可用于并发错误检测的方法.MUZZ[84]采用基于覆盖率的插桩、线程上下文插桩和调度干预插桩三种新的多线程插桩方法以提高模糊测试过程中动态种子生成和执行的效率,检测多线程程序中存在的并发错误.ConFuzz[85]将静态代码分析技术与模糊测试技术相结合检测应用程序中存在的并发错误.除此之外,很多研究人员将模糊测试技术用于操作系统内核的并发错误检测. KRACE 用模糊测试技术检测内核文件系统中存在的数据竞争问题[86].KRACE 通过覆盖率导向的模糊测试方法对并发线程的空间进行探测,提出了生成、变换和合并多线程系统调用序列的演化算法进行模糊测试,大大提高了文件系统代码的空间探测范围和效率.最后,采用 Lockset 锁集和 HB 关系相结合的数据竞争检测算法对内核数据竞争进行精确检测.KRACE 最终在 ext4,btrfs 和 VFS 中发现了 23 个数据竞争,其中 9 个数据竞争已被确认为有害的数据竞争错误.然而,由于 KRACE 只能探测到运行时的内核线程空间,其他潜在的内核执行空间则无法探测.因此,存在误报率较高的问题.

综上,动态检测方法的优势在于可以明确程序的执行过程,包括具体的共享变量访问情况,以及具体的函数调用路径等,可以帮助开发人员准确定位并发错误发生的位置及过程.然而,采用动态检测方法对操作系统内核的并发错误检测还面临如下挑战:

- 操作系统内核线程交叉空间难以探测,漏报率高.动态分析很难遍历到程序执行的所有路径,因此,动态检测方法无法检测到系统中存在但没有触发的并发错误.研究人员往往需要不断调整配置和输入参数,才能探测到尽可能多的路径,触发潜在的并发错误.相比于应用程序而言,操作系统内核执行过程中的线程数量多达几十甚至几百.尽管研究人员采取的压力测试和系统化调度的方法可以探测到更多的线程执行路径,可以从一定程度上降低误报率.如 SKI 方法和 Landslide 都从一定程度上降低了漏报率.如何探测所有线程的交叉执行过程仍是一个巨大的挑战.
- 操作系统内核运行时分析开销大.不管是动态插桩、动态调试方法、系统化调度方法还是模糊测试方法,由于操作系统内核代码复杂而造成运行时时间和空间开销很大.研究人员采用采样、离线分析都手段降低运行时开销.如 DataCollider 和 DRDDR 通过随机采样方法降低动态插桩的开销.

3.4 静态与动态相结合的检测方法

为了提高并发错误的检测效率,研究人员通过静态分析和动态分析相结合的方法进行并发错误检测.通过静态分析覆盖到尽可能多的并发错误,动态分析降低误报率,从而有效地检测出更多的并发错误.

Razzer[45]就是这样一个动态和静态相结合的数据竞争检测工具.Razzer 首先利用静态分析的方法获得潜在的数据竞争访存对,然后利用动态的模糊测试方法对这些访存对进行测试,从而检测出数据竞争.为了获得潜在的数据竞争并发访存对,Razzer 利用基于 LLVM 的静态分析方法,包括控制流分析,数据流分析以及指向分析等方法找出内核代码中可能的并发访存对.然后,基于 QEMU 和 KVM 实现了一个用于监控内核运行时行为,确定性地控制内核线程的调度管理程序.通过确定性的内核线程交叉执行,判断给定的并发访存对是否造成数据竞争.Razzer 用于 Linux kernel v4.16.-rc3 和 v4.18-rc3 并检测出了 30 个有害的数据竞争问题.论文还将 Razzer 与 SKI、Syzkaller 做比较,发现 Razzer 的检测准确性和运行时效率都比其他工具高.相比于 SKI 的随机线程调度,Razzer 需要更少的运行次数就可以触发数据竞争错误.由于 Razzer 依赖于静态分析的结果,静态分析过程可能会漏掉部分并发访存对而造成漏报. 目前,静态与动态分析相结合的方法是一种效率比较高的并发错误检测方法.静态分析作为前提,可以帮助动态分析降低运行时开销,更高效地触发并发错误.动态分析可以作为静态分析的补充,进一步确认静态分析的检测结果,降低静态分析的误报率.因此,静态和动态分析相结合的并发错误检测方法是一种比较有前景的并发错误检测方法.

3.5 总结与对比分析

对操作系统并发错误检测方法的评估主要包括两个方面:一是针对检测效果的评价,包括误报率、漏报率和准确率等;而是针对检测效率的评价,包括检测速度、运行时开销等.表 1 中选取了针对操作系统内核并发错误检测的 17 个代表性研究工作,并对其采用的技术手段、检测类型、检测效果和性能进行了对比分析.第 1 列代表该项研究工作所采用的检测方法类别,包含静态检测、动态检测、静态与动态相结合的检测和形式化验证四大类.第 2 列是每一项研究工作中具体采用的检测技术手段,包含数据流分析、符号执行、动态二进制插桩、系统化调度以及模型检验等方法.第 3 列为具体的研究方法名称.第 4 列给出了各项研究工作中针对的并发错误类型,如数据竞争、死锁、UAF、SAC 等.没有明确说明或检测结果中包含多种类型的方法即标记为所有并发错误类型.第 5 列为每项研究工作针对的操作系统内核版本和模块.第 6 列代表该研究方法采用的是 HB 关系算法还是 Lockset 锁集算法进行并发错误检测,没有使用这两种算法则不标记.第 7 列为该方法检测到的并发错误数目.第 8 列为该方法进行并发错误检测的检测速度以时间开销计算,时间开销越大,表明检测速度越慢.从表 1 中可以看出:

Table 1 Comparison of concurrency bug detection for OS kernels

表 1 操作系统内核并发错误检测方法对比分析

类别	名称	技术手段	错误类型	内核版本及模块	检测算法	检测结果	检测速度
静态检测	RacerX[63]	数据流分析	并发错误	Linux kernel v2.5.62	Lokset 锁集	18 个死锁,13 个数据竞争,其中 12 个误报	2-14 分/1.8MLOC
	DSAC[64]	混合数据流分析	并发 SAC 错误	Linux kernel v3.17.2 Linux kernel v4.11.1 驱动程序、文件系统、网络模块	-	Linux kernel 3.17.2 中检测到 215 个并发 SAC 错误,15 个误报; Linux kernel 4.11.1 中发现 340 个 SAC 错误,20 个误报.	108 分/7.3MLOC - 129 分/9.4MLOC
	RELAY[31]	符号执行、数据流分析	数据竞争	Linux kernel v2.6.15	相对 Lockset 锁集	53 个数据竞争	单机:72 小时/4.5MLOC 并行化后:5 小时/4.5MLOC
	WHOOP[35]	符号执行	并发错误	Linux kernel 4.0,驱动程序	符号对 Lockset 锁集	发现 45 个并发错误	144.5 秒/7.2KLOC
	aComment[66]	代码注释	并发错误	Linux kernel	-	12 个并发错误,其中 3 个误报	整个 Linux kernel 需要 265 分钟
	COBET[68]	模式匹配	并发错误	Linux kernel 2.6.30.4 文件系统、驱动程序、网络模块	-	10 个新的并发错误	48.85 秒/100KLOC
	DCUAF[25]	基于内核代码特性	并发 UAF 错误	Linux kernel v3.14 Linux kernel v4.19 驱动程序	基于摘要的 Locset 锁集	Linux kernel 3.14 中发现 559 个错误,33 个误报; Linux kernel 4.19 中发现 679 个 UAF 错误,39 个误报.	23 分/11.2KLOC 28 分/16.6KLOC
动态检测	DataCollider[78]	动态调试	数据竞争	Windows 7 kernel	-	发现 25 个数据竞争,其中 12 个已被修复	小于 1.3x 的开销
	DRDDR[78]		数据竞争	Linux kernel 文件系统	-	发现 2 个良性数据竞争	小于 1.1x 的开销
	Redflag[72]	动态二进制插桩	数据竞争 原子性违	Linux kernel 文件系统、驱动程	Lockset 锁集	在 Wrapfs 中发现 49 个数据竞争,其中 2 个	2.65x 的插桩和日志记录开销

			例	序		已被确认,45 个是良性数据竞争, 2 个是误报; 在 Btrfs 中发现 8 个良性数据竞争; 在 Noveau 中发现 11 个误报的数据竞争.	
	DILP[74]		数据竞争	Linux kernel v3.3.1 Linux v4.16.9 驱动程序	Lockset 锁集	在 Linux kernel 3.3.1 中发现 13 个数据竞争; 在 Linux kernel 4.16.9 中发现 25 个数据竞争.	平均 7.2x 的开销
	LockDoc[87]		并发错误	Linux kernel	-		291 分/5.7MLOC
	SKI[81]	系统化调度	并发错误	Linux kernel v2.6.28 – v3.13.5 文件系统	-	发现 190 个并发错误, 其中 76 个误报,53 个良性数据竞争,24 个恶性数据竞争.	-
	Landslide[79]		数据竞争	Pebbles kernel	-	用于课程教学,未发现真正内核数据竞争	-
	KRACE[86]	模糊测试、动态插桩	数据竞争	Linux kernel v5.4-rc5 文件系统 btrfs,ext4,VFS	Lockset 锁集和 HB 关系	发现了 23 个数据竞争, 其中 9 个恶性数据竞争,11 个良性数据竞争	7 天的模糊测试,96 小时的数据竞争检测
动静结合	Razzer[45]	指向分析、模糊测试	数据竞争	Linux kernel v4.16-rc3 -v4.18rc3	-	发现了 30 个恶性数据竞争,其中 16 个被开发人员确认.	整个 Linux kernel 需要 7 天
形式化验证	CertiKOS[88]	定理证明	并发错误	CertiKOS	-	-	-
	MOKERT[89]	模型检验	并发错误	Linux kernel v2.6 文件系统	-	发现了 1 个数据竞争	-

1) 从并发错误检测手段上,数据流分析、符号执行等技术仍是静态检测方法的重要手段,而研究人员也尝试将不同的技术手段结合起来提高检测的效率.如 RELAY 采用符号执行和数据流分析相结合的检测方法,DSAC 则采用流敏感分析和流不敏感分析相结合的混合数据流检测方法.其次,由于静态检测方法和动态检测方法各有优缺点,Razzer 通过静态检测与动态检测相结合的检测方法,以达到更好的检测效果.此外,形式化验证方法的模型检验[56]和操作系统内核验证[88]也为内核并发错误检测提供了新的思路.

2) 从并发错误类型上,大量(8/18)内核并发错误研究工作将数据竞争作为重点检测对象,如 RELAY 方法、DataCollider 方法和 DILP 方法等.这可能是由于一方面数据竞争在内核中占的比重较大,是 Linux 内核中除死锁外,发生最多的并发错误(如图 7 所示);另一方面研究人员对应用程序级数据竞争的研究已经取得了很大进展,数据竞争的定义和触发机制清晰,相比于其他并发错误的研究更容易开展.近年来,也有很多研究人员对原子性违例进行了研究,如 Shan Lu 等人提出的 AVIO[89]和 Ctrigger[36]方法用于应用程序级的原子性违例检测.另外,针对内核的并发错误特点,研究人员还关注了一些特殊类型的并发错误,如 DCUAF 方法用于检测 Linux 内核驱动程序中的 CUAF 错误,DSAC 方法用于 SAC 错误的检测.

3) 从检测算法上,很多研究工作(6/18)采用 Lockset 锁集相关的算法进行并发错误检测,一方面是由于 Lockset 锁集算法相比于 HB 关系算法更加轻量级,且易于实现;另一方面,由于内核中的线程并发和同步更加多样和复杂,很难对多线程并发和同步进行建模并和分析,例如设备中断可能随时打断线程的执行,DMA 的执行不在处理器内核上等因素使得 HB 关系算法用于操作系统内核的并发错误检测更加困难.然而,传统的基于 HB 和 Lockset 锁集的算法已不能满足研究的需要,研究人员提出了很多改进的算法,如相对 Lockset 锁集[33]和符号对 Lockset 锁集[35]等算法,以提高并发错误的检测效率.此外,还有很有研究人员采用 HB 关系和 Lockset 相结合的方法进行并发错误检测[32].近年来,很多动态检测方法如 DataCollider, DRDDR 和 SKI 等方法则避免使用 HB 关系和 Lockset 锁集进行并发错误检测,也取得了较好的检测效果.

4) 从涉及到的内核模块上,由于内核代码规模庞大且复杂,不管是静态检测方法还是动态检测方法,对整个内核的并发错误检测的分析开销都较大,实现起来较为困难.另外,操作系统内核各个模块运行机制和实现技术各不相同,难以用同样的技术方案对内核的不同模块进行处理,因此,研究人员会选择从并发错误发生较多的操作系统模块着手进行研究,如 DCUAF 方法利用了驱动程序接口的并发特性而实现了专门针对 Linux 内核驱动程序并发错误检测的方法,而 SKI 则针对 Linux 内核中文件系统模块进行数据竞争检测.从表 1 中可以看出,针对内核中的驱动程序和文件系统两个模块进行并发错误研究的工作较多.

5) 从检测效果上,静态检测方法能够检测出的并发错误数少则 9 个(aComment),多则 640 个(DCUAF),而动态检测方法则能检测到 1 至 190 个并发错误.从整体上,静态检测方法检测到的并发错误数目要比动态检测方法多.然而,静态检测方法的误报率较高,RacerX 有将近 40%的误报率,因此研究人员采用静态检测方法时主要考虑如何降低误报率.例如,RacerX 通过裁剪代码路径,移除不必要的锁集分析等方法降低死锁检测的误报率.动态检测方法虽然误报率较低,但漏报率很高.如 DRDDR 方法只能检测到 2 个良性的数据竞争.研究人员则通过系统化调度[81]、动态与静态相结合[45]等方法降低动态检测的漏报率.不管是静态检测方法还是动态检测方法,将其用于操作系统内核这种大规模代码中,都面临检测开销大的问题.为了追求更低的误报率和漏报率,研究人员需要进行更为精确的分析,而检测的开销也随之增加.如 Razzer 方法甚至需要花费 7 天的时间才能完成对整个 Linux 内核代码的检测.

从这些代表性的操作系统内核并发错误检测方法的对比和分析中,我们可以发现:继应用程序级并发错误的研究之后,操作系统内核并发错误的研究已成为近年的研究热点.研究人员从操作系统内核各种并发错误的类型、产生原因,以及内核本身的特性出发,并利用静态分析、动态分析和形式化验证等多种方法进行研究,并取得了一系列的研究成果,主要表现在:

1) 并发错误检测效果更好:从 2003 年的 RacerX 到 2019 年的 DSAC,并发错误检测的漏报率和误报率都在降低.从动态二进制插桩技术到系统化调度分析,并发错误检测的准确率变得更高.
2) 并发错误检测效率更高:从传统的 Lockset 锁集方法到基于摘要的 Lockset 锁集,分析的速度在加快;基于采样的动态分析相比于普通动态插桩方法大大降低了运行时开销.基于系统化调度和模糊测试的技术相比于随机调度方法也更加速了并发错误的触发.
3) 更加注重对操作系统内核本身特性的研究:研究人员更加注重解决针对操作系统内核特性导致的并发错误,如 aComment 提出针对中断的并发错误处理方法,SKI 中也着重考虑了内核中断的处理.
4) 引入其他缺陷检测方法用于并发错误检测:研究人员从软件工程中其他软件缺陷检测的方法中获得启发,将其用于并发错误检测,取得了较好的效果.如 Razzer 和 KRACE 都是基于模糊测试技术,并将其用于操作系统内核的数据竞争检测.

4 研究挑战与展望

4.1 研究挑战

基于现有的操作系统内核并发错误检测方法,我们认为,针对操作系统内核的并发错误研究,目前还面临如下挑战:

1) 操作系统内核并发错误检测的特殊性

由于操作系统内核处于硬件之上,软件栈最底层.内核线程的交叉执行分析往往需要借助于虚拟化技术 VMM 的支持.如 SKI[81]、RAZZER[45]和 KRACE[86]等研究中利用 QEMU 实现内核线程的系统化调度.然而,基于 QEMU 等虚拟化的方法只能复现那些可以被 QEMU 支持的硬件相关的并发错误,对于 QEMU 不支持的硬件相关的并发错误则无法被检测.其次,VMM 每次模拟一条指令的执行,随后将其结果立即传递给其他 CPU,而现有的 CPU 大多采用的是弱内存模型,那些误认为是强内存模型的并发错误则无法被暴露出来.此外,没有硬件支持的虚拟化会导致 Guest 系统运行效率大幅降低,从而导致内核中与时间(延迟)相关的默认配置不再适用,例如与 RCU 相关的 CONFIG_RCU_CPU_STALL_TIMEOUT 等参数需要重新配置.

操作系统运行过程中既有内核线程也有用户线程,区分不同内核线程的上下文也是一个重要挑战.这是因为内核运行时存在大量其他的线程,且部分线程是内核固有的,没有对应的用户进程.基于模糊测试的检测方法利用外部系统调用触发内核内部代码执行,因此需要识别与用户进程对应的内核线程,以便于调度控制和并发分析.此外,由于时间片的原因,每隔固定时间内核会发生时钟中断,触发调度算法运行并引发大量的同步操作,由此引入了大量的加锁与解锁操作，区分内核线程上下文能有效排除这部分干扰.

除了以上问题外,还存在基于模糊测试的检测方法无法控制和分析内核内部固有的线程,基于 Lockset 锁集算法的检测方法无法分析操作系统内核特有的 lock-free 同步机制(如 RCU 等)问题.

2) 操作系统内核新的并发错误类型检测

近年来,除了针对基本的并发错误,如死锁、数据竞争、原子性违例等之外,还有很多研究人员针对 CUAF、SAC 等其他类型的操作系统内核的并发错误进行检测.因此,如何从操作系统内核中发现新的并发错误类型,并提出针对此种类型的并发错误检测方法会是一项重要的挑战.

3) 检测方法的可扩展性

很多现有的操作系统内核并发错误检测方法都是针对特定的操作系统内核类型和特定的硬件架构实现的.如 DataCollider 和 DRDDR 都是基于 X86 架构的实现,而不能直接用于 ARM、Alpha 等其他架构下的并发错误检测.另外,很多并发错误检测方法都针对特定的操作系统,如 DataCollider 是基于 Windows 内核的数据竞争检测,而 DRDDR 是针对 Linux 内核的数据竞争检测.此外,操作系统内核有文件系统、驱动程序、内存管理等多个模块,不同模块的代码结构设计和运行时机制差异较大.很多针对特定内核模块的方法很难用于其他模块乃至整个内核的并发错误检测.如 DCUAF 方法是依据 Linux 内核驱动程序接口的并发特性而设

计的检测方法,是否能用于其他模块的 CUAF 检测还需要进一步研究.

4) 并发错误检测效果与速度之间的权衡

由于操作系统内核本身代码规模大,结构复杂.而更精确的并发错误检测方法往往带来更大的检测开销.如 Razzer 的检测效果虽好,但长达 7 天的分析时间开销在实际生产中是难以接受的.如何能用更低的分析开销获得更好的检测效果,是研究人员设计并发错误检测方法时必须考虑的问题.

4.2 未来研究趋势

基于目前已有的研究工作,操作系统内核并发错误检测的研究还需要从以下几个方面突破:

1) 结合操作系统内核的特性

内核代码复杂性高,且有其自身的特点.但以往的研究工作更注重从并发错的角度设计检测方案,这样的方法具有通用性,但针对内核的并发错误检测效果还需要提升.因此,如何从内核本身的特性入手研究针对内核并发错误的检测方法是一个值得深入研究的问题,如 DCUAF 针对 Linux 内核的并发接口模型而设计.

2) 发掘新的内核并发错误类型

内核本身的复杂性导致内核中出现的并发错误也多种多样,因此,为了提出更有效的内核并发错误检测方案,需要对内核中的并发错误进行仔细的研究和分类.除了传统的并发错误类型,内核可能还有其他重要的并发错误.研究人员可通过发现新的并发错误类型,提出更有效的内核并发错误检测方法.如 Jiaju Bai 等人提出的 DSAC 和 DCUAF 就是针对新的内核并发错误类型.

3) 静态与动态相结合的并发错误检测

在设计内核并发错误检测方案时,低开销、低误报率和漏报率是衡量内核并发错误检测方法的重要指标.静态与动态相结合的并发错误检测方法是目前提高并发错误检测有效性的有效方法之一,代表的工作如将指向分析等技术与模糊测试技术结合的并发错误检测方法 Razzer.因此.如何结合静态和动态分析方法,以及选择哪种静态和动态方法结合是一个值得关注的研究问题.

4) 基于形式化验证方法的并发错误检测

近年来形式化方法的研究取得了长足的发展,而采用形式化验证方法进行并发错误的研究也有了很多尝试.如基于定理证明的 CertiKOS 内核验证和基于模型检验的并发错检测 MOKERT 方法.因此,如何采用更好的形式化验证方法实现操作系统内核的并发错误检测也是一个重要的研究趋势.

5) 其他操作系统内核的并发错误检测

目前操作系统内核并发错误的研究大多针对 Linux 内核、Windows 内核,而对其他的操作系统内核如 Android 内核、MacOS 内核等,包括近几年很多研究人员提出采用 Rust 语言实现比 C/C++更高效的操作系统 RustOS 等内核的并发错误检测研究是一个值得探讨的问题.

5 结语

并发错误,特别是操作系统内核的并发错误检测,目前已经成为软件安全领域一个重要的研究课题.经过多年的发展,操作系统内核并发错误检测的方法虽然从检测效果和性能上都取得了显著的进步,但如何降低误报率和漏报率,降低并发错误检测的开销,仍是需要进一步深入研究的问题.

本文从操作系统内核并发错误的基本类型入手,总结了操作系统内核并发检测技术的最新研究进展.从静态分析、动态分析、静态与动态分析相结合的分析,以及形式化验证等角度总结和比较了各类检测技术的检测效果.相比于普通应用程序上的并发错误检测,针对操作系统内核的并发错误检测还面临着很多挑战,未来也需要更加有效的检测工具以帮助提高内核的安全可靠性.

References:

[1] Leveson NG, Turner CS. An investigation of the therac-25 accidents. Computer, 1993, 26(7):18–41.

[2] Zhang W, Lim J, Olichandran R, Scherpelz J, Jin G, Lu S, Reps T. ConSeq: detecting concurrency bugs through sequential errors. ACM SIGARCH Computer Architecture News, 2011, 39(1):251--264.

[3] Nasdaq's facebook glitch came from race conditions. 2020. https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html.

[4] Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L. How do fixes become bugs? In: Proc. of the Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). 2011. 26–36. [doi: https://doi.org/10.1145/2025113.2025121]

[5] Naik M, Aiken A, Whaley J. Effective static race detection for java. In: Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2006. 308-319. [doi: https://doi.org/10.1145/1133981.1134018]

[6] Cai Y, Zhang J, Cao L, Liu J. A deployable sampling strategy for data race detection. In: Proc. of the 24th ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE). Seattle, WA, USA: ACM Press, 2016. 810–821.[doi: https://doi.org/10.1145/2950290.2950310]

[7] Fonseca P, Li C, Rodrigues R. Finding complex concurrency bugs in large multi-threaded applications. In: Proc. of the sixth Conf. on Computer systems (EuroSys). Salzburg, Austria: ACM Press, 2011. 215-228. [doi: https://doi.org/10.1145/1966445.1966465]

[8] Weeratunge D, Zhang X, Sumner WN, Jagannathan S. Analyzing concurrency bugs using dual slicing. In: Proc. ofthe 19th Symp. on Software testing and analysis (ISSTA). Trento, Italy: ACM Press, 2010. 253-264. [doi: https://doi.org/10.1145/1831708.1831740]

[9] Zhang T, Jung C, Lee D. ProRace: practical data race detection for production use. ACM SIGPLAN Notices. 2017, 45(2):149-162. [doi: https://doi.org/10.1145/3093336.3037708].

[10] Liu H, Li G, Lukman JF, Li J, Lu S, Gunawi HS, Tian C. DCatch: automatically detecting distributed concurrency bugs in cloud systems. ACM SIGPLAN Notices. 2017, 45(1):677-691. [doi: https://doi.org/10.1145/3093337.3037735].

[11] Bond MD, Coons KE, McKinley KS. PACER: proportional detection of data races. ACM Sigplan Notices, 2010, 45(6): 255-268. [doi: https://doi.org/10.1145/1809028.1806626]

[12] Shan Lu, Soyeon Park, Yuanyuan Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. IEEE Trans.on Parallel and Distributed Systems, 2012, 23(6): 1060–1072. [doi: https://doi.org/10.1109/TPDS.2011.254]

[13] Lu S, Park S, Zhou Y. Finding atomicity-violation bugs through unserializable interleaving testing. IEEE Trans. on Software Engineering, 2012, 38(4): 844–860. [doi: https://doi.org/10.1109/TSE.2011.35]

[14] Flanagan C, Freund SN. FastTrack: efficient and precise dynamic race detection. ACM Sigplan Notices, 2009, 44(6): 121-133. [doi: https://doi.org/10.1145/1543135.1542490]

[15] Marino D, Musuvathi M, Narayanasamy S. LiteRace: effective sampling for lightweight data-race detection. Acm Sigplan Notices, 2009, 44(6):134-143.. [doi: https://doi.org/10.1145/1542476.1542491]

[16] Singhal M. Deadlock detection in distributed systems. Computer, 1989, 22(11):37-48. [doi: https://doi.org/10.1109/ 10.1109/2.43525].

[17] Serebryany K, Iskhodzhanov T. ThreadSanitizer: data race detection in practice. In: Proc. of the Workshop on Binary Instrumentation and Applications (WBIA). New York, New York: ACM Press, 2009. 62-71. [doi: https://doi.org/10.1145/1791194.1791203]

[18] Lucia B, Devietti J, Strauss K, Ceze L. Atom-aid: detecting and surviving atomicity violations. In: Proc. of the 2008 Symp. on Computer Architecture (ISCA). Beijing, China: IEEE, 2008. 277–288. [doi: https://doi.org/ 10.1109/ISCA.2008.4]

[19] Lu S, Park S, Seo E, Zhou Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In Proc. of the 13th Conf. on Architectural support for programming languages and operating systems. 2008. 329-339. [doi: https://doi.org/10.1145/1346281.1346323].

[20] Su X, Yu Z, Wang T, Ma P. A survey on exposing,detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015(11): 2215–2233 (in Chinese with English abstract). [doi: https://doi.org/ 10.11897/SP.J.1016.2015.02215]

[21] Atomic context and kernel api design. 2020. https://lwn.net/Articles/274695/.

[22] Fonseca P, Cheng Li, Singhal V, Rodrigues R. A study of the internal and external effects of concurrency bugs. In: Proc. of the 2010 IEEE/IFIP Conf. on Dependable Systems & Networks (DSN). Chicago, IL: IEEE, 2010. 221–230. [doi: https://doi.org/ 10.11897/ 10.1109/DSN.2010.5544315]

[23] Abbaspour Asadollah S, Sundmark D, Eldh S, Hansson H. Concurrency bugs in open source software: a case study. Journal of Internet Services and Applications, 2017, 8(1):1-15-.

[24] Shi JJ, Ji WX, Wang YZ, Huang LF, Guo YK, Shi F. Linux kernel data races in recent 5 years. Chinese Journal of Electronics, 2018, 27(3): 556–560. [doi: https://doi.org/ 10.1049/cje.2018.03.015]

[25] Bai JJ, Lawall J, Chen QL, Hu SM. DCUAF:effective static analysis of concurrency use-after-free bugs in linux device drivers. In Proc. of the 2019 USENIX Annual Technical Conference (ATC). 2019. 255-268.

[26] Williams A, Thies W, Ernst MD. Static deadlock detection for java libraries. Black A P. In: Proc. of European Conf. on Object-Oriented Programming (ECOOP). . Springer, Berlin, Heidelberg. 2005. 602–629. [doi: https://doi.org/10.1145/1735970.1736040]

[27] Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGARCH Computer Architecture News, 2010, 38(1): 167–178. [doi: https://doi.org/10.1145/1735970.1736040]

[28] Cai Y, Chan WK. MagicFuzzer: scalable deadlock detection for large-scale applications. In: Proc. of the 34th Conf. on Software Engineering (ICSE). 2012. 606-616. [doi: https://doi.org/10.1109/ICSE.2012.6227156]

[29] Lamport L. Time, clocks, and the ordering of events in a distributed system. 1978, 21(7): 8. [doi: https://doi.org/10.1145/3335772.3335934]

[30] Sheng T, Vachharajani N, Eranian S, Hundt R, Chen W, Zheng W. RACEZ: a lightweight and non-invasive race detection tool for production applications. In: Proc. of the 33rd Conf. on Software Engineering (ICSE). Waikiki, Honolulu, HI, USA: ACM Press, 2011. 401-410. [doi: https://doi.org/10.1145/1985793.1985848]

[31] Savage S. Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems. 1997, 15(4): 391-411. [doi: https://doi.org/10.1145/265924.265927]

[32] O'Callahan R, Choi JD. Hybrid dynamic data race detection. In: Proc. of the ninth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). New York, NY, USA: Association for Computing Machinery, 2003. 167–178. [doi: https://doi.org/10.1145/781498.781528]

[33] Voung JW, Jhala R, Lerner S. RELAY: static race detection on millions of lines of code. In: Proc. of the 6th Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). Dubrovnik, Croatia: ACM Press, 2007. 205-214. [doi: https://doi.org/10.1145/1287624.1287654]

[34] Pratikakis P, Foster JS, Hicks M. Locksmith: context-sensitive correlation analysis for race detection. Acm Sigplan Notices, 2006, 41(6): 320-331.. [doi: https://doi.org/10.1145/1133255.1134019]

[35] Deligiannis P, Donaldson AF, Rakamaric Z. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In: Proc. of the 30th IEEE/ACM Conf. on Automated Software Engineering (ASE). Lincoln, NE, USA: IEEE, 2015. 166–177. [doi: https://doi.org/ 10.1109/ASE.2015.30]

[36] Park S, Lu S, Zhou Y. CTrigger: exposing atomicity violation bugs from their hiding places. In: Proc. of the 14th Conf. on Architectural Support for Programming Languages and Operating Systems. 2009. 25-36. [doi: https://doi.org/10.1145/1508244.1508249].

[37] Sasturkar A, Agarwal R, Wang L, Stoller SD. Automated type-based analysis of data races and atomicity. In: Proc. of the tenth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). Chicago, IL, USA: ACM Press, 2005. 83-94. [doi: https://doi.org/10.1145/1065944.1065956]

[38] Lockdep. 2020. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[39] KASAN. 2020. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[40] Syzkaller. 2020. https://github.com/google/syzkaller.

[41] KTSAN. 2020. https://github.com/google/ktsan.

[42] Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 2005, 40(6): 190-200.. [doi: https://doi.org/10.1145/1064978.1065034]

[43] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 2007, 42(6): 89-100.. [doi: https://doi.org/10.1145/1273442.1250746]

[44] KCSAN. 2020. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[45] Jeong DR, Kim K, Shivakumar B, Lee B, Shin I. Razzer: finding kernel race bugs through fuzzing. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco, CA, USA: IEEE, 2019. 754–768. [doi: https://doi.org/ 10.1109/SP.2019.00017]

[46] Wang J, Zhan N, Feng X, Liu Z. Overview of formal methods. Ruan Jian Xue Bao/ Journal of Software, 2019, 30(1):33–61 (in Chinese with English abstract). http://www.jos.org.cn/html/2019/1/5652.htm. [doi: 10.13328/j.cnki.jos.005652]

[47] Owicki S, Gries D. An axiomatic proof technique for parallel programs i. Acta Informatica, 1976, 6(4): 319–340.

[48] Yang J, Hawblitzel C. Safe to the last instruction: automated verification of a type-safe operating system. In: Proc. of the 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York, NY, USA: Association for Computing Machinery, 2010. 99–110. [doi: https://doi.org/10.1145/1806596.1806610]

[49] Boogie: an intermediate verification language.2020. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/.

[50] Nelson L, Sigurbjarnarson H, Zhang K, Johnson D, Bornholt J, Torlak E, Wang X. Hyperkernel: push-button verification of an os kernel. In: Proc. of the 26th Symp. on Operating Systems Principles (SOSP). Shanghai China: ACM, 2017. 252–269.

[51] Sigurbjarnarson H, Bornholt J, Torlak E, Wang X. Push-button verification of file systems via crash refinement. 2016. 1–16. [doi: https://doi.org/10.1145/3132747.3132748]

[52] Xu F, Fu M, Feng X, Zhang X, Zhang H, Li Z. A practical verification framework for preemptive os kernels. In: Proc. of the Conf. on Computer Aided Verification. Springer, Cham, 2016. 59-79. .

[53] Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G. Comprehensive formal verification of an os microkernel. ACM Trans. on Computer Systems (TOCS), 2014, 32(1):1–70. [doi: https://doi.org/10.1145/2560537]

[54] Merz S. Model checking: a tutorial overview. In Summer School on Modeling and Verification of Parallel Processes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. 3–38.

[55] Visser W, Havelund K, Brat G, Seungjoon Park. Model checking programs. In: Proc. of the Conf. of Automated software engineering (ASE), 2003, 10(2):203-232..

[56] Kim M, Hong S, Hong C, Kim T. Model-based kernel testing for concurrency bugs through counter example replay. Electronic Notes in Theoretical Computer Science, 2009, 253(2):21–36. [doi: https://doi.org/10.1016/j.entcs.2009.09.049]

[57] Holzmann GJ, Joshi R. Model-driven software verification. Graf S, Mounier L. Model Checking Software. ACM Computing Surveys (CSUR), 2009, 41(4):1-54. [doi: https://doi.org/10.1145/1592434.1592438]

[58] Holzmann GJ. The model checker SPIN. IEEE Trans. on Software Engineering, 1997, 23(5):279–295. [doi: https://doi.org/ 10.1109/32.588521]

[59] Flanagan C, Ave L, Freund SN. Type-based race detection for java. In: Proc. of the ACM SIGPLAN 2000 Con. on Programming Language Design and Implementation (PLDI). 2000. 219-232. [doi: https://doi.org/10.1145/349299.349328].

[60] Sasturkar A, Agarwal R, Wang LQ, Stoller S D. Automated type-based analysis of data races and atomicity. In: Proc. of the tenth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2005. 83-94. [doi: https://doi.org/10.1145/1065944.1065956]

[61] Vojdani V, Apinis K, Rõtov V, Seidl H, Vene V, Vogler R. Static race detection for device drivers: the Goblint approach. In: Proc. of the 31st IEEE/ACM International Conf. on Automated Software Engineering (ASE). Singapore, Singapore: ACM Press, 2016. 391–402.

[62] Kildall GA. A unified approach to global program optimization. In: Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL). Boston, Massachusetts: ACM Press, 1973. 194–206. [doi: https://doi.org/10.1145/512927.512945]

[63] Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. ACM SIGOPS operating systems review, 2003, 37(5): 237-252.. [doi: https://doi.org/10.1145/1165389.945468]

[64] Bai JJ, Wang YP, Lawall J, Hu SM. DSAC: Effective static analysis of sleep-in-atomic-context bugs in kernel modules. In: Proc. of the 2018 USENIX Annual Technical Conference (USENIX ATC). 2018. 587-600.

[65] King J C. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7):385-394.. [doi: https://dl.acm.org/doi/10.1145/360248.360252]

[66] Tan L, Zhou Y, Padioleau Y. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In: Proc. of the 2011 33rd International Conf. on Software Engineering (ICSE). IEEE, 2011. 11-20. [doi: https://dl.acm.org/doi/ 10.1145/1985793.1985796]

[67] Breuer PT, Valls MG. Static deadlock detection in the linux kernel. In: Proce. of the International Conf. on Reliable Software Technologies. Springer, Berlin, Heidelberg, 2004. 52-64.

[68] Hong S, Kim M. Effective pattern-driven concurrency bug detection for operating systems. Journal of Systems and Software, 2013, 86(2): 377–388. [doi: https://doi.org/10.1016/j.jss.2012.08.063]

[69] Andrianov P, Mutilin V, Khoroshilov A. Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proc. of the International Conf. on Tools and Methods for Program Analysis. Springer, Cham, 2017. 11-23.

[70] Pozniansky E, Schuster A. Efficient on-the-fly data race detection in multithreaded C++ programs. In: Proc. of the ninth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2003. 179-190. [doi: https://doi.org/10.1145/781498.781529]

[71] Rajagopalan AK, Huang J. RDIT: race detection from incomplete traces. In: Proc. of the 10th Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). Bergamo, Italy: ACM Press, 2015. 914–917. [doi: https://doi.org/doi/10.1145/2786805.2803209]

[72] Seyster J, Radhakrishnan P, Katoch S, Duggal A, Stoller SD, Zadok E. Redflag: A framework for analysis of kernel-level concurrency. In: Proc. of the International Conf. on Algorithms and Architectures for Parallel Processing. Springer, Berlin, Heidelberg, 2011. 66-79.

[73] What is rcu? 2020. https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.

[74] Chen QL, Bai JJ, Jiang ZM, Lawall J, Hu SM. Detecting data races caused by inconsistent lock protection in device drivers. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou, China: IEEE, 2019. 366–376. [doi: https://doi.org/doi/10.1109/SANER.2019.8668017]

[75] The llvm compiler infrastructure project. 2020. https://llvm.org/.

[76] Kernel-strider. 2020. https://github.com/euspectre/kernel-strider.

[77] Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective data-race detection for the kernel. In: Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI). 2010..

[78] Jiang Y, Yang Y, Xiao T, Sheng T, Chen W. DRDDR: a lightweight method to detect data races in linux kernel. The Journal of Supercomputing, 2016, 72(4):1645–1659.

[79] Blum B. Landslide: systematic dynamic race detection in kernel space. 2012.

[80] Magnusson PS, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B. Simics: a full system simulation platform. Computer, 2002, 35(2):50–58. [doi: https://doi.org/doi/ 10.1109/2.982916]

[81] Fonseca P, Rodrigues R, Brandenburg BB. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In: Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2014. 415-431..

[82] QEMU. 2020. https://www.qemu.org/.

[83] Zhou C, Wang M, Liang J, Jiang Y. Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling. In: Proc. of the Conf. of Automated software engineering (ASE). 2020.

[84] Chen H, Guo S, Xue Y, Sui Y, Zhang C, Li Y, Wang H, Liu Y. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: Proc. of the 29th USENIX Security Symposium (USENIX Security). 2020. 2325-2342.

[85] Vinesh N, Sethumadhavan M. ConFuzz—a concurrency fuzzer. In: Proc. of the First International Conf. on Sustainable Technologies for Computational Intelligence. Springer, Singapore, 2020. 667-691.

[86] Xu M, Kashyap S, Zhao H, Kim T. Krace: data race fuzzing for kernel file systems. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco, CA, USA: IEEE, 2020.1643–1660.

[87] Lochmann A, Schirmeier H, Borghorst H, Spinczyk O. LockDoc: trace-based analysis of locking in the linux kernel. In: Proc. of the Fourteenth EuroSys Conf. 2019 (EuroSys). Dresden, Germany: ACM Press, 2019. 1–15. [doi: https://doi.org/10.1145/3302424.3303948]

[88] Gu R, Shao Z, Chen H, Wu X (Newman), Kim J, Sjöberg V, Costanzo D. CertiKOS: an extensible architecture for building certified concurrent os kernels. 2016. 653–669.

[89] Lu S, Tucek J, Qin F, Zhou Y. AVIO: detecting atomicity violations via access interleaving invariants. ACM SIGOPS Operating Systems Review, 2006, 40(5):37–48. [doi: https://doi.org/10.1145/1168917.1168864]

附中文参考文献:

[20] 苏小红, 禹振, 王甜甜,等. 并发缺陷暴露、检测与规避研究综述. 计算机学报, 2015, 395(11):2215-2233. [doi: https://doi.org/ 10.11897/SP.J.1016.2015.02215].

[46] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33-61.http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]