融合依赖推断与权重优化的Linux内核模糊测试

张 弋[1,2,3],王 豫[1,2,3*],王林章[1,2,3]
(1. 计算机软件新技术全国重点实验室,江苏南京 210023;2. 南京大学计算机学院,江苏南京 210023;
3. 软件新技术与产业化协同创新中心,江苏南京 210023)

摘 要: 在面向人机物融合场景的泛在操作系统中,内核的稳定性与安全性是支撑多元化应用和异构资源融合的基础。Linux内核作为当前具有代表性的操作系统内核之一,因规模庞大、逻辑复杂而容易产生缺陷,且运行在最高特权级,漏洞一旦被利用,将对泛在计算环境中的系统安全和应用可靠性造成严重威胁。模糊测试作为当前主流的内核漏洞挖掘方法,通常将系统调用序列作为测试用例输入,这些用例可以通过随机生成或在现有序列基础上变异得到。然而,随机组合不仅容易引发状态空间爆炸,还可能导致测试效率和漏洞发现能力难以保障。尽管已有研究尝试提升测试用例的有效性,但在系统调用依赖分析方面仍存在信息不完整、粒度不精细、利用率偏低、开销过高等问题,从而限制了模糊测试的有效性。对此,本文设计了一种融合依赖推断与权重优化的内核模糊测试机制。该方法首先通过静态分析Linux内核的源码,从中识别每个系统调用所涉及的资源和它们的访问特征;随后依据这些信息推导不同系统调用之间可能存在的依赖联系,并以此构造系统调用依赖图。模糊测试过程在该依赖图的指导下进行,以生成更具价值的测试用例。另外,通过收集运行阶段的反馈数据,对依赖图中边的权重不断进行调整,使其逐步反映更精确的调用关系,从而持续推动模糊测试效果的提升。本文基于该方法实现了原型工具SDKernelFuzzing,在4个不同版本的Linux内核上进行实验。与Moonshine相比,本文所提的方法在保持较低额外开销的前提下,能够更加高效地生成数量更丰富、内容更完整的系统调用依赖信息,从而构建出更加全面的依赖图。此外,相较于Syzkaller以及目前的最新相关研究,本文提出的方法在代码覆盖率和漏洞挖掘能力上均展现出明显优势。在相同实验环境下,该方法的平均代码覆盖率比最佳基线方案提升了7.69%,而发现的平均缺陷数量则提升了15.77%。消融实验结果表明,系统调用依赖图中不同的构建策略对模糊测试性能均产生了显著的影响。

关键词: 模糊测试;内核;Linux;依赖关系;测试;漏洞

基金项目: 教育部基础学科和交叉学科突破计划(No.JYB2025XDXM118);国家重点研发计划(No.2024YFB2505604);国家自然科学基金(No.62232001,No.62172200)

中图分类号: TP311.5 **文献标识码:** A **文章编号:** 0372-2112(2026)04-1629-22

电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20251150

Fuzz Testing of the Linux Kernel Integrating Dependency Inference and Weight Optimization

ZHANG Yi[1,2,3], WANG Yu[1,2,3*], WANG Linzhang[1,2,3]
(1. *State Key Laboratory for Novel Software Technology*, *Nanjing*, *Jiangsu* 210023, *China*;
2. *School of Computer Science*, *Nanjing University*, *Nanjing*, *Jiangsu* 210023, *China*;
3. *Collaborative Innovation Center of Novel Software Technology and Industrialization*, *Nanjing*, *Jiangsu* 210023, *China*)

Abstract: In ubiquitous operating systems for human-machine-object integrated scenarios, the stability and security of the kernel form the foundation for supporting diversified applications and heterogeneous resource integration. As one of the most representative operating system kernels today, the Linux kernel is prone to defects due to its large scale and complex logic. Running at the highest privilege level, any exploited vulnerability could seriously threaten system security and application reliability in ubiquitous computing environments. Fuzzing, as the most mainstream method for discovering kernel vulnerabilities, typically uses sequences of system calls as test case inputs. These test cases can be randomly generated or derived by mutating existing sequences. However, random combinations not only easily lead to state space explosion but also result in low testing efficiency and limited vulnerability discovery capability. Although existing studies have attempted to improve the effectiveness of test cases, issues remain in system call dependency analysis, including incomplete information, coarse granularity, low utilization, and high overhead, which in turn limit the effectiveness of fuzzing. To enhance the effectiveness of kernel fuzzing, this paper introduces a testing framework that fuses dependency inference with weight opti-

收稿日期:2026-02-06;录用日期:2026-03-30;责任编辑:宋甲英
*通信作者:王豫

mization. The core idea of this approach is to conduct static analysis on the Linux kernel source code to identify the resources accessed by each system call and characterize their access patterns. Based on this information, potential dependency relationships among system calls are inferred, and a corresponding dependency graph is constructed. The fuzzing process is then guided by this graph to generate more effective test cases. In addition, feedback collected during runtime is used to continuously update the weights of the graph's edges, enabling the dependency graph to increasingly reflect accurate call relationships and thereby further improving the overall fuzzing performance. Based on this method, a prototype tool named SDKernelFuzzing was implemented, and experiments were conducted on four different versions of the Linux kernel. Compared with Moonshine, the proposed approach can generate system-call dependency information that is both more comprehensive and richer in content, while incurring only minimal additional overhead, thereby enabling the construction of a more complete dependency graph. In addition, relative to Syzkaller and other state-of-the-art techniques, our method demonstrates significant advantages in both code coverage and bug-finding capability. Under identical experimental conditions, the average code coverage achieved by our method exceeds the best baseline by 7.69%, and the average number of detected bugs increases by 15.77%. Ablation experiment results show that different construction strategies of system call dependence graph have a significant impact on fuzzing performance.

Keywords: fuzzing; kernel; Linux; dependencies; testing; vulnerability

Foundation Item(s): Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No.JYB2025XDXM118); National Key Research and Development Program (No.2024YFB2505604); National Natural Science Foundation of China (No.62232001, No.62172200)

0 引言

随着人机物融合与泛在计算的不断发展,操作系统作为计算环境的基础设施,已从传统的单机和网络环境扩展到支持多元化场景与异构资源的关键平台。内核作为操作系统的核心组件,不仅需要高效管理计算、存储、网络与设备资源,还要在泛在操作系统中为跨场景应用提供稳定、安全的支撑。为了满足复杂场景需求,内核通常规模庞大且结构复杂,容易隐藏潜在缺陷。在众多操作系统内核中,Linux内核凭借其开源、可裁剪和广泛应用的特性,已成为服务器、移动终端、嵌入式设备以及物联网系统的核心基础。然而,正因为其广泛部署与高特权运行环境,Linux内核中的漏洞一旦被攻击者利用,可能引发信息泄露、拒绝服务或权限提升等严重安全事件,对人机物融合环境中的系统安全和应用可靠性造成威胁[1]。因此,如何在泛在计算环境下高效发现并修复内核缺陷,已成为保障系统鲁棒性与支撑人机物融合应用的关键问题[2]。

关于Linux内核漏洞检测的现有研究可分为静态分析、动态测试和动静结合三类。其中,动态测试以模糊测试(fuzzing)为主。在用户态软件安全检测中,静态分析与模糊测试已取得良好实践。静态分析通过审查源代码或其中间表示,推断程序在运行时可能呈现的行为,从而定位潜在问题。然而,在处理规模庞大的软件时,不可避免地需要在分析速度与结果精度之间作出权衡。同时,静态分析多针对Linux内核的特定模块,如设备驱动[3-5]和文件系统[6],或聚焦特定漏洞类型[7-9],如数据竞争[8]。虽然这些工作已经取得了一定的成效,但受限于代码规模,同时其通用性和全面性很有限。模糊测试则依赖虚拟化[10]等技术通过持续输入非预期数据触发异常[11-16]。在现有的自动化内核模糊测试工具中,Syzkaller[12]已成为被广泛采用的框架之一,许多相关研究工作[5,13,15-16]都将它作为实验基础或核心测试平台。虽然其速度快且误报率低,但覆盖范围依赖于测试用例质量,难以遍历全部执行路径。与普通应用相比,内核代码与功能更为复杂,尽管模块化设计降低了部分耦合度,但模块间交互频繁且与硬件高度绑定。这给静态分析带来性能与精度的双重挑战,也扩大了模糊测试[17-19]需要探索的状态空间。因此,将静态分析与模糊测试相结合的动静态协同技术,正逐渐成为提升内核漏洞发现效率的重要研究方向,即先对内核源码进行静态分析,再将分析结果用于指导模糊测试,从而兼具两者优点。例如,HFL[16]将符号执行与模糊测试结合,探索更深层的代码路径,但仍受制于内核复杂性与约束求解器性能。已有研究[11,13-14,20]也尝试利用静态分析或机器学习精简真实系统调用序列作为模糊测试的种子,但测试的充分性依赖于种子的多样性,且缺乏后续用例生成的有效指导。SyzDirect[11]对Linux内核进行静态分析来得到系统调用及其参数条件等信息,KernelGPT[14]通过大语言模型自动合成系统调用规范以增强内核模糊测试,但它们都缺乏关于系统调用的依赖分析,从而在探索更深层次的路径方法上具有局限性。SyzGen++[17]基于外部行为模式推断显式依赖关系,但缺乏从内核内部源码推断依赖关系的能力,导致对隐式依赖或依赖细节的识别不足。

Moonshine[13]的工作指出系统调用之间并非孤立,但其分析流程主要聚焦于有限的调用关系,对指针行为和深层次的函数调用链缺乏处理。因此,该方法最终得到的依赖信息仍显不完整,尤其是对间接关联的刻画较为不足,进而限制了其在实际场景中的使用效果。

总体来看,当前研究在提取系统调用依赖信息方面仍面临诸多局限,例如依赖内容覆盖不全、分析粒度偏粗、生成信息难以充分利用、成本较高等,这些因素共同制约了模糊测试能力的进一步提升。需要注意的是,系统调用在内核中的执行过程会受到内部状态的影响,不同的状态会触发不同的执行路径,并在运行后进一步改变内核状态。而系统调用之间往往建立在特定状态的前提之上。内核状态通常可由相应的资源表现出来,因此从资源访问关系切入,对系统调用与资源间的交互进行刻画,不仅有助于形成更加完整的依赖模型,还能够更有效地辅助测试用例的生成与变异的过程。

当前方法往往只捕获部分调用间的关联,缺乏丰富的依赖描述,并且未能将其有效地用于优化测试过程,本文针对这一痛点提出了一种融合依赖推断与权重优化的Linux内核模糊测试新方法。该方法的基本思路如下:结合静态分析和内核模糊测试过程,静态分析用于推断系统调用访问的内核资源和交互方式,并推导调用之间可能存在的多层次依赖关系(包括显式与隐式依赖、资源关联、约束前置条件等);随后利用由这些依赖关系构建而成的依赖图引导模糊测试过程。同时,该方法还结合运行时收集的执行数据动态调整依赖图权重,使依赖模型在持续迭代中不断完善,从而进一步强化模糊测试。

本文的核心贡献在于首次将面向资源的系统调用依赖推断与动态权重优化机制结合起来,使依赖关系在模糊测试过程中能够不断自适应地完善,从而实现了比现有方法更全面、更精细且更具可操作性的内核系统调用依赖建模体系,有效提升了内核模糊测试的路径探索深度与覆盖能力。其中,“更全面”体现为依赖建模不仅覆盖显式的系统调用顺序关系,还进一步结合了基于内核资源访问的隐式依赖,从而能够刻画更多潜在的依赖关系;“更精细”体现为通过权重机制对不同依赖关系在引导模糊测试发现新行为过程中的贡献程度进行量化与区分,而非将所有依赖关系视为等价;“更具可操作性”则体现为依赖关系通过动态权重优化机制,能够根据覆盖反馈在模糊测试过程中持续调整与优化。在依赖的静态分析方面,除引入轻量指针分析外,本文从内核资源访问视角切入,对系统调用在执行过程中涉及的关键资源进行统一建模,并基于资源的访问、修改与状态约束关系推导系统调用之间的潜在依赖,从而显著增强依赖信息的完整性与语义表达能力。在权重的动态优化方面,本文引入依赖关系有效比率来衡量该依赖关系对发现新行为的贡献程度,从而作为权重调整依据。该比率定义为某一依赖关系在被模糊测试选择并发生突变的情况下,能够产生新增覆盖的比例,即该依赖关系对应的有效突变次数与其被选择突变次数之比。同时,在测试过程中对序列中相邻系统调用执行顺序进行受控翻转,通过比较翻转前后覆盖变化来判断是否存在顺序约束。基于这两个策略,本文所提的方法使得依赖图逐渐逼近实际的运行语义。

基于本文提出的方法,我们实现了原型工具SDKernelFuzzing,并设计了4类研究问题来验证本文提出的方法在分析能力、依赖表达能力以及模糊测试实际效果上的优势。在依赖图构建阶段(RQ1),SDKernelFuzzing在维持可接受的时间与空间开销的同时,通过更全面的资源访问分析与更精细的指针/调用链处理,显著提高了依赖提取的精度与覆盖范围。在依赖信息完整性评估中(RQ2),SDKernelFuzzing所得到的系统调用依赖组数在不同内核版本上均达到Moonshine的8~10倍,对单条依赖关系内部属性的刻画能力也明显优于现有方法。具体而言,本文提出的方法为每一条系统调用依赖关系引入的权重信息量化了该依赖在引导模糊测试发现新行为过程中的实际贡献程度,从而刻画出依赖关系的强度。进一步地,在模糊测试实践中(RQ3),依赖图对测试序列生成与变异策略形成了有效指导,使得KCOV覆盖数和crash发现数在多个内核版本上相比于最优基准工作仍实现显著提升,特别是在网络、驱动、文件系统等复杂子系统中有效触发了深层级行为。最后,消融实验(RQ4)进一步展示了依赖构建策略的边际贡献,间接调用分析、对多种资源类别访问的建模以及权重设计与动态优化均对模糊测试效率具有累积促进作用。

综上所述,实验系统性地证明了本文依赖推断和动态优化的协同机制能够构建高质量、可演化的系统调用依赖模型,并在真实内核模糊测试场景下带来实质性的覆盖率提升和漏洞发现能力增强。

本文的主要贡献可概括为3个方面。

(1)搭建基于内核资源访问特征的新型系统调用依赖建模框架。该框架结合了函数指针分析技术,能够更准确、更全面地刻画系统调用之间的潜在依赖关系。同时,该框架通过引入权重对依赖关系进行量化。

(2)提出融合依赖推断与权重优化的内核模糊测

试策略。该策略在测试用例的生成与变异阶段依据依赖权重选择系统调用、拼接调用序列、调整调用参数,以提高测试用例与内核状态的一致性。同时,模糊测试过程中的反馈信息也可以进一步指导依赖权重信息。

(3)实现该方法的原型工具SDKernelFuzzing,并对其有效性进行评估。结果表明,SDKernelFuzzing能够高效构建信息丰富的依赖图,并显著提高代码覆盖率和发现缺陷的能力。

1 基础知识与相关工作

本章第一部分对操作系统内核及其工作模式进行了详细说明,第二部分则侧重于探讨当前已有相关工作的进展情况。

1.1 操作系统内核与工作模式

作为支撑现代计算系统运行的关键软件,操作系统承担着协调硬件与应用程序、抽象底层资源并提供统一接口的重要角色。无论是大型服务器、工业控制设备,还是轻量级的智能终端,这一软件层始终在后台维护系统的正常运行。而处于操作系统核心位置的是内核,它负责处理任务调度、内存分配、文件系统组织、外设交互以及网络通信等基础机制,是整个系统功能得以实现的根本。

1.1.1 内核资源

内核资源是操作系统内核管理和调度的各类硬件(如CPU)与软件(如内存页)实体的总称,用于支撑系统的基本功能和服务。内核通过抽象机制将硬件资源封装为可被程序访问的软件资源。这些软件资源为用户态程序提供了一种间接访问硬件的手段,同时确保不同程序之间的资源隔离和安全性。

1.1.2 系统调用

系统调用是用户程序与操作系统内核之间的桥梁,即用户程序与内核资源之间的桥梁。在操作系统中,内核负责管理底层硬件资源,并提供各种核心服务,例如任务调度、内存管理和网络通信。由于用户程序运行在受限的权限环境中,无法直接访问这些资源或服务,这就需要通过系统调用向内核请求操作或服务。系统调用的执行通常伴随着从用户态切换到内核态,从而在保证安全和隔离的前提下使程序能够合法地使用底层资源。

1.1.3 内核工作模式

操作系统内核的工作模式可以理解为“由系统调用驱动、受内核状态约束的动态执行过程”。用户态程序必须通过系统调用访问内核资源,但系统调用的执行依赖于当前内核状态及调用顺序。以文件操作为例,write系统调用在向文件写入数据前通常需要先调用open打开文件,由内核返回文件描述符fd,并在后续的write调用中使用该fd。若fd非法(未与文件对象建立映射或无写权限),write将在浅层合法性检查中直接返回错误,无法触达核心逻辑。换言之,只有当内核状态正确时,系统调用才能深入执行核心功能。

图1为内核工作模式的状态示意图。用户程序通过系统调用进入内核,内核根据资源状态决定调用是否能深入执行,条件满足则执行核心逻辑并更新状态,否则直接返回错误。

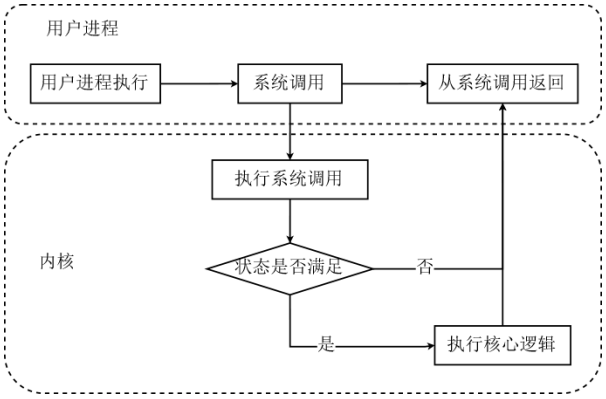


图1 内核工作模式状态示意图

Figure 1 State diagram of kernel working mode

对内核模糊测试而言,这一工作模式提示我们,若测试用例未满足内核状态条件,系统调用仅停留在浅层检查,难以触达核心逻辑。因此,模糊测试方法必须考虑内核状态和系统调用依赖,以提升测试的有效性和覆盖率。

1.2 相关工作

目前,内核的缺陷检测主要包括静态分析、动态测试(主要是模糊测试)以及结合静态与动态分析的混合方法。本节将按照方法类别依次综述相关研究进展。

1.2.1 静态分析方法

静态分析无需运行内核即可检测缺陷,但依赖内核源代码。由于主流商业操作系统(如Windows、macOS)闭源,研究多集中于开源的Linux内核。设备驱动作为内核与外设交互的重要接口,其代码通常也纳入内核缺陷检测的范围。

Linux静态分析工具在早期通常是和内核紧密耦合的,由内核开发者实现。Sparse[21]支持对地址混用、锁配对等问题的检测,但缺乏数据流分析的能力。而Smatch[22]支持复杂检测任务。Coccinelle[23]基于泛化补丁语法,根据代码结构定义操作规则。这些工具已集成到内核构建系统,并在工业界中被实际

部署[24]。

对内核进行静态分析的通用工具(如Coverity[25])在支持长期内核分析上有着很大的局限性。这一局限主要源于内核代码的特殊性,例如其结构的高度复杂性和代码规模的庞大[15,26]。此外,静态分析依赖抽象语法树(Abstract Syntax Tree,AST)或中间表示(Intermediate Representation,IR)等中间结果,不同工具链间的不兼容性进一步限制了其应用[27]。为应对这些挑战,研究通常采取两种思路:一是聚焦于文件系统、驱动程序或网络子系统等相对独立、外部交互频繁且易受攻击的模块[4];二是针对并发错误、内存错误或语义错误等特定缺陷类型开展专门检测。围绕特定子模块的研究中,Machiry等人[4]对驱动程序进行分析;Bai等人[6]提出了DSAC方法检测SAC缺陷并自动生成修复补丁,但未能处理函数指针问题。针对特定缺陷的研究中[9,28-32],Wang等人[29]结合污点分析与约束求解检测整数错误;Xu等人[30]形式化定义double-fetch缺陷,并借助符号执行加以确认;Bai等人[31]基于摘要的LockSet算法检测UAF缺陷,但同样未处理函数指针相关情况;ErrHunter[9]则通过系统静态分析检测Linux内核中的错误处理错误。钱振江等人[32]对内存管理模块设计和实现的正确性进行了形式化验证。Cocktail[33]将间接函数调用划分为函数参数、结构体字段和全局变量三类,之后基于该分类来指导指针分析精度的调度策略,即对不同类型的函数调用应用不同精度的指针分析。本文在静态分析阶段也将函数调用进行分类,但并未将函数指针分类用于决定是否提升分析精度,而是将它作为函数指针统一标识与建模的基础,将函数指针解析问题转化为指向约束传播问题,从而避免对完整内存别名关系的分析。

此外,在静态内存缺陷检测领域,稀疏值流分析已成为一种重要的技术手段。其核心思想是通过跟踪程序中值的传播路径,构建稀疏值流图,从而检测诸如内存泄漏、空指针解引用等缺陷。早期工作Saber[34]提出了全稀疏值流分析框架,为后续研究奠定了基础。随后,为提升在大规模代码上的可扩展性,Pinpoint[35]提出了面向百万行代码的快速而精确的稀疏值流分析方法。在分析框架与工具层面,SVF[36]提供了一个可扩展的跨过程静态值流分析平台。进一步地,SILVA[37]通过引入分层与增量更新机制,提出了分层增量稀疏值流分析方法,显著提升了在大型代码库上的分析效率。总体而言,这些基于稀疏值流的静态分析工作为内核及系统级代码中的内存缺陷检测提供了坚实的理论基础和工具支持。

综上,静态分析方法覆盖范围广,适合在运行前发现潜在问题,但受限于代码规模、复杂性与工具链兼容性,同时其通用性和全面性也很有限。

1.2.2　模糊测试

模糊测试(fuzzing)[17,38-41]是目前内核缺陷动态检测的核心手段,通过自动化生成输入并观察运行时异常来发现漏洞。在内核领域,早期的模糊测试方法[19]缺乏有效的反馈机制,通常直接在物理机上运行且没有完善的监控模块,效率较低。之后,Google开发的Syzkaller[12]显著推动了研究进展,并且成为近年多数工作的平台基础。Syzkaller支持自动生成测试用例、内核监控与崩溃复现,是目前用于内核的漏洞挖掘研究最广泛的工具。近年来,很多工作在传统模糊测试的框架上做出了扩展。例如,KBinCov[41]提出了内核二进制覆盖率反馈,用内核二进制的覆盖率作为输入生成的指导。

此外,模糊测试的输入不局限于系统调用序列,还可以扩展至硬件与操作系统之间的交互边界。例如,Song等人[42]提出的PeriScope框架对设备驱动与硬件交互进行细粒度分析。

总体而言,内核模糊测试仍需应对状态空间爆炸、多架构适配、外设多样性等挑战。

1.2.3　动静结合方法

动静结合方法旨在利用静态分析结果制导动态测试,从而兼具前者的覆盖广度与后者的准确性,这些方法通常是静态分析和模糊测试的结合。静态分析能够在测试前识别出潜在的高风险代码位置,而动态测试则在运行中验证这些位置是否可触发缺陷,同时反向提供运行时的信息,以优化静态分析。

符号执行是该类方法中较为常见的实现路径,可在生成测试用例时降低盲目性。Renzelmann等人[3]开发了SymDrive系统,结合静态分析与源到源转换,降低测试成本。混合模糊测试通过结合符号执行与传统模糊测试,根据分支条件动态调整输入生成策略,以覆盖更深层次的执行路径。Kim等人[16]的研究对该流程进行了改进,包括在编译阶段将间接控制流转换为直接控制流、通过指针分析推断系统调用依赖关系并优化其执行顺序,以及结合符号与具体执行推断嵌套参数结构。另一类方法则基于日志或系统调用跟踪生成初始种子输入,例如Han等人[18]从API(Application Programming Interface)日志推断调用依赖关系并生成调用序列。然而,这些工作的有效性严重依赖于种子的质量,因此被种子的丰富性限制。与这些工作不同,本文的方法基于静态构建的系统调用依赖图来指导测试用例生成,并且初始阶段无需任何可用种子。因此,本文的方法不受种子数量和质量的限制。

为了提高对Linux内核的模糊测试效率，针对系统调用有效性的工作陆续出现。Pailoor等人[13]通过种子蒸馏在保留依赖关系的前提下裁剪调用轨迹，并据此实现了工具Moonshine。他们提出不同的系统调用之间有着直接与间接的依赖，但未考虑间接函数调用与指针分析，导致依赖关系不完整且应用有限。Tan等人[11]提出的SyzDirect采用定向灰盒模糊测试的方法，对Linux内核进行静态分析并识别有价值的信息，例如正确的系统调用及其参数条件，从而到达目标位置。Yang等人[14]提出的KernelGPT利用大语言模型在预训练期间已经积累的大量内核代码、文档和用例来提高系统调用规范合成的有效性。SyzGen++[17]定义了插入和查找操作的两个基本构建块及其配对，以准确识别依赖关系，可基于外部行为模式推断显式依赖关系。这些工作虽然提升了模糊测试的效率，但是在系统调用依赖分析方面仍存在信息不完整、粒度不足、利用率低、开销高等问题，限制了模糊测试效率的提升。例如，SyzDirect[11]和KernelGPT[14]缺乏关于系统调用的依赖分析，从而在探索更深层次的路径方法方面具有局限性；SyzGen++[17]则缺乏对隐式依赖或依赖细节的识别。

另外，也有一些研究针对内核中的特定问题开展了检测工作。RIV[43]是一种基于“推断-验证”模式的内核数据竞争检测方法，先通过分析线程执行情况与内存访问信息来推断潜在竞争变量对，再通过内存观测点与延时注入方式对潜在竞争变量进行定向验证。

2 方法

2.1 方法框架

软件的正确性与安全性极为重要[44-45]，内核作为计算系统运行的关键软件更是如此。本研究提出一种融合依赖推断与权重优化的内核模糊测试策略，旨在利用系统调用之间的依赖关系，提升模糊测试的效率。内核不同系统调用的正常执行往往依赖于先前系统调用对内核状态的设置。在内核实现层面，这些依赖关系可通过内核资源的使用情况加以刻画。现有研究在依赖关系的完整性、依赖信息的丰富性以及依赖信息在模糊测试中的利用方面仍存在不足。为此，本文针对这一痛点，利用对各个系统调用的内核资源访问内容及其访问方式的静态分析来推断各个系统调用之间存在的潜在依赖关系。被推断的依赖关系用于构建带有依赖权重的系统调用依赖图，从而为后续模糊测试中测试用例的生成与变异提供指导。这有助于提高测试用例对内核状态和依赖关系的匹配度，增强深层逻辑的覆盖能力。

本文的方法框架如图2所示，具体包括系统调用依赖图构建和依赖图驱动的模糊测试两个方面。

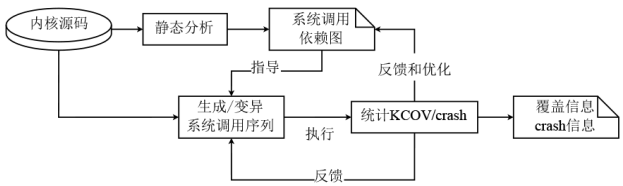


图2 融合依赖推断与权重优化的内核模糊测试框架

Figure 2 Kernel fuzzing framework fusing dependency inference and weight optimization

第一个方面是系统调用依赖图的构建过程。该过程主要利用的技术是指针分析。在这一过程中，本文的方法构建函数调用图和每个系统调用的详细信息（包括参数、返回值类型和内核资源）。之后，利用这些信息来推断其潜在依赖关系。依赖关系以权重形式描述，并以依赖图的方式组织，用于直观表示系统调用间的依赖程度。依赖图驱动的模糊测试则在测试用例生成和变异过程中引入依赖图信息，依据依赖权重随机选择或拼接系统调用序列，并对插入、删除和参数变异等操作进行指导，从而提高测试用例的质量和有效性。同时，模糊测试过程的反馈信息又反过来被用于指导调整依赖权重，进一步提升测试效率。

2.2 基于静态分析的系统调用依赖图构造

本文提出的由静态程序分析生成调用依赖图的过程，其核心是内核状态和内核资源。内核状态是内核在运行过程中对各类资源和系统信息的整体描述，它反映了内核的当前运行情况和环境条件。在执行系统调用时，内核状态会动态变化，不同的系统调用依赖于特定的状态前提才能正确执行。因此，随机调用组合的内核模糊测试往往效率极低，而内核资源是构成内核状态的具体实体。本节基于内核资源访问模式进行分析，该方法如图3所示。

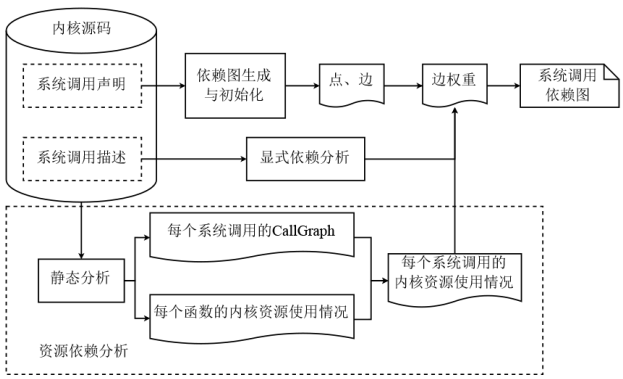


图3 由静态程序分析生成调用依赖图

Figure 3 The call dependence graph is generated by static program analysis

该方法包含以下步骤。

(1)基于内核源码生成不精确的依赖图。通过系统调用的声明信息生成依赖图。

(2)显式依赖分析。依托各个系统调用的详细信息初步生成带权重的显式系统调用依赖关系。

(3)资源依赖分析。该步骤分为内核资源使用提取和函数调用依赖关系重建两个主要环节。内核资源使用提取直接由静态分析完成,而函数调用依赖关系重建则利用函数指针分析对函数指针的指向信息进行收集,从而构建调用图。在完成上述两个环节后,将提取的资源使用信息和调用关系结合,可完成对整个系统调用依赖图的完善(例如边的权重信息)。

2.2.1　轻量级函数调用图构造方法

大多数操作系统内核由C语言与汇编语言编写。由于C语言不具备类继承或虚函数等面向对象特性,所有间接调用均依赖函数指针,因此函数指针分析是构建调用图的核心难点。现有方法主要包括基于类型分析和基于常规指针分析两种。基于类型分析将与间接调用点函数类型一致的所有函数视为可能目标,其中的函数类型包含返回值、参数类型及顺序。该方法速度快、漏报率低,但在模块化的内核中误报率高。基于常规指针分析,通过分析所有指针变量与内存位置的映射关系确定指向,误报率低,但时间复杂度高,不适合大规模内核代码。

为在准确性与效率间取得平衡,本文提出了一种轻量级的函数调用图构建方法。该方法的设计思想是仅针对函数指针变量进行重点分析,同时结合类型信息进行近似处理,并对指针分析的约束条件进行简化,从而在保证合理精度的前提下显著降低分析开销。具体来说,该方法对程序中的函数指针进行分类,并为每类函数指针分配唯一的标识。这个标识既用于后续函数指针指向约束的分析过程,也用于标记间接调用的位置,形成统一的分析依据。随后,通过扫描代码中的函数指针赋值语句,分析得到各个函数指针可能指向的约束信息,而其对应的具体函数集合则通过迭代求解方式推导。这一过程能够尽可能准确地还原函数指针可能的目标函数。该方法有以下优势:一方面,显著减少了分析的规模;另一方面,关键语义信息得以保留。与常规指针分析相比,该方法降低了时间与空间开销,与纯类型分析相比则有效减少了误报。此外,它可与内核资源分析同步执行,从而支持后续的资源依赖推导。

我们对Linux内核中的主要函数指针进行了调研与分类,从而对这些函数指针进行标识,以用于函数指针分析。调研结果包含三类函数指针:结构体或联合体成员、函数参数以及其他类型。第一类在具体的内核结构中较为常见,例如文件系统操作函数、驱动接口中的回调函数等。第二类通常作为回调函数在函数间传递,实现灵活的功能扩展和事件处理。第三类包括其他类型的函数指针变量,主要分为全局变量和局部变量,并包含函数指针数组。这类指针的作用域相对明确,通过全局或局部标识可以快速定位其使用范围。具体标识方法如下。

(1)结构体或联合体成员:“struct 结构体名.域成员名”。例如,对于ext4文件系统定义的“struct file_operations ext4_file_operations”,基于该规则被具体标识为“struct file_operations.open”。另外,为了降低分析开销,我们对该类成员的标识不区分具体结构体变量。也就是说,包含同样域名的不同具体结构体变量有着同样的标识。

(2)函数参数:“函数名@参数序号”。其中,参数序号从0开始。例如,在“register_chrdev_regio(struct dev_t *dev, handler_t handler, void *t)”中,第1个参数“handler”为函数指针类型,那么按照规则将其标识为“register_chrdev_regio@1”。

(3)其他函数指针变量:用“Global”和“函数名”来区别全局和局部变量,并附变量名。具体来说,表示为“Global@变量名”和“函数名@变量名”。例如,“assign_position_fix()”中的“azx_get_pos_callback_t callbacks”直接标识为“assign_position_fix@callbacks”,不对数组下标进行区分,从而降低分析的复杂度。

在完成对函数指针的统一标识后,下一步是对函数指针变量进行指向分析。函数指针的指向信息在程序运行期间并非随时变化,而是仅在特定操作下发生改变,主要包括变量的初始化和赋值操作。初始化操作可以被视为赋值的一种特殊形式,即在变量声明的同时将其指向某个函数或另一个指针,因此在指向分析中可以用统一的处理流程进行解析。可见,指向分析的核心任务是识别程序中所有与函数指针相关的赋值语句,并对这些语句的语义进行详细解析。通过对赋值表达式的分析,可以逐条记录函数指针变量可能的指向对象,从而逐步累积出完整的指向集合,为后续分析提供准确依据。在Linux内核中,函数指针赋值可抽象为val1:=func|&func|val2。其中,func为具体函数名,取地址符&仅为语法兼容,与直接使用函数名等价。

在分析过程中,将目标函数func或val2指向的信息加入val1的指向集合中,逐步形成函数指针可能的指向关系网络。这种方法不仅可以处理直接赋值和间接赋值,还可以为复杂的跨函数调用和函数指针传递提供基础数据支持,从而为分析奠定基础。

来自参数的函数指针本质上等同于局部函数指针变量，但为支持跨过程分析，本文对其单独建模。在函数指针的跨过程分析中，有两种情况需要被视为特殊的指向约束：一种是函数声明时，其参数中定义了函数指针类型；而另一种情况是实际传递给该函数的实参是函数指针类型。具体而言，对于此类函数调用，我们将函数的形参视为一个函数指针节点，而实参所指向的函数或函数指针集合则作为其目标，通过建立“形参→实参”的映射关系来刻画跨过程的指向约束。这样可以保证在调用链跨越多个函数时，函数指针的潜在指向不会丢失，同时能够支持对复杂函数回调和间接调用的静态分析。接下来，以图4为参考，对过程间函数指针指向分析的具体操作流程进行详细说明，包括如何识别形参、解析实参，以及如何将这些信息整合到指向集合中，从而为调用图构建和系统调用依赖分析提供准确的数据基础。

```
1 /* Linux-5.8, kernel/sched/completion.c*/
2 static inline long __sched do_wait_for_common(struct completion *x,
                                long (*action)(long), long timeout, int state) {
3     …
4     timeout = action(timeout);
5     …
6 }
7 static inline long __sched __wait_for_common(struct completion *x,
                                long (*action)(long), long timeout, int state) {
8     …
9     timeout = do_wait_for_common(x, action, timeout, state);
10    …
11 }
12 static long __sched wait_for_common(struct completion *x,
                                            long timeout, int state) {
13    return __wait_for_common(x, schedule_timeout, timeout, state);
14 }
15 static long __sched wait_for_common_io(struct completion *x,
                                            long timeout, int state) {
16    return __wait_for_common(x, io_schedule_timeout, timeout,state);
17 }
```

图4 示例代码片段

Figure 4 Example code snippet

我们从上而下分析，在第9行的函数调用中，函数指针action被作为实参传递给被调用函数do_wait_for_common，从而在跨过程分析中形成了一条明确的指向约束。具体来说，该调用将调用函数的第1个形参与对应的实参集合{wait_for_common@1}关联起来，建立了“形参→实参”的映射关系，其中调用函数的第1个形参标识为do_wait_for_common@1；接下来对第13行的分析可以得到wait_for_common@1→{schedule_timeout}；最后，对第16行分析，并将结果合并：wait_for_common@1→{schedule_timeout，io_schedule_timeout}。

表1展示了图4所示代码片段经过指向分析后得到的函数指针标识与对应指向约束的集合情况。为了便于存储和查询这些分析结果，我们采用一种映射结构（Map）来表示函数指针与其指向集合的对应关系。在该映射中，标识每个函数指针变量的函数指针标识和函数指针可能指向的函数集合pts分别作为Map的key和value。通过这种结构，我们可以高效地追踪和管理函数指针在程序执行中的潜在目标函数。

表1 图4所示代码经过指向分析后得到的函数指针标识与对应指向约束的集合情况

Table 1 Function pointers identify and correspond to a set of constraints collected from the analysis of the code snippet of figure 4

用Map表示收集到的指向约束	
key 函数指针标识fp	value 指向集合pts
do_wait_for_common@1	__wait_for_common@1
__wait_for_common@1	schedule_timeout, io_schedule_timeout

我们接下来对指向约束进行求解。迭代处理每个函数指针标识，将集合中已知的具体函数逐步替换标识中的抽象引用，直到所有指向均解析为具体函数，也即采用工作集（Worklist）算法进行处理。算法1给出了求解过程的具体步骤。该算法将程序中的所有函数指针标识收集到一个初始工作集，用于跟踪尚未解析的指针（第1～2行）。在迭代过程中，对工作集中的每个函数指针标识对应的指向集合进行处理，若集合中的元素在Map中存在对应的指向信息，则用其指向集合中的具体函数替换该元素，否则将其移除（第5～14行）。通过不断迭代和替换，逐步将抽象指针标识解析为具体函数，直至所有指向集合收敛。

为了构建内核的调用图，从分析每个函数的调用关系入手，并以CalleeInfoSet作为记录调用目标的核心数据结构。在解析源代码时，首先处理显式调用语句。对于这类语句，被调函数可以直接从代码结构中获得，因此能够被立即加入对应函数的CalleeInfoSet中。而对于涉及函数指针的间接调用，处理方式则有所不同，我们不试图在此阶段解析其最终调用目标，而是将当前调用点抽象成一个“函数指针标识”，并将该标识放入CalleeInfoSet中，作为后续解析的占位符。为了说明这种做法，我们继续以图3中所展示的代码作为例子。该示例中，do_wait_for_common在第4行的timeout=action（timeout）中通过第1个参数timeout触发了一次间接调用，因此该位置对应的函数指针标识被记为do_wait_for_common@1。基于这一标识，我们将其加入do_wait_for_common的CalleeInfoSet，用以指示该函数存在一个尚未解析的间接调用点。图4

算法1　函数指针指向约束求解

输入: 函数指针与其指向集合的对应关系Map
输出: 具体函数指向信息的指向集合
1. 定义W为空的worklist
2. $W \leftarrow$ Map.keys()
3. **WHILE** $W \neq \emptyset$ **DO**
4.　从W中选择一个fp
5.　**FOR** f in Map[fp] **DO**
6.　　**IF** f是具体的函数 **THEN**
7.　　continue
8.　　**ELSE**
9.　　在Map[fp]指向集合中删除f
10.　　**IF** f不在Map.keys()中 **THEN**
11.　　在Map[fp]中加入Map[f]指向集合中的元素
12.　　**END IF**
13.　**END IF**
14.　**END FOR**
15.　**IF** Map[fp]中指向集合内不存在非具体函数 **THEN**
16.　　$W = W - \{fp\}$
17.　**END IF**
18. **END WHILE**

所展示的另一段代码同样经过上述处理流程。对其进行分析后,可以得到各个函数的CalleeInfoSet的初始内容,这些结果整理后列于表2中。

需要强调的是,此时的CalleeInfoSet中仍可能包含若干函数指针标识,它们尚未映射到具体的函数实体。在完成函数指针指向集合的求解后,构造完整调用图的关键步骤便是“回填”这些占位符。具体而言,我们通过查询指向集合,将CalleeInfoSet中的每个函数指针标识替换成其可能指向的具体函数,从而将间接调用还原为实际的函数调用关系。

通过上述步骤,所有直接与间接调用将被统一解析,最终得到一张涵盖真实调用关系的完整调用图。

表2　对图4代码片段分析得到的CalleeInfoSet信息

Table 2　CalleeInfoSet information obtained by analyzing the code fragment in figure 4

Caller	CalleeInfoSet	
	Type	Identifier
do_wait_for_common	Indirect Call	do_wait_for_common@1
__wait_for_common	Direct Call	raw_spin_lock_irq
	Direct Call	do_wait_for_common
	Direct Call	raw_spin_unlock_irq
wait_for_common	Direct Call	__wait_for_common
wait_for_common_io	Direct Call	__wait_for_common

总体而言,本文提出的轻量级函数指针分析方法在准确性上优于纯类型分析,同时比常规指针分析更加高效。常规指针分析虽然在大多数场景中都能保持较低的误报率,但分析代价较高,难以在大规模内核中实际应用。而相比于传统纯类型分析,本文提出的方法可以有效降低误报率。具体来说,如表3所示,从多个典型函数指针使用场景出发,对本文的轻量级分析方法与传统纯类型分析技术的准确性、漏报与误报特点进行定性比较。具体来说,传统纯类型分析由于完全依赖类型匹配,在多个场景下均存在明显的误报问题,例如结构体成员函数指针、回调函数参数传递以及模块化内核的分析。相比之下,本文提出的轻量级分析方法在这些场景中均表现出较为一致的优势:其通过对函数指针进行分类标识,并结合指针赋值关系构建约束,以迭代方式推导出可达目标,从而能在不同场景下有效剔除类型一致但不可达的目标函数,显著减少误报。同时,本文提出的轻量级方法在保持较低开销的前提下仍能保留关键语义信息,例如回调函数等常用指针的调用关系,从而使得略有增加的漏报可控,且保持了较高的准确性。

表3　轻量级函数指针分析方法与传统方法的定性对比

Table 3　Qualitative comparison between lightweight function pointer analysis methods and traditional methods

场景/分析对象	纯类型分析	本文方法	具体优势
结构体/联合体成员函数指针	误报高:类型一致的函数均被认为可达; 漏报低	误报适中; 漏报低	成员标识和约束求解过程剔除大部分不可达函数目标
回调函数作为参数传递	误报较高:参数类型匹配导致目标太多; 漏报低	误报适中; 漏报较低	通过跟踪传递链与语义保留缩小候选集
模块化内核场景(大量子系统、驱动并存)	误报极高:类型重名、接口复用严重; 漏报低	误报适中; 漏报低	分类标识和迭代求解能有效限制可达范围
复杂指针传播路径(多层间接赋值)	误报高; 漏报较低	误报适中; 漏报略高但可控	通过简化传播约束减少分析开销,但保留关键语义确保正确性

与Cocktail[33]相比,Cocktail通过按需提升指针分析精度,在对分析成本进行一定平衡的情况下实现了较高的精度;本文提出的方法则通过面向内核的函数指针建模,尽管相比之下精度较低,但显著提升了可

扩展性和分析稳定性,更适合大规模内核代码的调用图构造及后续依赖分析。具体来说,由于分析过程不依赖逐级升级的指针分析,本文方法的分析时间和结果质量更加稳定,不易受代码规模或复杂性的影响。

2.2.2 基于静态分析的内核资源检测方法

在操作系统中,内核资源是系统内部状态的直接物化形式,而Linux内核通常通过结构体的方式来组织和管理这些资源。一个典型例子是内存管理子系统使用的struct vma_struct,它不仅记录了进程对虚拟内存的需求,还提供了配套的管理逻辑。由于这类结构体在语义上绑定了具体的内核状态,我们将它们视为可分析的资源实体。

对于这类结构体,采用字段级粒度的建模方式。以等待队列相关代码为例,struct completion.done字段直接反映同步对象的完成状态:对该字段的写操作往往对应状态推进,而对其读操作则体现状态检查。相比于将整个struct completion作为一个资源对象,字段级建模能够更准确地区分不同状态变量的访问行为,从而避免引入不必要的依赖。

为了更细致地观察内核状态的变化,我们仍然使用第2.2.1节中的标识方式,对所有与具体内核状态绑定、可视为资源实体的结构体字段进行分析。另外,若实际分析时有具体的目标子系统或研究任务,可以仅选择其中具备相关语义的结构体字段作为资源对象。除结构体字段外,内核的全局变量同样是系统状态的重要组成部分,因而也被纳入资源体系。为了保持标识的统一性,以“Global 变量名”的方式记录全局变量资源。对于数组类型的全局变量,由于静态分析难以准确推导其索引变化,我们不对其下标进行区分,以避免引入不必要的复杂度。

总体而言,本文采用中等粒度的资源建模方式(如结构体字段、全局变量)。相较于函数级或子系统级的粗粒度建模,该方式能够更精确地反映内核状态的实际读写行为;而相较于语句级或指针别名级的细粒度建模,又降低了分析开销,使其适合在大规模内核代码上应用。

基于上述资源建模方法,本文提出的静态资源检测框架由两部分组成。

(1)函数级资源分析。为每个内核函数维护资源使用集合,遍历代码语句识别资源访问并标记读(R)或写(W)。赋值符号左侧的资源视为写,其余视为读;若同一资源既被读又被写,则标记为读写(R/W),以反映其双重访问行为。结合这一规则,对图4所示代码执行分析即可得到表4中展示的函数级资源使用结果。

(2)系统调用级整合。在这个阶段,关注点从单个函数拓展至系统调用。利用已经构建的调用图,从每个系统调用作为起始节点出发,逐步向外展开其可达的直接和间接被调函数,将沿途遇到的所有函数的资源集合不断合并,即可得到完整的资源使用情况。可采用队列机制逐层扩展调用关系,直至无新函数加入。

表4 图4所示代码的函数级资源使用结果

Table 4 Function-level resource usage results of the code fragment in figure 4

Function	Kernel Resource		
	Type	Identifier	R or W
do_wait_for_common	Structure.Field	struct completion.done	R/W
__wait_for_common	Structure.Field	struct completion.wait	R
	Structure.Field	struct swait_queue_head.lock	R
wait_for_common	Global	schedule_timeout	R
wait_for_common_io	Global	io_schedule_timeout	R

本文所提方法将同一类型的结构体成员视为同类资源,借鉴了基于类型分析的别名假设,从而保持分析的轻量性。实践中,Linux内核对结构体域的访问多为显式访问,少见完全隐式传递,避免了资源识别的系统性漏报。

2.2.3 系统调用依赖图构造方法

本节旨在系统化呈现系统调用依赖图的构建过程。整个流程从最初的图结构生成逐步推进到边权重的细化与优化,最终得到能够准确反映系统调用间关联性的依赖图。整体方法可划分为三个相互衔接的阶段,每个阶段侧重处理不同的信息来源。第一阶段,基于系统调用声明创建初始依赖图;第二阶段,考虑到系统调用之间可能通过参数类型或返回值类型建立隐式联系,本文提出的方法利用专家知识编制的系统调用说明文档,以识别这些类型级别的依赖关系,并据此调整图中边的权重,使其更贴近实际的类型关联强度;第三阶段,内核中资源使用的具体情况被用于细化边权重。

经过上述三个阶段后,系统调用依赖图得到完整构建。下面给出该依赖图的形式化定义。

定义1 系统调用依赖图

该图包含了丰富的语义信息,其本质上是一个带权的有向图。用四元组对其进行形式化描述:

$$D=(V,\Omega,E,\Phi) \tag{1}$$

其中,V表示节点集合。图中的每一个节点对应一个系统调用,节点的标识即系统调用的名称,用于表示不同系统调用点集合。$\Omega(a)$表示记录在节点a上的信息。E是有序边集合,若边$e=<a,b>\in E$,表示系统调

用b依赖于系统调用a。$\Phi(e)$表示边信息函数。对于每一条边$e=<a,b>$,该函数记录边的属性,其中核心的是边权重φ。边权重用于度量依赖强度,反映系统调用b在实际执行中跟随a出现的可能性或倾向。

在本节,只关注Ω和Φ中的边权重信息φ,其余信息用于对依赖图的动态执行信息的融入,将在第2.3节具体介绍。因此,本节将系统调用依赖图简略表示为$D=(V,E,\varphi)$。算法2展示了系统调用依赖图的生成过程,可分为初始化阶段与权重优化阶段。

算法2　系统调用依赖图构造

输入: 系统调用声明Decls,系统调用描述Desc,系统调用资源使用情况Res

输出: 系统调用依赖图$D=(V,E,\varphi)$

```
1. 初始化图D=(V,E,φ)
2. FOR 每个系统调用sys∈Decls DO
3.     在V中添加节点sys
4. FOR 每对节点(a,b)∈V×V DO
5.     在E中加入边<a,b>,设置φ(<a,b>)←0
6. END FOR
7. // -------- 参数-返回值类型分析 --------
8. 定义Map:type→(retSet,paramSet)
9. FOR 每个系统调用sys∈Desc DO
10.    将sys的返回类型加入Map[type].retSet
11.    将sys的参数类型加入Map[type].paramSet
12. END FOR
13. FOR 每个类型key∈Map.keys() DO
14.    FOR i∈Map[key].paramSet,j∈Map[key].paramSet,i≠j DO
15.        φ(<i,j>)←φ(<i,j>)+1 END FOR
16.    FOR i∈Map[key].retSet,j∈Map[key].paramSet DO
17.        φ(<i,j>)←φ(<i,j>)+1 END FOR
18. END FOR
19. // -------- 内核资源使用分析 --------
20. FOR 每对系统调用(a,b)∈V×V DO
21.    IF a写资源R且b读同一资源R THEN
22.        φ(<a,b>)←φ(<a,b>)+1 END IF
23. END FOR
24. 返回D=(V,E,φ)
```

首先,在初始化阶段,根据系统调用的声明信息为每个系统调用创建对应的节点。为了避免在后续步骤中不断修改图的结构,为任意两节点a和b预先构造双向边$<a,b>$和$<b,a>$,并将其初始权重均设为0,表示两者之间尚未发现依赖关系。

在完成基础图结构的搭建后,需要通过两类信息对边权重进行完善:一类是基于系统调用"参数-返回值类型"的语义关联,另一类是基于内核资源访问模式的潜在依赖关系。

为了从类型角度刻画系统调用间的潜在依赖,引入Syzkaller提供的系统调用描述作为信息源。它基于专家知识,以声明式语言精细刻画了大量更细粒度的系统调用的参数类型、返回类型以及调用语义的约束。基于这些高质量的语义描述,对依赖图中的边权重进行第一轮调整,主要包括以下两类情况。

(1)若系统调用i与j至少共享一种参数类型,则增加<sys_i, sys_j>和<sys_j, sys_i>的权重值,增加的值为1。

(2)若系统调用i的返回类型可以作为系统调用j的某一参数类型,则意味着j的执行可能依赖于i的执行结果。对于这种关系,本文提出的方法增加依赖图中<sys_i, sys_j>的权重值,增加的值为1。

在具体数据结构上,通过构建Map的方式来提高性能,其中key为类型,value为二元组(ret, param),它们分别记录使用该类型作为返回值和作为参数的调用集合;再基于同一类型下的调用集合,按照上述规则批量调整依赖图的权重。

第二轮的权重优化依赖于系统调用对内核资源的访问模式。其核心思想可以概括为谁写资源、谁读资源,写者对读者形成潜在影响。也就是说,如果系统调用a会修改某个内核资源,而调用b在执行中会读取该资源,那么b的行为可能受a的执行结果影响,因此应提高依赖图中$e=<a,b>$的权重,增加的权重值为1。基于第2.2.2节中的每个系统调用的资源使用集合以及资源访问的读写属性,本阶段无需进行额外的代码分析工作,只需对系统调用之间的资源访问模式进行两两比对,并依据既定模式和规则在依赖图中增加相应边的权重。

为了更直观地展示依赖图的构造过程,下面以4个常见的系统调用(open、read、write和close)为例。首先根据系统调用声明得到点集$V=\{$open, read, write, close$\}$,并为任意两个系统调用之间添加有向边,初始权重均为0。随后进行"参数-返回值类型"分析,由于open的返回值类型为文件描述符fd,而read、write和close都需要fd作为参数,本文在依赖图中增加边<open, read>、<open, write>和<open, close>的权重。进一步结合内核资源使用情况,如open对资源fd进行写操作,而read、write和close对fd进行读操作,再次增加边<open, read>、<open, write>和<open, close>的权重。同时,write对缓冲区buf执行写操作,而read对buf执行读操作,因此在依赖图中增加边<write, read>的权重。最终得到的依赖图如图5所示。实际上,图5中包括自环边,但将其用于第2.3节中的模糊测试流程,因此在此处省略。另外,尽管本文静态分析方法相较于纯类型分析误报率较低,但整体误报率仍处于适中的水平。在后续动态测试过程中,进

一步结合实际运行反馈对依赖图的准确性进行持续的优化和修正。

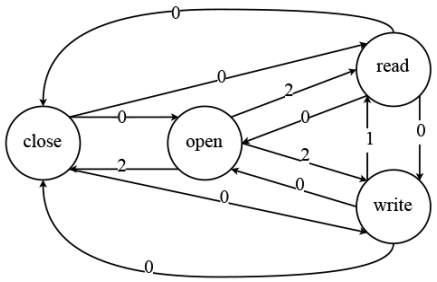


图5 系统调用依赖图示例

Figure 5 Example of system call dependency graph

2.3 融合依赖推断与权重优化的内核模糊测试

整体而言,对于本文提出的融合依赖推断与权重优化的内核模糊测试方法,依赖图在测试执行中承担两项关键功能:(1)指导测试用例的生成与变异,通过依赖关系调整系统调用序列的构造方式,使生成的序列更符合内核状态演化规律;(2)根据运行时信息对依赖图进行动态更新,利用执行反馈反映真实依赖强度,使依赖图在测试过程中不断自我修正和优化。

本文的方法框架如图6所示。首先对每个系统调用进行独立测试,为其单独生成最小化的测试用例并执行一次,将覆盖数据记录并写入依赖图中对应的节点,作为后续依赖推断与权重调整的基准。测试过程中,Fuzzer维护一个按FIFO调度的测试用例队列。当队列为空时,新的测试用例被Fuzzer主动生成并入队,基于依赖图生成一定数量、长度的系统调用序列,这些序列被直接执行以获取覆盖。当队列非空时,从队列取出用例并基于依赖图进行插入、删除、拼接、参数变异、位置翻转等操作生成新用例,再由执行器执行。每次生成或变异均更新依赖图中系统调用及依赖关系的选择次数,有效覆盖还会进一步更新其有效次数和有效比,用于周期性调整权重。

2.3.1 融入动态执行信息的系统调用依赖图

静态分析构建的依赖图能刻画两两系统调用的依赖关系与依赖程度,但缺乏全局视角下的依赖特征。为此,为图中每个节点引入以下全局属性。

(1)全局依赖权重(平均入度)μ_{sys_a},反映系统调用sys_a依赖其他调用的整体程度。

(2)全局被依赖权重(平均出度)μ'_{sys_a},反映其他调用依赖sys_a的整体程度。

计算公式为

$$\mu_{sys\_a}=\frac{\sum_{a\in(V_{syscalls}-\{sys_a\})}W_{xa}}{N-1} \tag{2}$$

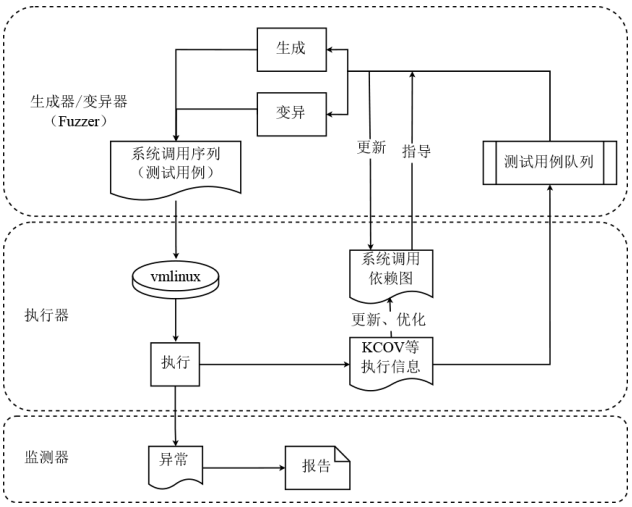


图6 融合依赖推断与权重优化的Linux内核模糊测试框架

Figure 6 Linux kernel fuzzing framework fusing dependency inference and weight optimization

$$\mu'_{sys\_a}=\frac{\sum_{a\in(V_{syscalls}-\{sys_a\})}W_{ax}}{N-1} \tag{3}$$

其中,Wxa为边<sys_x,sys_a>的权重,N为节点总数。

为了使系统调用依赖图不仅在初始构建阶段发挥作用,还能够在整个模糊测试过程中持续提供指导,我们进一步将动态执行信息融入依赖图,使其成为一个随测试进展不断演化的结构。在这一机制中,执行反馈被分为两类:一类用于更新依赖图的节点,主要反映单个系统调用在真实运行环境中的行为特征;另一类用于更新依赖图的边,重点刻画不同系统调用之间在实际执行中表现出来的依赖强度与顺序关系。

其中,记录在节点上的信息有以下两种。

(1)基准覆盖信息COVbase_a。这是单独执行sys_a时的覆盖数,用于衡量其在无依赖条件下对测试的贡献(固定值)。

(2)系统调用被选择的次数SelectCntN_a。这是在变异操作中sys_a被选中的次数。

在边上记录的信息则包含以下3个方面。

(1)系统调用依赖关系被模糊测试的突变选择的次数SelectCntE_<a,x>,指该依赖关系在变异中被“打破”的次数。

(2)系统调用依赖关系的被选择突变且突变有效的次数ValidCnt_<a,x>,指该依赖关系在一次变异执行后产生新增覆盖的次数。

(3)依赖关系有效比率ValidRatio_<a,x>,指上述两个信息的比值。

由此,定义1中四元组$D=(V,\Omega,E,\Phi)$包含了全部信息。其中,记录在节点a上的信息$\Omega(a)$和记录

在边e上的信息$\Phi(e)$均用四元组来表示,分别包含多个信息。具体的表示如下:$\Omega(a)=(\mu_{sys_a}, \mu'_{sys_a}$, COVbase_$a$, SelectCntN_$a$);$\Phi(e)=(\varphi(e)$, SelectCntE_$e$, ValidCnt_$e$, ValidRatio_$e$)。

2.3.2　依赖图指导内核模糊测试的策略

本节从测试用例生成和测试用例变异两方面来说明依赖图在内核模糊测试中的作用。在测试用例生成方面,本文提出的方法假设初始阶段不存在任何可用种子,测试队列为空,即测试用例需从零生成,其过程如算法3所示,主要过程如下。

(1)确定测试范围与序列长度。依赖图所有节点对应的系统调用均为测试目标(可选择子集)。利用测试队列的动态状态决定生成序列的预期长度。每当测试队列为空时,记录一次空队列事件,并将该计数值EmptyCnt作为当前需要生成的系统调用序列的长度。这样,序列长度会随测试进程自适应变化,而无需预先固定。

(2)确定序列起点的选择策略。在生成序列的第一个系统调用时,采用了一套避免偏置的选择机制。具体来说,优先挑选SelectCntN较低的系统调用,使得先前较少被选中的调用能获得更多测试机会。同时,优先选择平均入度μ较小的系统调用,降低其在后续生成中被过度选中的概率,防止陷入局部最优。

(3)确定后续系统调用的选择原则。以当前序列最后一个系统调用为起点,依据依赖图边的权重$\varphi(e)$和选择次数SelectCntE进行概率选择。SelectCntE越小,对应终点被选中的概率越高;边权重越大,表示该组合越可能符合内核状态并触发深层路径,其选中概率越高。为允许连续重复系统调用,每个节点添加自环边,其权重为该节点出边权重平均值。

(4)依赖图更新。每新增系统调用sys_a,都需更新其节点选择次数以及与前驱调用的边选择次数。

(5)参数合理化。依据Syzkaller系统调用声明填充参数,确保生成值符合语义与类型约束,减少无效测试空间。对运行时依赖参数(如指针)通过预构建的参数池随机选取。

(6)重复生成直至达到目标长度。最终生成的调用序列兼顾依赖关系与覆盖均衡性,且更易探索深层内核路径。

在测试用例变异方面,执行器运行测试用例后,监控器收集覆盖信息并将新增覆盖的用例加入队列,之后从队列取出用例进行依赖图指导的变异。变异流程如算法4,主要操作包括以下步骤。

(1)插入系统调用。插入点常选在序列末尾,与生成过程相似。若插入在序列开头或中间,则需结合前驱与后继调用的边权重综合计算候选调用的概率。

算法3　依赖图指导的测试用例生成

输入: 系统调用依赖图$D=(V,\Omega,E,\Phi)$,期望生成序列长度EmptyCnt
输出: 系统调用序列Seq

```
1. 初始化Seq为空序列
2. 对每个v in V,SelectCntN[v]←0
3. 对每条e in E,SelectCntE[e]←0
4. WHILE length(Seq)<EmptyCnt DO
5.   IF Seq为空 THEN
6.     候选节点←V,按SelectCntN和入度排序
7.     sys_next←从候选节点中按顺序选取一个节点
8.   ELSE
9.     sys_prev←Seq[1]
10.    E_prev←sys_prev的出边集合(含自旋边)
11.    对每条e in E_prev,W[e]←φ(e)/(1+SelectCntE[e])
12.    sys_next←按归一化W[·]概率选择终点系统调用
13.  END IF
14.    Seq.append(sys_next)
15.    SelectCntN[sys_next]+=1
16.    IF length(Seq)>1 THEN
17.      SelectCntE[(sys_prev,sys_next)]+=1
18.    END IF
19.    为sys_next生成合理参数值
20. END WHILE
21. 返回Seq
```

插入后,更新依赖图的节点和边选择次数。

(2)删除系统调用。优先删除平均出度或入度较大的调用,以打破更多依赖关系。删除后,更新对应节点与边的选择次数。

(3)拼接调用序列。从队列随机选取另一调用序列,在目标序列的拼接点后替换为该序列。此操作不更新依赖图。

(4)变异系统调用参数。优先选择参数多、类型复杂,或平均入度和出度较大的调用。必要时参考前后调用参数进行协同变异。

2.3.3　基于动态执行信息的系统调用依赖图优化

由于静态分析在进行基础分析时的局限性(如非上下文敏感的指针分析),其构建的系统调用依赖图不可避免的不准确以及不精确。为此,本文提出在模糊测试过程中融合动态执行信息,以持续优化依赖图的准确性,从而更有效地指导测试。具体包括基于依赖关系有效比率和基于覆盖反馈的权重校准两个策略。

基于依赖关系有效比率的策略如下。为提升依赖图的准确性,本文进一步引入了依赖关系有效比率作为权重调整依据,即ValidRatio。其用于刻画该依赖在真实执行过程中带来新增覆盖或触发异常的相对频度,从而反映该依赖在实际测试中的价值。其计

算法4 依赖图指导的测试用例变异

输入: 系统调用依赖图$D=(V,\Omega,E,\Phi)$,原始系统调用序列Seq
输出: 变异后的系统调用序列Seq′
在Seq中选择变异点p
随机选择变异操作op∈{Insert,Delete,Concat,ParamMutate}
IF op=Insert **THEN**
  IF p在末尾 **THEN**
    sys_new←按SelectCntE与$\varphi(e)$概率选取候选系统调用
  ELSE IF p在首部 **THEN**
    sys_new←按入边权重概率选取候选系统调用
  ELSE
    对每个候选sys_x,W[sys_x]=φ(prev,sys_x)+φ(sys_x,next)
      sys_new←按归一化$W[\cdot]$概率选择
  END IF
  在位置p插入sys_new
  更新SelectCntN,SelectCntE
ELSE IF op=Delete **THEN**
  sys_del←按平均入度/出度概率选取
  从Seq删除sys_del
  更新SelectCntN,SelectCntE
ELSE IF op=Concat **THEN**
  随机选择另一序列seq_new
  在Seq中选取拼接点q
  删除q之后的部分并追加seq_new
ELSE IF op=ParamMutate **THEN**
  sys_param←按参数复杂度与节点度概率选择某系统调用
  param←在sys_param中按复杂度选取参数
  对param执行变异
  更新SelectCntN[sys_param]
END IF
参数合理化处理
返回Seq′

算基于有效次数(ValidCnt)与依赖边被选次数(SelectCntE)的比值。这个过程的重要环节是ValidCnt更新和权重调整。

(1)ValidCnt更新是仅在“插入”或“删除”系统调用后,且该变异引发新覆盖或crash时,更新相关边的ValidCnt。判断是否产生新覆盖时,需比较变异前后覆盖数量与全局已发现覆盖。同时,还需要结合该系统调用的基准覆盖量进行过滤。

(2)权重调整策略是对起点为sys_a的所有出边,若某边的ValidRatio高于该节点的平均ValidRatio,则将其权重$\varphi(e)$增加1,否则不调整。更新完成后,重新计算节点的平均出度。

为避免频繁调整带来的性能开销,本文的优化过程采用定时触发,而非每次变异后立即执行。

除了利用有效依赖关系对边权重进行调整之外,本文还引入了一种基于覆盖反馈的权重校准策略。其核心思想是对测试序列中相邻的两个系统调用进行顺序翻转,随后比较两种执行路径所产生的覆盖差异。若翻转前后的覆盖情况出现显著变化,则说明这两个系统调用之间存在潜在的顺序约束,据此对依赖图中对应的两个反向边的权重进行重新评估与校正,使其更贴近真实执行语义。具体而言,当相邻两个系统调用的执行顺序被翻转后,如果新的执行路径带来了更多的覆盖信息,且当前依赖图中对原有顺序的权重低于其反向顺序,则表明现有权重未能准确反映二者真实的顺序关系。基于该覆盖反馈,算法通过交换这两个系统调用在依赖图中对应的两条反向边的权重,对其顺序依赖进行重新评估与校准,从而强化与实际执行行为更一致的顺序约束。

与静态分析相比,该方法更能反映两个调用在不同顺序下的实际依赖强度,但依赖于当时的内核状态和参数,存在较高的漏报率。因此,本研究仅在动态测试阶段用其修正静态分析构造的依赖图,减少权重偏差,而非替代静态方法。

基于以上两个策略,整体系统调用依赖图的权重优化过程和具体细节如算法5所示。

其中,第1~11行体现了第一个策略的内容,而

算法5 系统调用依赖图权重优化

输入: 系统调用依赖图$D=(V,\Omega,E,\Phi)$,原始系统调用序列Seq
输出: 更新后的系统调用依赖图$D'=(V,\Omega,E,\Phi)$
定时触发权重优化(非每次变异)
FOR 每个节点sys_$a\in V$的出边集合OutEdges(sys_a) **DO**
  计算每条出边e的ValidRatio(e)=ValidCnt(e)/SelectCntE(e)
  avgRatio=OutEdges(sys_a)的ValidRatio平均值
  FOR 每条出边e IN OutEdges(sys_a) **DO**
    IF ValidRatio(e)>avgRatio **THEN**
      $\varphi(e)$←$\varphi(e)$+1
    END IF
  END FOR
  重新计算节点sys_a的平均出度
END FOR
从Seq中随机选择相邻的两个系统调用(sys_i,sys_j)
翻转顺序执行seq′=Seq[...sys_j,sys_i...]
计算覆盖差异Δcov=Coverage(seq′)-Coverage(Seq)
IF Δcov>0&&φ(sys_i,sys_j)<φ(sys_j,sys_i) **THEN**
  t←φ(sys_i,sys_j),φ(sys_i,sys_j)←φ(sys_j,sys_i)
  φ(sys_j,sys_i)←t
END IF
S=所有边$e\in E$的权重之和,n=所有E的数量
FOR 每条边$e\in E$ **DO**
  $\varphi(e)$←$n\times\varphi(e)/S$
END FOR
返回更新后的$D'=(V,\Omega,E,\Phi)$

第12~18行体现了第二个策略的内容。除此之外,第19~22行在每轮更新后执行全局归一化操作,使所有权重始终保持在一定范围内,从而使得权重有界。在足够长的时间下,会优先选择次数偏低的边,因此所有边都将得到足够的探索次数,即动态更新的第一个策略依赖的ValidRatio(e)会趋近于稳定,而第二个策略也会逐渐不再满足触发条件。在此基础上,各个边的权重在动态更新过程中收敛至一个稳定的分布。具体来说,考虑一个极端的案例:现有两个系统调用a和b,其中真实依赖方向为$a\to b$,即只有执行a之后再执行b才可能产生有效语义,而其他任何执行(例如反向执行$b\to a$)完全无效。假设初始权重$\varphi(a,b)$为1,而$\varphi(b,a)$为0。在迭代过程中,两个策略会共同影响权重的变化:基于有效比率的更新和基于覆盖反馈的顺序校准。第一个策略基于有效比率,只有$a\to b$有效,因此模糊测试执行一段时间后只有该边有效比率大于0且大于对应平均值,即满足第一个策略的更新条件。同时,第二个策略的更新条件不满足。因此,在归一化之前,一次动态更新后的权重如下:$\varphi(a,b)$为2,$\varphi(b,a)$为0。而接下来,归一化算法会调整权重(其中n为4):$\varphi(a,b)$为4,$\varphi(b,a)$为0。之后多轮的动态更新与此最终结果一致,即之后其权重始终保持一种稳定的状态:$\varphi(a,b)$始终为4,而$\varphi(b,a)$始终为0。

2.3.4　权重计算与动态更新机制的合理性分析

本文的权重相关机制由静态的基于语义信息的初步构造与动态行为更新两部分组成。

首先,权重基于静态的基于语义信息的初步构造过程天然具备低敏感性特征:系统调用之间的边权来源于明确的语义关联,例如共享的参数类型共享、返回值与参数之间的关系,以及对相同内核资源的读写依赖。这些依赖关系及其重要程度均由系统调用接口本身的语义所决定,而不依赖具体测试数据,因此具备低敏感性特征。

在权重更新策略上,本文采取了一系列措施来防止陷入局部最优或过度拟合。为了防止陷入局部最优,本文基于ValidRatio增加权重,而其计算基于有效次数(ValidCnt)与依赖边被选次数(SelectCntE)的比值。一方面,当依赖边被选次数逐渐变多时,会降低其权重增加的可能性;另一方面,在测试用例生成和变异时,引入了降低选择被选次数较多的依赖边的可能性的机制,从而避免陷入局部最优,使得其转向探索其他依赖关系。另外,引入测试过程中实时覆盖反馈的权重调整,通过翻转系统调用顺序比较覆盖差异,确保权重调整考虑多种执行路径,避免由于过拟合只执行单条路径。

3　工具实现与实验

为评估本文提出的方法在实际内核环境中的表现,我们实现了一套完整测试工具SDKernelFuzzing,并将其部署在多个Linux内核上进行实验测试。

整个工具体系由静态分析与依赖图构建模块、依赖图驱动的模糊测试模块这两个主要组件构成。其中,静态分析与依赖图构建模块基于LLVM(Low Level Virtual Machine)工具链开发,其构造的初始系统调用依赖图从存储形式为JSON文件;依赖图驱动的模糊测试模块则直接基于Syzkaller扩展。

实验运行环境和配置如表5所示。

表5　实验配置说明

Table 5　Experiment configuration description

项目	配置说明
操作系统	Ubuntu 18.04
处理器(CPU)	Intel(R) Core(TM) i7-10750H 2.60 GHz
内存(RAM)	16 GB

3.1　研究问题和实验对象

本研究提出融合依赖推断与权重优化的Linux内核模糊测试方法,其核心在于依赖图的构建与使用。因此,实验设计也聚焦于依赖图本身的构造能力及其在模糊测试阶段所能带来的实际改进。从总体上看,从以下维度验证该方法:依赖图能否在合理时间内从真实内核源码中构建出来、依赖图所表达的系统调用关系是否充分,以及是否能够切实提升模糊测试效果。基于上述考虑,本文提出了4个需要回答的研究问题(RQ),用于指导后续实验与分析。

RQ1:依赖图构建阶段的性能表现如何?

RQ2:依赖图呈现的系统调用依赖关系的全面性如何?

RQ3:在模糊测试实践中,引入依赖图是否能够提升测试效率?

RQ4:系统调用依赖图中不同依赖构建策略对模糊测试有效性的边际贡献如何?

在实验对象选择上,选取了最新的支持Clang编译前端的4个长期支持的Linux内核版本,分别是v5.15、v6.1、v6.6、v6.12,均采用x86_64_defconfig配置。

3.2　RQ1:系统调用依赖图的构造效率

3.2.1　实验设计

在系统调用依赖图构建方面,将本文提出的SDKernelFuzzing静态分析模块与现有的分析工具Smatch(Moonshine)进行了对比实验。在已有的开源工具中,Moonshine[13]对系统调用间依赖关系进行了相对全面的分析。SyzGen++[17]与KernelGPT[14]并未通过静态分析构建系统调用间的依赖关系,它们本身

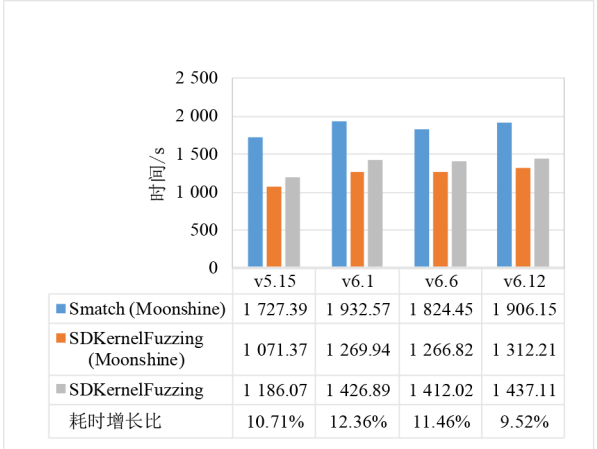
不生成系统调用依赖图，因此无法在依赖图构造效率或依赖信息全面性方面进行可比性评估。基于此，将两者排除。

为了保证端到端的公平对比，在统计分析时间时，将SDKernelFuzzing的AST构建时间计入总耗时，同时剔除了Smatch中内核构建所需的时间，从而消除了两者因编译流程差异带来的影响。鉴于GCC与Clang编译效率的差异（具体而言，Smatch基于GCC来编译，而SDKernelFuzzing基于Clang来编译），我们实现了一个对照版本SDKernelFuzzing（Moonshine）。该版本在SDKernelFuzzing中实现了Moonshine的间接依赖分析。所有实验均在单作业模式下进行，以保证一致性。

3.2.2 实验结果与分析

实验结果如图7所示，其中包括各个工具依赖图构建的时间开销和SDKernelFuzzing相较于目标对照组（即SDKernelFuzzing（Moonshine））的耗时增长比。

实验结果表明，对于所有4个Linux内核版本，Smatch（Moonshine）的静态分析所需要的时间显著高于其他工具。这主要是由于Smatch在分析过程中采用了I/O操作较多数据库存储机制，用于记录函数调用关系、变量读写状态等临时信息，从而显著增加了分析时间。而本文提出的SDKernelFuzzing分析耗时略高于其Moonshine对照版本（即SDKernelFuzzing（Moonshine））。这是因为它在静态分析中额外引入了函数指针分析，这使得其分析的逻辑更为复杂，并且能够识别更多的系统调用依赖和变量访问关系，从而在提高分析精度的同时，略微增加了总耗时（增长比维持在9%~13%范围内）。总体来看，SDKernelFuzzing在兼顾效率和依赖关系全面性方面实现了较好的平衡。

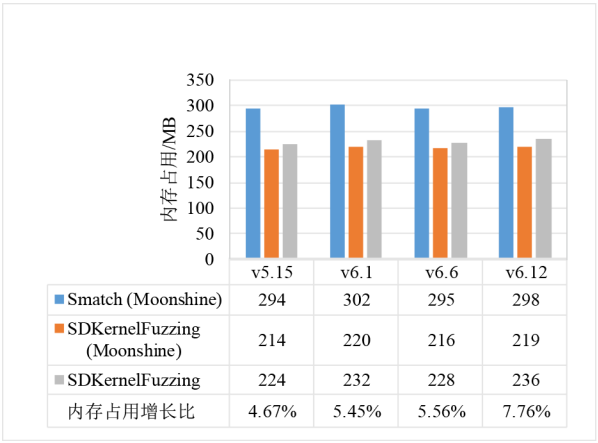


	v5.15	v6.1	v6.6	v6.12
Smatch (Moonshine)	1 727.39	1 932.57	1 824.45	1 906.15
SDKernelFuzzing (Moonshine)	1 071.37	1 269.94	1 266.82	1 312.21
SDKernelFuzzing	1 186.07	1 426.89	1 412.02	1 437.11
耗时增长比	10.71%	12.36%	11.46%	9.52%

图7 依赖图构建的时间开销统计图

Figure 7 Statistical graph of the time cost of dependency graph construction

在对依赖图构建的时间开销进行评估的同时，对不同工具构建依赖图的空间开销也进行了系统的测试。使用GNU time对各个工具在各个内核版本的完整依赖图构建阶段的峰值内存占用（Max RSS）进行记录，结果如图8所示。结果表明，在各个内核版本下，SDKernelFuzzing比对照组SDKernelFuzzing（Moonshine）的峰值内存占用只略高，不超过8%。略高的原因主要是SDKernelFuzzing包含的依赖分析更多且依赖信息更丰富，因此中间表达的内存占用较多。



	v5.15	v6.1	v6.6	v6.12
Smatch (Moonshine)	294	302	295	298
SDKernelFuzzing (Moonshine)	214	220	216	219
SDKernelFuzzing	224	232	228	236
内存占用增长比	4.67%	5.45%	5.56%	7.76%

图8 依赖图构建的空间开销统计图

Figure 8 Statistical plot of space overhead for dependency graph construction

3.3 RQ2：系统调用依赖图的依赖信息全面性

3.3.1 实验设计

本实验旨在评估依赖信息的完整性，对比Moonshine中Smatch提取的系统调用依赖与本文SDKernelFuzzing构建的依赖信息。

3.3.2 实验结果与分析

实验结果如表6所示，包含系统调用组合总数以及两个工具分析所得到的依赖组数和总比例。各个内核版本的系统调用数n是ground truth，直接统计得到，而组合数即$n\times(n-1)$。需要说明的是，这并不意味着这些系统调用对均对应真实存在的依赖关系。由于内核中真实的系统调用依赖集合在实践中难以完整观测，本文将所有有序系统调用对的组合数视为可能依赖关系的上限，并将此作为评估不同方法在依赖发现覆盖能力方面的参考基准。

此外，SyzGen++通过人工检查少量驱动源码构建ground truth，用于评估显式的、参数级别的系统调用依赖推断，该评估方式在其研究场景下是可行的。然而，本文的工作主要关注由内核状态与资源语义等所引入的间接系统调用依赖。这类依赖往往并不直接体现在系统调用接口参数中，且缺乏清晰且统一的人

表6　静态分析结果对比

Table 6　Comparison of static analysis results

内核版本	x86_64_defconfig		Moonshine		SDKernelFuzzing	
	系统调用数	两两系统调用组合数	依赖组数	占比/%	依赖组数	占比/%
v5.15	449	201 152	18 043	8.97	165 588	82.32
v6.1	451	202 950	19 223	9.47	163 868	80.74
v6.6	454	205 662	20 008	9.73	168 342	81.85
v6.12	463	213 906	19 881	9.29	172 119	80.46

工标注标准。构建完整而客观的 ground truth 不仅成本极高,而且难以保证一致性与准确性。因此,本文采用下游模糊测试效果作为主要评估指标,以间接反映所构建系统调用依赖信息的有效性。

结果表明,在所有被测内核版本上,SDKernelFuzzing 能够分析出的依赖组数和占比显著优于 Moonshine。SDKernelFuzzing 分析出的依赖组数占比平均比 Moonshine 高出 71.98%。这种差异主要源于两方面原因:首先,Moonshine 仅考虑系统调用之间的直接函数调用关系,而 SDKernelFuzzing 在分析中同时包含了间接函数调用,从而捕获了更多潜在依赖;其次,本文提出的依赖判定条件较 Moonshine 更为宽松和全面,涵盖了更多全局变量及结构体成员变量。

除此之外,如表 7 所示,SDKernelFuzzing 相比于 Moonshine 为每条依赖关系额外分配了刻画依赖强度的权重信息。由此可见,SDKernelFuzzing 构建的系统调用依赖图在依赖信息的完整性和内部属性的刻画能力方面均优于 Moonshine。

表7　系统调用依赖图的依赖信息示例

Table 7　Example of dependency information in the system call dependency graph

Moonshine	SDKernelFuzzing

{'acct': set(['accept4',
'fadvise64_64',
'memfd_create',
'mmap_pgoff',
'mq_open',
'open',
…
}
	{"clock_adjtime": {
	"adjtimex": 81,
	"clock_adjtime": 0,
	"clock_getres": 203,
	"clock_nanosleep": 215,
	"clock_settime": 70,
	…
	}

3.4　RQ3:系统调用依赖图对内核模糊测试效率的影响

3.4.1　实验设计

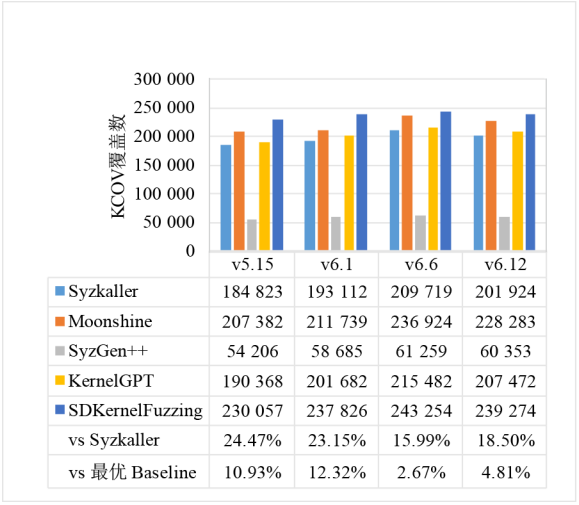
本研究问题旨在评估依赖图在模糊测试中的实际效用,基于此,本实验将本文提出的 SDKernelFuzzing 内核模糊测试模块与原始 Syzkaller、Moonshine、SyzGen++、KernelGPT 进行对比,并以 KCOV 覆盖数和 crash 信息为评价标准。

内核的构建和配置过程如下:使用的编译器为 GCC;配置为 x86_64_defconfig;启用的配置选项有 KCOV 和 KASAN,分别用于统计 KCOV 覆盖数和内存错误的检测。

实验运行方式如下:每组测试同时启动 4 台分配了 4 GB 内存和 4 个 fuzzing 进程的 QEMU 虚拟机。所有模糊测试均在无初始种子、覆盖全部系统调用的条件下进行,并持续 24 h。实验结束后,收集 KCOV 覆盖数和过滤掉非崩溃类事件的有效 crash 信息。

3.4.2　实验结果与分析

实验结果分为两类:一是代码覆盖率的对比,包括如图 9 所示的 KCOV 覆盖数对比和如图 10 所示的覆盖数增长的子系统的分布情况;二是 crash 发现数目的对比,如图 11 所示。其中,SyzGen++在该实验中的覆盖数和崩溃数量相比于其他工具较少,这是由于其设计目标并非面向整个内核的全面性 fuzzing,而是专注于支持 Linux drivers 的典型接口形式(如 open、ioctl、read、write),并围绕驱动程序特性优化其依赖推断与规格生成流程。



	v5.15	v6.1	v6.6	v6.12
Syzkaller	184 823	193 112	209 719	201 924
Moonshine	207 382	211 739	236 924	228 283
SyzGen++	54 206	58 685	61 259	60 353
KernelGPT	190 368	201 682	215 482	207 472
SDKernelFuzzing	230 057	237 826	243 254	239 274
vs Syzkaller	24.47%	23.15%	15.99%	18.50%
vs 最优 Baseline	10.93%	12.32%	2.67%	4.81%

图9　SDKernelFuzzing与相关工具KCOV覆盖数对比图

Figure 9　Comparison of KCOV coverage number between SDKernelFuzzing and related tool

从图 9 可以看出,与 Syzkaller 和其他相关工具相比,SDKernelFuzzing 在 4 个 Linux 内核版本上均取得了更高的 KCOV 覆盖数。其中,相比于 Syzkaller,所有版本的覆盖数提升均超过 15%。而与其他工具相比,在 v5.15 和 v6.1 上,SDKernelFuzzing 的覆盖数甚至比表现最优的 Baseline 仍高出 10% 以上。

为了进一步分析覆盖率提升的来源,对各个内核子系统覆盖数的平均增长情况进行了统计(相较于最优 Baseline,即 Moonshine)。图 10 展示了覆盖数有所

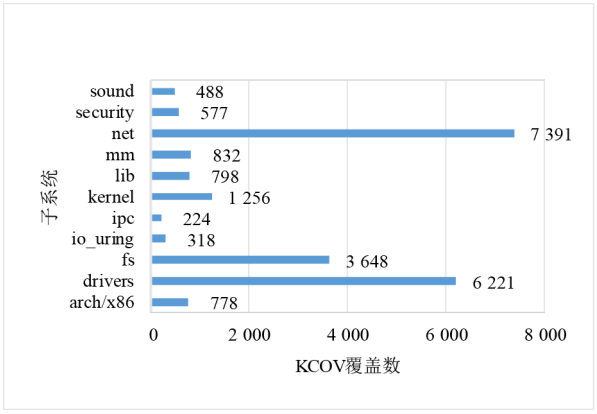


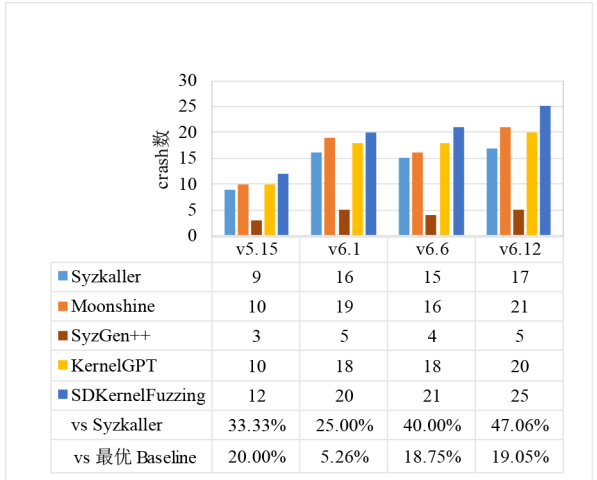
图 10 SDKernelFuzzing 的 KCOV 覆盖数平均增长的子系统的分布

Figure 10 SDKernelFuzzing's distribution of subsystems whose KCOV coverage number grows on average

增长的不同子系统的分布情况。可以看到,在若干关键子系统中,KCOV 覆盖数有显著增长。其中,网络子系统(net)增长的 KCOV 覆盖数最多,为 7 391;其次是驱动子系统(drivers,6 221)以及文件系统(fs,3 648)。这一结果表明,SDKernelFuzzing 能够有效驱动模糊测试深入资源交互复杂、潜在缺陷更多的关键代码区域,而不是产生浅层或随机的覆盖增长,从而进一步验证了依赖图驱动测试策略的有效性。

关于 crash 发现数目,从图 11 中可以观察到,与 Syzkaller 和其他相关工具相比,SDKernelFuzzing 在 4 个 Linux 内核版本上均发现了更多的 crash。其中,在 v5.15 和 v6.12 上,SDKernelFuzzing 的 crash 发现数甚至比表现最优的 Baseline 仍高约 20%。

这些结果表明,本文提出的融合依赖推断与权重优化的 Linux 内核模糊测试机制显著提高了模糊测试的执行效率与漏洞挖掘能力。



	v5.15	v6.1	v6.6	v6.12
Syzkaller	9	16	15	17
Moonshine	10	19	16	21
SyzGen++	3	5	4	5
KernelGPT	10	18	18	20
SDKernelFuzzing	12	20	21	25
vs Syzkaller	33.33%	25.00%	40.00%	47.06%
vs 最优 Baseline	20.00%	5.26%	18.75%	19.05%

图 11 SDKernelFuzzing 与相关工具 crash 数对比图

Figure 11 Comparison of crash numbers between SDKernelFuzzing and related tools

3.4.3 未知缺陷发现能力

对于每一个能够触发 crash 的测试用例,进一步在其余所有被测试的内核版本上尝试复现相同的异常行为,并在此基础上进行跨版本去重分析,最终共识别出 69 个不同的 crash。在这些 crash 中,8 个在全部 4 个内核版本中均可复现;3 个仅出现在较新的两个内核版本中;1 个仅在被测的最新稳定版本中出现;另有 1 个在中间版本中被修复但在最新版本中重新出现;其余 crash 均已在测试的最新稳定版本中得到修复。

针对仍在最高版本中存在的 crash,进一步尝试在实验开展时最新的 Linux-rc 内核版本上进行复现,结果表明其中仍有 8 个缺陷持续存在。这些未知缺陷的类型分布如表 8 所示。我们已将上述问题报告给内核开发者,截至本文撰写时,其中 Paging Request Error 中的 3 个缺陷已得到了开发者的确认,其余报告仍在进一步跟进中。

值得注意的是,这 8 个缺陷在使用其他内核模糊测试工具的测试过程中均未被发现。进一步分析表明,这主要得益于本文提出的方法对系统调用与内核资源依赖关系的融合。例如,本文发现了多处与分页处理相关的错误,而此类缺陷在现有工作的测试结果中尚未涉及。这类缺陷通常需要多个系统调用对同一内核资源进行创建、部分释放以及隐式回收等跨阶段操作,才能进入特定的分页异常处理路径。通过引入融合内核资源的依赖分析,本文提出的方法能够更有效地构造满足上述状态约束的测试序列,从而暴露出相关缺陷。

综上所述,实验结果表明,本文方法在未知缺陷发现方面能够有效补充现有内核模糊测试工具的不足,在缺陷覆盖范围上具有良好的互补性。

3.5 RQ4:系统调用依赖图中不同依赖构建策略对模糊测试有效性的边际贡献

3.5.1 实验设计

为深入探究依赖图的不同构建策略对模糊测试效果的影响,设计了一组消融实验,逐步关闭或启用依赖图构建过程中关键机制。被评估的核心机制包括间接函数调用分析、除结构体类型的全局变量外的资源类别扩展、依赖权重计算与动态更新机制。

基于此,构建以下工具变体:启用所有依赖构建策略的 SDKernelFuzzing、禁用依赖权重的计算与动态更新并保留间接调用分析和全部资源类别的变体 No-Weight、进一步禁用其他资源类别而仅保留结构体全局变量依赖的变体 Struct-Only(继续保留间接调用分析)、进一步禁用间接调用分析的变体 No-IndCall(仅保留基于结构体全局变量的直接调用依赖),以

表8　SDKernelFuzzing发现的新缺陷

Table 8　New bugs detected by SDKernelFuzzing

缺陷类型	模块	影响的函数
Slab-Out-Of-Bounds Read	drivers/tty/vt/vt.c	vcs_scr_readw
Use-After-Free	drivers/video/console/vgacon.c	vgacon_scroll
	drivers/tty/n_tty.c	n_tty_receive_buf_common
	fs/ext4/namei.c	ext4_empty_dir
General Protection Fault	fs/quota/dquot.c	dquot_add_space
Paging Request Error	mm/kasan/quarantine.c	qlist_free_all
	fs/quota/dquot.c	dquot_add_inodes
	lib/stackdepot.c	stack_depot_save

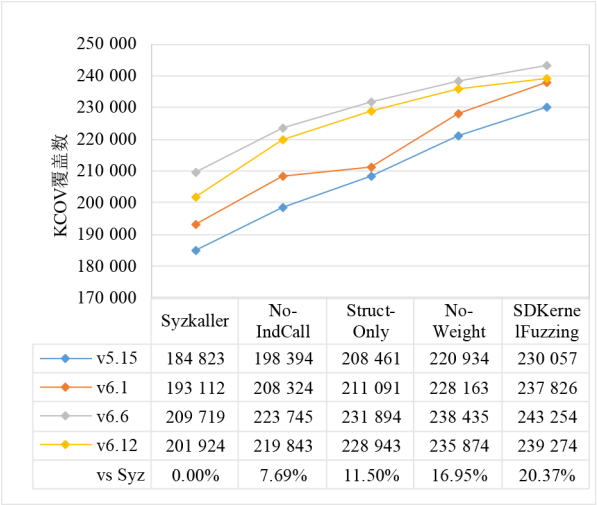
及禁用所有核心机制的方法,即原生的Syzkaller。

所有实验均在与第3.4.1节(RQ3)相同的实验环境与配置下执行,测试时间和指标保持一致。

3.5.2　实验结果与分析

消融实验结果表明,系统调用依赖图中的不同构建策略对模糊测试性能均产生了显著影响。

不同变体的KCOV覆盖数如图12所示,其中"vs Syz"表示相比于Syzkaller的提升情况。具体而言,No-IndCall(仅保留结构体全局变量依赖的直接调用分析)相比原生Syzkaller仅带来7.69%的平均提升。当进一步启用间接函数调用分析(Struct-Only)后,平均增长提高到11.50%,说明间接依赖对触发深层执行路径具有关键作用。在此基础上,启用更多类别的资源依赖(No-Weight)后平均增长进一步扩大至16.95%。最终,在此基础上加入依赖权重的计算与动态更新机制(SDKernelFuzzing完整版本),平均提升达到20.37%,表明完整依赖构建策略在提高模糊测试效率方面具有累积且显著的促进作用。

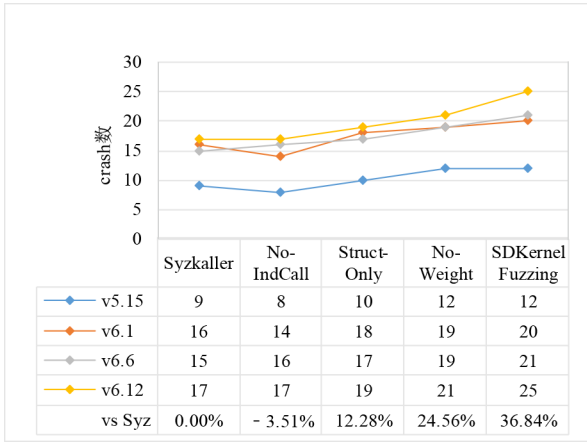


	Syzkaller	No-IndCall	Struct-Only	No-Weight	SDKernelFuzzing
v5.15	184 823	198 394	208 461	220 934	230 057
v6.1	193 112	208 324	211 091	228 163	237 826
v6.6	209 719	223 745	231 894	238 435	243 254
v6.12	201 924	219 843	228 943	235 874	239 274
vs Syz	0.00%	7.69%	11.50%	16.95%	20.37%

图12　SDKernelFuzzing不同变体的KCOV覆盖数对比图

Figure 12　Comparison of KCOV coverage numbers for different variants of SDKernelFuzzing

不同依赖构建变体在4个内核版本上的crash发现数量如图13所示,其中"vs Syz"表示相较于Syzkaller的提升情况。



	Syzkaller	No-IndCall	Struct-Only	No-Weight	SDKernel Fuzzing
v5.15	9	8	10	12	12
v6.1	16	14	18	19	20
v6.6	15	16	17	19	21
v6.12	17	17	19	21	25
vs Syz	0.00%	- 3.51%	12.28%	24.56%	36.84%

图13　SDKernelFuzzing不同变体的crash数对比图

Figure 13　Comparison of crash numbers for different variants of SDKernelFuzzing

可以观察到,随着依赖构建策略逐步完善,模糊测试对漏洞的触发能力持续增强。具体而言,No-IndCall仅保留结构体全局变量的直接调用依赖,其总体表现与Syzkaller基线接近,甚至在部分内核版本略有下降(平均–3.51%),说明仅依赖最基础的直接调用关系难以带来明显收益。当进一步加入间接调用分析(Struct-Only)后,crash数量出现实质性提升,平均增幅达到12.28%,验证了间接依赖在捕获深层代码路径方面的关键作用。继续启用结构体之外的更多资源类别(No-Weight)后,整体crash数进一步提升至24.56%,表明丰富的资源依赖类型能够有效增强模糊测试对更多执行路径的探索能力。在上述基础上,引入依赖权重的计算与动态更新机制(SDKernelFuzzing完整版本)后,crash数达到最高,平均提升36.84%,这充分表明不同类型的依赖信息在生成高质量系统调用序列方面形成了显著的累积效应。

4　结束语

本文提出了一种将静态分析构建的系统调用依赖图融入内核模糊测试的方法。同时,本文结合动态执行信息优化权重,引导生成更合法有效的系统调用序列。基于该方法实现的原型工具SDKernelFuzzing在多个Linux内核版本上的实验表明,该方法显著提高了模糊测试的执行效率和漏洞挖掘能力。

本文提出的静态分析方法在设计上避免了对人工维护规则或特定内核版本系统调用规范(如特定子系统语义)的过度依赖,而是直接基于内核源码,通过统一的资源建模方法和静态分析流程自动构建系统调用依赖图,具有较好的通用性。因此,当内核版本持续演进或引入新的系统调用与驱动代码时,相关依赖关系可通过重新分析源码自动获取,使其具备较好的可扩展性与较低的维护成本。尽管如此,其在面对持续演进的内核生态时仍存在进一步提升的空间。未来工作将重点关注对新型内核特性和编程范式的适应能力,例如更复杂的动态注册机制和跨子系统的资源共享。对于这些场景,可结合更加具有针对性的上下文信息对静态依赖推断进行校准与增强。

除此之外,该方法仍然有一些未来可以改进的其他方向。一方面,可以通过引入更精确的静态分析技术,如参数污点分析结合数据流和指针分析,以提升内核资源检测和依赖图构建的准确性;另一方面,可以采用误报较少的动态分析方法,尤其是内存访问跟踪,并结合动静态分析以降低性能开销。此外,当前方法主要聚焦于系统调用级别的依赖建模,未来工作可进一步探索与更高层语义信息的结合,例如协议状态机、子系统级行为或内核模块间的交互语义,从而在系统调用之上引入更具语义感知能力的抽象,以提升对复杂内核行为的整体刻画能力。

参考文献

[1] 涂序文, 王晓锋, 甘水滔, 等. Diskaller: 基于覆盖率制导的操作系统内核漏洞并行挖掘模型[J]. 信息安全学报, 2019, 4(2): 69-82.
Tu Xuwen, Wang Xiaofeng, Gan Shuitao, et al. Diskaller: Kernel vulnerability parallel mining model based on coverage guidance[J]. Journal of Cyber Security, 2019, 4(2): 69-82. (in Chinese)

[2] 熊忻, 谈心, 张源. 基于错误路径行为一致性的内核引用计数缺陷检测[J]. 计算机研究与发展, 2023, 60(7): 1489-1500.
Xiong Xin, Tan Xin, Zhang Yuan. Kernel refcount bug detection based on the consistency of error path behavior[J]. Journal of Computer Research and Development, 2023, 60(7): 1489-1500. (in Chinese)

[3] Renzelmann M J, Kadav A, Swift M M. SymDrive: Testing drivers without devices[C]//Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2012: 279-292.

[4] Machiry A, Spensky C, Corina J, et al. DR. CHECKER: A soundy analysis for Linux kernel drivers[C]//Proceedings of the 26th USENIX Security Symposium. Berkeley: USENIX Association, 2017: 1007-1024.

[5] Corina J, Machiry A, Salls C, et al. DIFUZE: Interface aware fuzzing for kernel drivers[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2017: 2123-2138.

[6] Bai J J, Wang Y P, Lawall J, et al. DSAC: Effective static analysis of sleep-in-atomic-context bugs in kernel modules[C]//Proceedings of the USENIX Annual Technical Conference. Berkeley: USENIX Association, 2018: 587-599.

[7] Song C Y, Lee B, Lu K J, et al. Enforcing kernel security invariants with data flow integrity[C/OL]//Proceedings of the Network and Distributed System Security Symposium, 2016. https://doi.org/10.14722/ndss.2016.23218.

[8] Erickson J, Musuvathi M, Burckhardt S, et al. Effective data-race detection for the kernel[C]//Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2010: 151-162.

[9] Zhan D Y, Yu X Z, Zhang H L, et al. ErrHunter: Detecting error-handling bugs in the Linux kernel through systematic static analysis[J]. IEEE Transactions on Software Engineering, 2023, 49(2): 684-698.

[10] 郑臣明, 姚宣霞, 周芳, 等. 基于硬件虚拟化的云服务器设计与实现[J]. 工程科学学报, 2022, 44(11): 1935-1945.
Zheng Chenming, Yao Xuanxia, Zhou Fang, et al. Design and implementation of a cloud server based on hardware virtualization[J]. Chinese Journal of Engineering, 2022, 44(11): 1935-1945. (in Chinese)

[11] Tan X, Zhang Y, Lu J D, et al. SyzDirect: Directed greybox fuzzing for Linux kernel[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2023: 1630-1644.

[12] Vyukov D, Konovalov A. Syzkaller: An unsupervised coverage-guided kernel fuzzer[EB/OL]. [2026-01-22]. https://github.com/google/syzkaller.

[13] Pailoor S, Aday A, Jana S. Moonshine: Optimizing OS fuzzer seed selection with trace distillation[C]//Proceedings of the 27th USENIX Security Symposium. Berkeley: USENIX Association, 2018: 729-743.

[14] Yang C Y, Zhao Z J, Zhang L M. KernelGPT: Enhanced kernel fuzzing via large language models[C]//Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2025: 560-573.

[15] Jeong D R, Kim K, Shivakumar B, et al. Razzer: Finding kernel race bugs through fuzzing[C]//IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2019: 754-768.

[16] Kim K, Jeong D R, Kim C H, et al. HFL: Hybrid fuzzing on the Linux kernel[C]//Proceedings of the Network and Distributed Systems Security. San Diego: Internet Society, 2020: 24018.

[17] Chen W T, Hao Y, Zhang Z, et al. SyzGen++: Dependency inference for augmenting kernel driver fuzzing[C]// Proceedings of the IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2024: 4661-4677.

[18] Han H, Cha S K. IMF: Inferred model-based fuzzer[C]// Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2017: 2345-2358.

[19] Lu S B, Lin Z H, Zhang M. Kernel vulnerability analysis: A survey[C]//Proceedings of the IEEE Fourth International Conference on Data Science in Cyberspace. Piscataway: IEEE, 2019: 549-554.

[20] Li D, Chen H. FastSyzkaller: Improving fuzz efficiency for Linux kernel fuzzing[J]. Journal of Physics: Conference Series, 2019, 1176(2): 022013.

[21] Torvalds L, Triplett J, Li C, et al. Sparse[EB/OL]. [2026-01-22]. https://sparse.docs.kernel.org/en/latest/.

[22] Brown N. Smatch: Pluggable static analysis for C[EB/OL]. [2026-01-22]. https://lwn.net/Articles/691882/.

[23] De S C, Wall J, Martinez T, et al. Coccinelle: A program matching and transformation tool for systems code[EB/OL]. [2026-01-22]. https://coccinelle.gitlabpages.inria.fr/website/.

[24] Vaughan-Nichols S. Commit 1 million: The history of the Linux kernel[EB/OL]. [2026-01-22]. https://www.zdnet.com/article/commit-1-million-the-history-of-the-linux-kernel/.

[25] Black Duck Software, Inc. Coverity Scan: Static analysis[EB/OL]. [2026-01-22]. https://scan.coverity.com/.

[26] 石剑君, 计卫星, 石峰. 操作系统内核并发错误检测研究进展[J]. 软件学报, 2021, 32(7): 2016-2038.
Shi Jianjun, Ji Weixing, Shi Feng. Recent progress of concurrency bug detection in operating system kernels[J]. Journal of Software, 2021, 32(7): 2016-2038. (in Chinese)

[27] Zhang B W, Chen W, Yao P S, et al. SIRO: Empowering version compatibility in intermediate representations via program synthesis[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2024: 882-899.

[28] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks[C]//Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York: ACM, 2003: 237-252.

[29] Wang X, Chen H G, Jia Z H, et al. Improving integer security for systems with KINT[C]//Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2012: 163-177.

[30] Xu M, Qian C X, Lu K J, et al. Precise and scalable detection of double-fetch bugs in OS kernels[C]//IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2018: 661-678.

[31] Bai J J, Lawall J, Chen Q L, et al. Effective static analysis of concurrency use-after-free bugs in Linux device drivers[C]// Proceedings of the USENIX Annual Technical Conference. Berkeley: USENIX Association, 2019: 255-268.

[32] 钱振江, 刘永俊, 姚宇峰, 等. 微内核架构内存管理的形式化设计和验证方法研究[J]. 电子学报, 2017, 45(1): 251-256.
Qian Zhenjiang, Liu Yongjun, Yao Yufeng, et al. Research on method of formal design and verification of memory management based on microkernel architecture[J]. Acta Electronica Sinica, 2017, 45(1): 251-256. (in Chinese)

[33] Cai Y D, Zhang C. A cocktail approach to practical call graph construction[J]. Proceedings of the ACM on Programming Languages, 2023, 7(OOPSLA2): 1001-1033.

[34] Sui Y L, Ye D, Xue J L. Static memory leak detection using full-sparse value-flow analysis[C]//Proceedings of the 2012 International Symposium on Software Testing and Analysis. New York: ACM, 2012: 254-264.

[35] Shi Q K, Xiao X, Wu R X, et al. Pinpoint: Fast and precise sparse value flow analysis for million lines of code[C]// Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2018: 693-706.

[36] Sui Y L, Xue J L. SVF: Interprocedural static value-flow analysis in LLVM[C]//Proceedings of the 25th International Conference on Compiler Construction. New York: ACM, 2016: 265-266.

[37] Wang J Y, Wang Y, Wang K, et al. SILVA: A scalable incremental layered sparse value-flow analysis[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(8): 1-40.

[38] Liang H L, Pei X X, Jia X D, et al. Fuzzing: State of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.

[39] Manès V J M, Han H, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2021, 47(11): 2312-2331.

[40] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

[41] Liu J Z, Shen Y H, Xu Y R, et al. Leveraging binary coverage for effective generation guidance in kernel fuzzing[C]// Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2024: 3763-3777.

[42] Song D, Hetzelt F, Das D, et al. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary[C]//Proceedings of the Network and Distributed Systems Security. San Diego: Internet Society, 2019: 23176.

[43] 郑浩然, 白家驹, 张涔, 等. 一种基于推断-验证模式的内核数据竞争检测方法[J]. 电子学报, 2025, 53(10): 3593-3607.
Zheng Haoran, Bai Jiaju, Zhang Cen, et al. A kernel data race detection method based on inference-verification mode[J]. Acta Electronica Sinica, 2025, 53(10): 3593-3607. (in Chinese)

[44] 曹鹤玲, 刘昱, 韩栋. 基于自注意力机制神经机器翻译的软件缺陷自动修复方法[J]. 电子学报, 2024, 52(3): 945-956.
Cao Heling, Liu Yu, Han Dong. Self-attention neural machine translation for automatic software repair[J]. Acta Electronica Sinica, 2024, 52(3): 945-956. (in Chinese)

[45] 苏越阳, 姚迪, 毕经平. 基于噪声标签重加权的车辆轨迹异常检测方法[J]. 电子学报, 2025, 53(1): 182-192.
Su Yueyang, Yao Di, Bi Jingping. A vehicle trajectory anomaly detection method based on noise label re-weighting[J]. Acta Electronica Sinica, 2025, 53(1): 182-192. (in Chinese)

作者简介

张 弋 女, 2001 年 7 月出生于河南省新乡市。现为南京大学计算机学院软件工程组博士研究生。主要研究方向为软件分析与测试、编译器等。
E-mail: yi@smail.nju.edu.cn

王 豫 男, 1991 年 10 月出生于河南省商丘市。现为南京大学计算机学院准聘助理教授, CCF 系统软件专委会委员、CCF 软件工程专委会委员。主要研究方向为智能化软件分析测试。
E-mail: yuwang_cs@nju.edu.cn

王林章 男, 1973 年 4 月出生于江苏省建湖县。现为南京大学计算机学院教授、博士生导师, 计算机软件新技术全国重点实验室主任助理, 中国计算机学会理事、会士, 入选国家级高层次人才。主要研究方向为软件工程、软件安全、软件架构、智能化基础软件可信保障。
E-mail: lzwang@nju.edu.cn