一种基于推断-验证模式的内核数据竞争检测方法

郑浩然,白家驹*,张 涔,关振宇
(北京航空航天大学网络空间安全学院,北京 100191)

摘 要: 数据竞争是操作系统内核中最隐蔽且危害最严重的并发问题之一. 当两个或多个内核执行线程在缺少适当同步机制的情况下,并发访问同一块共享内存,且至少有一个访问是写操作时会导致数据竞争. 数据竞争会引发数据损坏、逻辑错误和内核崩溃,甚至可能被攻击者利用构造提权或拒绝服务攻击. 因此,在操作系统开发与测试阶段,设计高效且精准的数据竞争检测机制,对保障系统的稳定性和安全性至关重要. 然而,内核并发环境的复杂性与不确定性为数据竞争的检测带来了巨大挑战,现有的动态检测方法因需要追踪锁集或发生序关系、检测严重依赖内核自发产生的线程交错,存在性能开销大、复杂问题发现能力弱等局限性,严重影响了数据竞争检测的效率与准确性. 为解决上述挑战,本文提出了一种基于推断-验证模式的内核数据竞争检测方法 RIV(Racepair Inference-Validator). RIV 的核心思想源是"推断-验证"检测模式. RIV 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对进行定向验证,以实现数据竞争的精确检测与复现. 同时,RIV 利用静态污点分析识别潜在共享变量,以减少被插桩代码量和降低运行性能开销;并通过采集变量访问的内存地址和发生时间,确保数据竞争检测准确度. 为了验证 RIV 的有效性,本文在 6 款广泛使用的 Linux 内核模块上进行了实验评估,发现了 31 个真实的数据竞争且没有被误报,其中 12 个被 Linux 内核开发者确认. 在性能对比方面,相比现有内核数据竞争检测方法 KCSAN、DILP 及 SDILP,RIV 分别提升 1.5 倍、6.7 倍与 1.8 倍性能,并基于独特的推断-验证机制发现了更多真实的数据竞争,证明了其在解决复杂竞争发现能力弱这一核心问题上的突破. 综上所述,RIV 为操作系统内核的并发安全提供了一种高效、精准且实用的自动化检测方案.

关键词: 操作系统内核;数据竞争检测;动态分析;观测点采样
基金项目: 国家自然科学基金(No.62572021);北京市自然科学基金(No.4252019)
中图分类号: TP316 **文献标识码:** A **文章编号:** 0372-2112(2025)10-3593-15
电子学报 URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20250792

A Kernel Data Race Detection Method Based on Inference-Verification Mode

ZHENG Hao-ran, BAI Jia-ju*, ZHANG Cen, GUAN Zhen-yu
(*School of Cyber Science and Technology*, *Beihang University*, *Beijing* 100191, *China*)

Abstract: Data race is one of the most critical concurrency issues in operating system kernels. A data race occurs when two or more kernel execution threads concurrently access the same shared memory location without proper synchronization, and at least one of the accesses is a write. Data races can cause data corruption, logical errors, and kernel crashes, and can even be exploited by attackers to construct privilege escalation or denial-of-service attacks. Thus, designing efficient and precise data race detection mechanisms during the operating system development and testing phases is crucial for ensuring system stability and security. However, the complexity and non-determinism of the kernel's concurrent environment pose significant challenges to data race detection. Existing dynamic detection methods suffer from limitations such as high runtime overhead and a weak capability of finding complex races, as they need to track locksets or happens-before relationships and rely heavily on spontaneous thread interleaving produced by the kernel. These issues severely impact the efficiency and accuracy of data race detection. To address these challenges, this paper proposes RIV (Racepair Inference-Validator), a kernel data race detection method based on an "inference-verification" model. The core idea of RIV is its "inference-verification" detection model: RIV first infers potential racy variable pairs by analyzing thread execution traces and

收稿日期:2025-09-10;录用日期:2025-10-20;责任编辑:覃怀银
*通讯作者:白家驹

memory access patterns, and then performs directed verification of these potential racy variable pairs through memory watchpoints and delayed injection, to achieve precise detection and reliable reproduction of data races. Meanwhile, RIV uses static taint analysis to identify potential shared variables, reducing code instrumentation and decrease runtime overhead. By capturing memory addresses and timing information of variable accesses, RIV can ensure high detection accuracy. To validate the effectiveness of RIV, we conducted experimental evaluations on 6 widely used Linux kernel modules. We discovered 31 real data races with no false positives, 12 of which have been confirmed by Linux kernel developers. In performance comparisons, RIV demonstrated performance improvements of 1.5 times, 6.7 times, and 1.8 times over existing kernel data race detection methods KCSAN, DILP, and SDILP, respectively. Furthermore, based on its unique "inference-verification" model, RIV discovered more real data races, proving its breakthrough in addressing the core problem of a weak ability to find complex races. In conclusion, RIV provides an efficient, precise, and practical automated detection solution for the concurrency security of operating system kernels.

Key words: operating system kernel; data race detection; dynamic analysis; watchpoint sampling

Foundation Item(s): National Natural Science Foundation of China (No.62572021); Beijing Natural Science Foundation (No.4252019)

1 引言

数据竞争(data race)作为并发编程中的典型安全隐患,在操作系统内核中也普遍存在并且会带来严重危害. 当两个或多个并行执行的内核线程或中断处理程序在缺乏显式同步机制的情况下,对同一共享内存区域进行非原子化访问,其中至少包含一次写操作,即构成数据竞争条件. 这种现象可能导致不可预测的程序行为,包括内存状态不一致及安全边界突破等严重后果. 若竞争的数据点能够影响程序的数据流或控制流,则可能产生严重的安全漏洞,如内核提权等. 许多研究已经表明,数据竞争问题已经是重要的内核漏洞产生原因之一[1~4]. 例如,"Dirty Cow"[5]漏洞利用Linux子系统对写时拷贝机制并发竞争处理的缺陷来提升权限;"Dirty Pipe"[6]漏洞基于Linux内核管道缓冲区的未初始化,通过并发的方式修改标志位以实现权限提升. 除此之外,还有CVE-2022-0185[7]与CVE-2025-21998[8]等都是数据竞争导致的内核漏洞,可造成操作系统崩溃、数据非法访问等严重后果.

针对上述问题,研究者们提出了多种方法来检测数据竞争. 部分研究基于静态分析(static analysis)[9~18]的方式检测数据竞争,静态分析基于源程序代码或中间表示构建控制流图(control flow graph)和程序依赖图(program dependence graph)进行分析,但是由于流分析的不准确性以及缺乏运行时信息,静态分析检测数据竞争通常误报率较高. 为了减少误报率,一些学者使用动态分析(dynamic analysis)的方式检测数据竞争,具体利用基于编译时插桩技术来实时收集分析程序的运行时信息,具体技术包括锁集分析(lockset analysis)[19~23]、发生序分析(happens-before analysis)[24~28]、观测点采样分析(watchpoint sampling analysis)[29~31]等. 然而这些动态分析方法存在性能开销大、复杂问题发现能力弱等局限性,严重影响了数据竞争检测的效率与准确性. 为了覆盖难以触发的并发情况,一些学者提出了使用模糊测试[32~37]检测数据竞争,通过描述线程并发覆盖度来指导测试用例生成,以发现常规测试难以找到的数据竞争. 但是,对于给定测试用例,这些模糊测试方法通常使用传统动态数据竞争检测技术,难以充分发挥每个测试用例在数据竞争检测方面的能力.

为了解决传统动态数据竞争分析技术的局限性,本文提出了一种基于推断-验证模式的内核数据竞争检测方法RIV(Racepair Inferencer-Validator). 相较于静态分析方法,RIV结合了运行时信息,具有无误报的特点. 相较于传统的动态分析方法,RIV无需对锁关系、发生序关系进行建模,同时不依赖随机采样,而是以推断的方式查找潜在的竞争变量对,并通过延时注入定向验证,具有"定向"这一核心优势.

RIV包含插桩-推断-验证三个工作阶段. 首先,RIV利用静态污点分析对每个函数中的变量传递情况进行分析,识别与全局变量和函数参数相关的变量访问,作为潜在共享变量访问. 其次,RIV在内核模块运行过程中实时收集线程执行情况与变量访问信息,并写入日志文件,并通过并行机制和信号控制来减少日志收集开销;在内核模块运行完成后,RIV对日志文件进行分析,如果两个变量访问va_x和va_y的内存地址一样、所在线程不同、发生时间相近、至少一个是写操作,则RIV会将其推断为潜在竞争变量对$<va_x, va_y>$,并进行记录每个变量访问发生的上下文信息. 最后,RIV再次运行内核模块,通过内存观测点与延时注入方式对每个潜在竞争变量对$<va_x, va_y>$进行定向验证. 具体来讲,如果变量访问va_x先发生,则启动va_x的内存观测点来注入延时等待,若在延时等待过程中变量访问va_y发生且两者的内存地址一样,则$<va_x, va_y>$被验证为真实发生的数据竞争;若延时等待大于设定的阈值(默认是500 ms)且变量访问va_y一直没有发生,证明$<va_x, va_y>$被验证

为难以真实发生的数据竞争,则其被判断为误报情况并被自动过滤. 通过上述步骤,RIV可实现内核数据竞争的“零误报”检测,并通过运行信息推断和定向位置延时注入方式来扩大数据竞争检测范围,从而提升复杂问题发现能力.

本文基于LLVM(Low Level Virtual Machine)编译框架实现RIV方法,并在Linux操作系统中进行了适配应用,可实现内核数据竞争的高效精确检测. 总体来讲,本文的主要贡献如下:

(1)本文详细分析了现有动态数据竞争检测技术在内核代码分析方面的局限性,并提出了一种推断-验证的新型数据竞争检测模式,从而提升复杂数据竞争的发现能力.

(2)基于上述推断-验证模式,本文设计实现了一种内核数据竞争检测方法RIV,并利用并行机制和信号控制来减少性能开销,且通过内存观测点与延时注入方式来实现数据竞争的“零误报”检测.

(3)本文利用RIV对6款常用的Linux内核模块进行了实际测试,发现了31个真实的数据竞争,其中12个已被Linux内核开发者确认和修复;相比现有内核数据竞争方法KCSAN(Kernel Concurrency SANitizer)、DILP(Detecting Inconsistent Lock Protection)与SDILP(Static-Dynamic Inconsistent Lock Protection),RIV分别提升了1.5倍、6.7倍与1.8倍性能,并发现了更多真实的数据竞争.

2　背景

2.1　数据竞争

数据竞争是一种常见的程序并发问题. 两个并发执行的线程在访问同一个共享变量时,缺少必要的同步,且至少一个是写操作,会导致数据竞争错误[38]. 在操作系统内核中,数据竞争会造成内存访问时出现数据不一致现象,进一步可能导致内核未定义行为、系统崩溃等严重危害[39]. 为了避免数据竞争和保证并发正确性,内核通常会使用互斥锁、内存屏障、原子操作等同步原语来防止两个线程在同一时刻访问同一变量. 实际上,由于在内核中频繁使用同步操作会造成较大的系统性能开销,很多内核模块会尽可能减小程序的临界区以减少同步操作,因此会带来数据竞争发生的风险.

2.2　数据竞争实例

图1展示了一个Linux内核USB驱动程序中由数据竞争引发的空指针解引用漏洞CVE-2023-52855[39]. 在驱动程序实际运行过程中,函数_dwc2_hcd_urb_enqueue()与_dwc2_hcd_urb_dequeue()会并发执行. 当线程1执行函数_dwc2_hcd_urb_dequeue()时,在第4 812行中持有自旋锁的情况下,读取并判断urb->hcpriv是否为空指针;当线程2执行函数_dwc2_hcd_urb_enqueue()时,在第4 773行中未持有自旋锁的情况下,将urb->hcpriv赋值为空指针. 因此,这两处读写操作之间会发生数据竞争. 在函数_dwc2_hcd_urb_dequeue()中的第4 816行会调用dwc2_hcd_urb_dequeue并传入参数urb->hcpriv进行解引用操作,因此该数据竞争会引发空指针解引用漏洞,从而导致驱动程序和整个操作系统发生崩溃. 由于数据竞争触发的关键在于_dwc2_hcd_urb_enqueue()与_dwc2_hcd_urb_dequeue()函数间urb->hcpriv相关指针操作的并发交错执行,此窗口期极短且依赖线程调度器行为,常规动态测试工具难以引发和捕捉此并发执行情况,导致该数据竞争难以被现有动态测试工具发现.

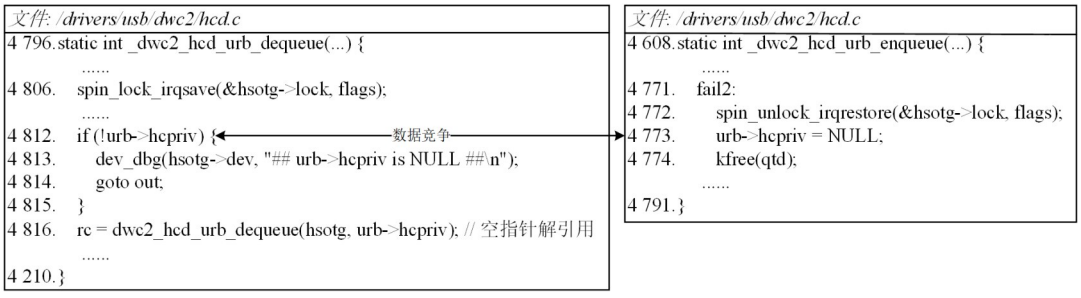


图1　Linux内核网络协议的已知数据竞争

2.3　现有的数据竞争检测方法

现有的数据竞争检测方法主要可归为静态分析[9~18]和动态分析[19~37]两大类. 由于静态分析的误报率较高,部分研究选择使用动态分析技术检测数据竞争,该方法利用编译时插桩机制实时收集程序的运行时信息进行检测. 然而,动态分析方法如锁集分析[19~23]与发生序分析[24~28]需要监控每次内存访问与同步操作,观测点采样分析[29~31]等选择性采样技术漏报率较高,表现出动态分析技术普遍存在性能开销显著和复杂问题检测能力有限等局限性,严重制约了数据竞争检测的效率与准确性. 为覆盖难以触发的并发场景,模糊测试[32~37]技术被引入用于数据竞争检测,其通过对

线程并发覆盖度的描述指导测试用例生成，以期发现常规测试难以覆盖的数据竞争；但需指出，对于生成测试用例，这些模糊测试方法往往仅采用传统动态数据竞争检测技术，针对单个测试用例，若在当次执行的线程调度未能暴露该竞争，仍然会导致数据竞争的遗漏，现有的模糊测试方法未能充分挖掘每个测试用例在数据竞争检测方面的能力.

具体到操作系统内核的数据竞争动态检测，现有方法主要依赖锁集分析与观测点采样分析两种方法；至于发生序分析，由于追踪和推断内核中变量访问发生序关系需要高昂的时空开销，因此尚未有方法将其应用于内核数据竞争检测中. 通过对典型的内核锁集分析与观测点采样分析方法进行研究，本文总结出了现有内核数据竞争动态检测方法存在的两大重要局限性：

(1)基于锁集分析的检测方法(如 DILP[20] 与 SDILP[40])在程序运行时需要监控所有变量访问操作和动态维护锁集，因此需要较大的时间开销；而这些时间开销会严重影响内核线程的调度行为，导致一些时序敏感的测试负载无法正常工作，从而降低了数据竞争检测的效率.

(2)基于观测点采样分析的检测方法(如 KCSAN[31])在程序运行时向变量访问位置随机注入延时来检测数据竞争，因此采样随机性会导致检测过程具有较强的盲目性，使得很多不常发生的复杂并发况难以被有效覆盖，从而遗漏了不少深层次数据竞争.

3 RIV方法设计及实现

3.1 RIV基本思想和相关概念定义

为了解决现有内核数据竞争检测方法的局限性，本文提出了一种基于推断-验证模式的内核数据竞争检测方法 RIV，其基本思想如下：首先，分析程序的线程执行情况与内存访问信息，来推断可能发生数据竞争的变量访问对(简称“潜在竞争变量对”)；然后，在程序运行时通过内存观测点与延时注入方式对潜在竞争变量对进行定向验证，以实现数据竞争的精确检测与复现. 相比 DILP、SDILP 和 KCSAN 等现有内核数据竞争检测方法，RIV 在内核代码插桩量和运行时信息采集量方面更少，并利用并行机制和信号控制来提升运行时信息采集性能，从而可以实现更小的性能开销；同时，RIV 在推断阶段能够识别很多不常发生的潜在复杂数据竞争，并在验证阶段定向检查这些潜在数据竞争的真实性，从而发现更多深层次数据竞争.

下面利用形式化方式对 RIV 用到的相关概念定义进行描述.

定义 1 竞争变量对. 令线程集合为 $\mathrm{TID}=\{\mathrm{tid}_1,\mathrm{tid}_2,\cdots,\mathrm{tid}_i,\cdots,\mathrm{tid}_n\}$，其中 tid_i 为变量所在线程的线程号，各线程 tid_i 的内存访问序列定义为 $\mathrm{VA}_i=\langle \mathrm{va}_{i1},\mathrm{va}_{i2},\cdots,\mathrm{va}_{ij},\cdots,\mathrm{va}_{im}\rangle$，其中 va_{ij} 为针对内存访问定义的六元组事件，详见定义 3. 在访问序列中，i 代表变量访问对应的线程索引编号，j 表示在该线程中是第几次变量访问. 当且仅当满足以下条件时，称二元组 $\langle h(\mathrm{va}_{pq}),h(\mathrm{va}_{rs})\rangle$ 构成竞争变量对，简写为 $\langle \mathrm{va}_{pq},\mathrm{va}_{rs}\rangle$.

(1)地址一致性. va_{pq} 与 va_{rs} 指向相同物理地址，即 $\mathrm{addr}(\mathrm{va}_{pq})==\mathrm{addr}(\mathrm{va}_{rs})$.

(2)线程互斥性. $\mathrm{tid}_p\neq\mathrm{tid}_r$，即访问来自不同线程.

(3)时序邻近性. $\exists\Delta t<\tau$，使得 $\left|\mathrm{ts}(\mathrm{va}_{pq})-\mathrm{ts}(\mathrm{va}_{rs})\right|<\Delta t$，其中 τ 为预设时间窗口阈值.

该定义可形式化表述为

$$\begin{aligned}\mathrm{RacePair}=&\langle h(\mathrm{va}_{pq}),h(\mathrm{va}_{rs})\rangle\\&\left|\mathrm{addr}(\mathrm{va}_{pq})==\mathrm{addr}(\mathrm{va}_{rs})\right.\\&\wedge\mathrm{tid}_p\neq\mathrm{tid}_r\\&\wedge\left|\mathrm{ts}(\mathrm{va}_{pq})-\mathrm{ts}(\mathrm{va}_{rs})\right|<\Delta t\end{aligned}\tag{1}$$

其中，$\mathrm{addr}(\cdot)$ 为变量访问的内存地址；$\mathrm{ts}(\cdot)$ 为变量访问的高精度时间戳，具体详见定义 3；$h(\cdot)$ 为基于变量访问上下文的哈希生成函数，详见定义 2.

竞争变量对的采集与分析流程如图 2 所示，采集流程分为两阶段：

(1)编译时插桩阶段. 通过静态污点分析，识别可能的数据竞争点位，对相应的变量读写点进行插桩.

(2)运行时采集阶段. 通过桩点的回调函数捕获候选潜在竞争变量对，记录其哈希标识与时间戳.

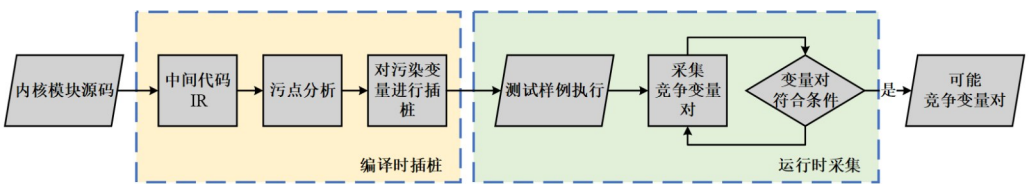


图 2 竞争变量对采集与分析流程

定义 2 变量访问上下文. 为了唯一标识程序执行过程中的变量，本文基于变量访问序列定义了变量访问上下文. 给定线程 tid_i 的访问序列 $\mathrm{VA}_i=\langle \mathrm{va}_{i1},\mathrm{va}_{i2},\cdots,\mathrm{va}_{ik}\rangle$，其第 k 次访问的上下文哈希计算值为

$$h(\mathrm{va}_{ik}) = \mathrm{SHA}_{256}(\mathrm{VA}_i[1:k]) \tag{2}$$

该哈希值通过访问插桩点进行更新，可唯一标识特定执行路径下的内存访问操作．相较于静态PC值，此方法能够消除因循环/递归导致的指令混淆，同时能够基于地址一致性快速排除非竞争访问．其中变量访问 va_{ik} 为针对内存访问定义的六元组事件，具体详见定义3.

定义3　访问记录元组．每个内存访问事件可形式化描述为六元组：

$$\mathrm{va} = \langle \mathrm{name}, \mathrm{addr}, \mathrm{ts}, \mathrm{tid}, T, C \rangle \tag{3}$$

其中，$\mathrm{name} \in \overset{*}{\sum}$ 为符号化变量标识，通过编译时插桩传入，用于识别当前访问指令符号；$\mathrm{addr} \in N_{64}$ 为访问的物理/虚拟内存地址，在运行时分配，用于判断当前访问是否与其他访问冲突；$\mathrm{ts} \in R$ 为高精度时间戳，基于内核接口实现，实现纳秒级精度采集；$\mathrm{tid} \in N_{16}$ 为全局唯一线程标识，用于标记当前访问指令所在线程；T 为类型描述结构体，包含 $T = (\mathrm{base}_{\mathrm{type}}, \mathrm{struct}_{\mathrm{name}}, \mathrm{size}, \mathrm{align})$，其中 $\mathrm{base}_{\mathrm{type}}$ 为变量访问的类型（包括读与写两种），$\mathrm{struct}_{\mathrm{name}}$ 为变量名称或结构体名称（具体通过插桩识别传入），size为变量大小，align为变量的对齐方式；C 为执行上下文指纹，定义为 $C = \mathrm{SHA}_1(\mathrm{CallStack}[1:6] \| \mathrm{Preempt}_{\mathrm{count}} \| \mathrm{IRQ}_{\mathrm{flags}})$，其中 $\mathrm{CallStack}[1:6]$ 是当前线程执行的调用栈内的最近6个函数，$\mathrm{Preempt}_{\mathrm{count}}$ 为当前线程中断数，$\mathrm{IRQ}_{\mathrm{flags}}$ 为当前线程的中断情况．

在后文表示中，由于推断过程中需要表明两个变量来自不同的线程的不同访问序列，将内存访问事件表示为 va_{ik}；当推断完毕，变量访问组成潜在竞争变量对后，仅需区分不同的内存访问事件时，将内存访问事件简写为 va_x，其中 x 代表内存访问事件的标号．

3.2　RIV方法框架

基于以上定义，本文提出了一种基于推断-验证模式的内核数据竞争检测方法RIV，其核心技术路线由编译时插桩、运行时潜在竞争变量对采集推断与定向验证三个阶段构成．该框架通过LLVM工具链实现多层次代码注入，具体技术实现流程如下：首先利用Clang前端将C/C++源程序转换为LLVM中间表示（Intermediate Representation，IR），在IR层面实施控制流敏感的过程间插桩，注入包含内存地址、线程上下文及访问类型的元数据采集点，生成可重定位的插桩目标文件；其次在运行时潜在竞争变量对采集推断阶段，通过可加载内核模块（Linux Kernel Module，LKM）机制实现运行时观测点动态部署，采用无锁环形缓冲区管理日志写入流；最终在定向验证阶段，使用延时注入技术在推断出的竞争变量点处注入延时，以验证目标点位的数据竞争存在情况．具体而言，RIV框架包含下面三个阶段：

（1）代码编译与插桩阶段．基于Clang与LLVM在编译时使用静态污点分析查找可能存在的竞争变量点，并对变量点进行插桩，最终生成可执行的被测程序．

（2）潜在竞争变量对推断阶段．在运行时收集被测程序的变量访问信息并推断潜在竞争变量对．通过回调的方式在变量访问时将变量访问信息写入日志，在用户态对日志中的变量访问进行分析，基于定义提取出潜在竞争变量对．

（3）竞争变量对验证阶段．基于延时注入技术对分析出的潜在竞争变量对进行逐个验证．通过向变量访问桩点注入延时，并通过动态观测点检测技术来判断是否真实发生了数据竞争．

RIV整体流程如图3所示．

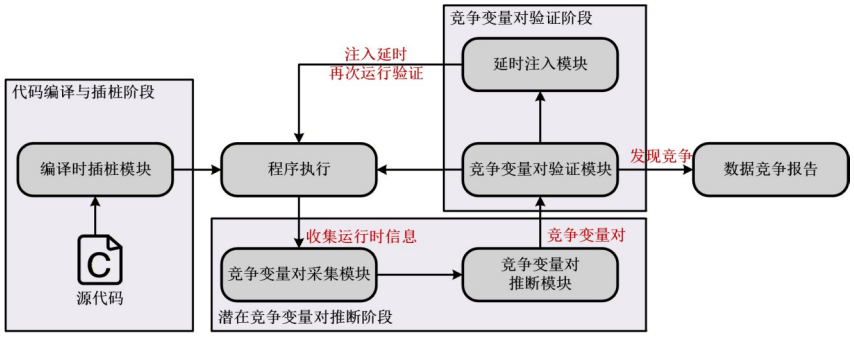


图3　RIV整体流程

3.3　RIV工作阶段

3.3.1　代码编译与插桩阶段

本文提出一种基于静态污点分析的内核竞争变量识别方法，旨在通过精准识别共享变量的传播边界，为动态数据竞争检测提供高效且低侵入性的插桩目标．该方法通过构建数据流与控制流的联合传播模型，系统性追踪共享变量在并发环境中的传播轨迹．具体而言，以内核函数的传入参数以及全局变量等作用域超越函数自身限制的变量为污染源起点，采用路径敏感的污点分析算法，追踪污点的流动路径，精确划分出被

污染的作用域可能超越函数自身的污染域.

在实现层面,本文将待扫描的函数F,待插桩的指令集合OP作为输入,将被污染的访问指令集合V_{access}作为输出. 首先,基于内核模块的调用关系与内存管理机制,显式标注函数入口参数、全局变量及共享内存区域为初始污染源,确保污染传播的起点覆盖所有可能引发数据竞争的共享资源. 其次,通过遍历函数F的控制流图,逐条解析指令的操作数(operands)与返回值(return values),对涉及污染变量的指令进行动态标记:若指令的输入操作数属于当前污染集,则其输出操作数及关联内存地址将被同步标记为污染状态. 这一过程结合路径敏感的分支条件约束,避免将污染传播至不可达代码路径,从而显著降低冗余标记的概率. 最终,算法将动态生成的污染变量指令集合与预设的插桩指令集合OP求取交集,输出具有实际观测价值的变量访问点V_{access}为后续动态检测提供高置信度的插桩目标. 具体实现如算法1所示.

算法1 静态污点分析算法

输入: 被插桩函数F,待插桩的指令集合OP

输出: 被污染的访问指令集合V_{access}

1. $V_{\text{access}}=\varnothing$
2. V_{tainted} = FuncParams $\cup$ GlobalVariables // funcParams代表函数参数、GlobalVariables代表全局变量
3. FOREACH codepath IN F DO // codepath代表代码路径
4. currentTaints=V_{tainted} // currentTaints代表当前函数的污点变量集合
5. FOREACH instruction IN codepath DO // instruction代表指令,codepath代表代码路径
6. operands=getOperands(instruction) // operands代表指令相关操作变量
7. IF operands $\cap$ currentTaints$\neq\varnothing$ THEN
8. ret_{val}=getResultValue(instruction) // ret_{val}代表指令返回值
9. currentTaints=currentTaints $\cap \{\text{ret}_{\text{val}}\}$
10. IF instruction $\in$ {currentTaints $\cap$ OP} THEN
11. V_access =V_access $\cup$ {instruction}
12. END FOREACH
13. END FOREACH
14. RETURN V_{access}

采用静态污点分析识别竞争变量的双重目标在于平衡动态检测的准确性与执行效率. 一方面,通过静态预筛选缩小插桩范围,可避免全量监控引入的性能开销,尤其针对高频访问的共享变量如自旋锁状态标志、进程调度队列指针,仅对潜在冲突点实施细粒度观测,使得运行时信息采集的资源消耗降低至可接受范围. 另一方面,为避免过度依赖保守的锁分析(lock analysis)或别名分析(alias analysis)可能导致的欠污染问题,本文在精度与效率的权衡中选择路径敏感分析作为核心策略. 通过追踪条件分支与循环结构中的污点传播路径,确保同一变量在不同执行上下文中的污染状态可被精确区分.

3.3.2 潜在竞争变量对推断阶段

在竞争变量对推断阶段,本文提出了一种基于异步日志管道的变量采集架构,以及基于时序关联规则的竞争变量对推断方法. 通过上一阶段预设的插桩点,来实时监控变量访问信息,并推断出潜在竞争变量对以供后续验证,具体而言包含以下三部分处理:

首先在基于异步日志管道的变量信息采集阶段,RIV通过预置的静态插桩点实时捕获共享变量访问事件. 为避免采集操作对原本内核线程的执行影响,本文使用异步处理的方式记录变量访问. 具体而言,当执行至变量访问桩点时,会先将变量访问信息写入缓冲区,当缓冲区满时,将缓冲区内容写入日志进行记录以供后续竞争变量的分析.

为了解决缓冲区及日志过大的问题,需要在记录变量访问时对变量信息进行筛选. 本文针对变量的地址范围进行了区分,主要是排除普通栈变量以及中断栈变量来减少缓冲区大小. 针对普通栈变量,RIV通过current结构体获取进程内核栈的起始地址,并通过THREAD_SIZE获取内核栈的总大小来计算普通栈变量的范围. 针对中断栈变量,RIV通过获取每个CPU(Central Processing Unit)的中断栈指针,使用IRQ_STACK_SIZE宏判断中断栈的总大小,计算获得中断栈的变量指针范围. 实验表明,通过排除栈变量来筛选运行时的变量访问信息使得日志总变量数减少了7%左右.

最后针对收集到的竞争变量信息,RIV采用基于时序关联规则挖掘的竞争推断机制对日志数据进行深度分析. 该机制首先对采集的变量访问序列进行地址哈希分组,通过红黑树数据结构建立以内存地址为键值的时序索引,确保每个内存单元对应的读/写操作能够以$O(\log n)$时间复杂度完成快速检索. 分析引擎扩展了Lamport定义的并发冲突判定准则[41],即对于任意两个访问事件va_x和va_y,当且仅当满足如下五项条件时,判定二者构成潜在竞争变量对:(1)线程标识符$\text{tid}(\text{va}_x)\neq\text{tid}(\text{va}_y)$,其中$\text{tid}(\text{va}_x)$为内存访问事件$\text{va}_x$所在线程的线程标识符;(2)内存地址$\text{addr}(\text{va}_x)=\text{addr}(\text{va}_y)$,其中$\text{addr}(\text{va}_x)$为内存访问事件$\text{va}_x$在此次内存操作中访问的内存地址;(3)操作类型集合$\{\text{write}(\text{va}_x),\text{write}(\text{va}_y)\}\neq\varnothing$,其中$\text{write}(\text{va}_x)$表示此次内存访问事件$\text{va}_x$为写入操作;(4)时序间隔$\Delta t<\tau$,其中$\Delta t$表示两次内存访问事件$\text{va}_x$与$\text{va}_y$的时间间隔,$\tau$为预设的临界持锁时间阈值,即500 ms;(5)访问同一结构体$T_{\text{va}_x}==T_{\text{va}_y}$,其中$T_{\text{va}_x}$表示内存访问事件$\text{va}_x$操作的结构体. 值得注意的是,临界时间阈值$\tau$的设定需综合考量目标系统的调度粒度与同步原语特性.

图4展示了RIV执行一轮采集与推断的具体流程.从程序执行开始,RIV将不同线程的变量访问信息va_{1i}、va_{1j}、va_{2l}、va_{2m}记录在日志中,推断潜在的竞争变量对,图中变量va_{1i}与变量va_{2l}满足竞争变量对条件,将二者的变量访问上下文记录为潜在的竞争变量对$\langle va_{1i}, va_{2l}\rangle$.

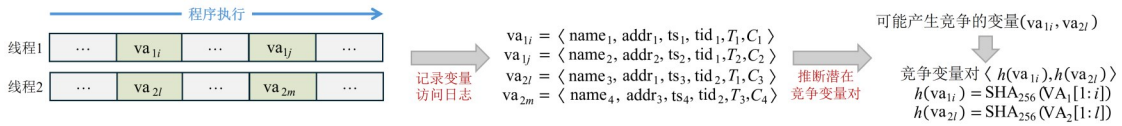


图4　竞争变量对采集与分析

具体实施过程中,系统采用滑动时间窗口算法对有序日志流进行增量式扫描,对于每组日志访问条目区间内的两条日志访问条目,分析其可能的竞争情况.详见算法2.

算法2　静态污点分析算法

输入: 变量访问日志L

输出: 潜在竞争变量对集合S

```
1. FOREACH itemInterval IN L DO // itemInterval代表访问条目区间
2.   FOREACH item1 ,item2 IN itemInterval DO // item1, item2代表访问条目
3.     IF tid(item1) ≠ tid(item2) THEN
4.       CONTINUE
5.     IF addr(item1) ≠ addr(item2) THEN
6.       CONTINUE
7.     IF NOT ( write(item1) OR write(item2) ) THEN
8.       CONTINUE
9.     IF t(item1) ≠ t(item2) THEN
10.      CONTINUE
11.    S=S ∪ ⟨item1, item2⟩
12.  END FOREACH
13. END FOREACH
```

3.3.3　竞争变量对验证阶段

在竞争变量对验证阶段,RIV针对上一阶段推断出的潜在竞争变量对,使用基于延时注入方法定向验证竞争变量对.针对一对推断阶段得出的竞争变量对$\langle va_x, va_y\rangle$,RIV会向其中一个变量访问$va_x$处注入延时进行等待,当另一变量$va_y$所在线程执行至对应的点位时,RIV会判断两个变量是否访问到了同一地址且一读一写,以真实触发数据竞争并产生报告.由于此时已经确保了两个变量处于不同线程,因此必然满足Lamport准则,故报告出的必然是真实的数据竞争.具体在执行时,RIV会再次执行测试样例,并对不同的线程设置监控,当一个线程访问至va_x的上下文时,设置观测点并注入延时,当另一线程访问至va_y的上下文时,触发数据竞争检测与报告.详见算法3.

在工程实践中,竞争变量对的延时验证包含以下两个部分.

算法3　竞争变量对验证算法

输入: 原测试样例testCase,潜在竞争变量对$\langle va_x, va_y\rangle$

```
1. startTestExecution(testCase) // startTestExecution代表启动当前测试样例
2. setupThreadMonitoring() // setupThreadMonitoring代表开启监控
3. FOREACH variableAccess DO // variableAccess代表已插桩的变量集合
4.   currentContext = getCurrentThreadContext() // currentContext代表当前线程上下文
5.   IF currentContext == h(vax) THEN
6.     setObservationPoint(vax) // setObservationPoint代表设置观测点
7.     delayInjection(vax) // delayInjection代表延时注入
8.   ELSE IF currentContext == h(vay) THEN
9.     triggerRaceDetection(vay) // triggerRaceDetection代表触发数据竞争检测
10.  END IF
11. END FOREACH
```

如图5所示,在竞争变量对验证阶段,RIV首先会从推断出的竞争变量对集合中挑出一对竞争变量对$\langle va_x, va_y\rangle$作为验证阶段的输入,之后RIV会再次执行测试样例,此处执行的测试样例是与采集阶段相同的测试样例,目的是以与采集阶段相同的方式收集每条变量访问指令的访问上下文$h(va_x)$.当访问至与va_x同样的上下文时,会通过忙等的方式进行延时,等待另一线程执行.当另一线程执行到与va_y相同的上下文时,就会触发数据竞争的检测机制.

RIV的数据竞争检测机制基于动态观测点采样方法实现.具体而言,在访问变量点va_x进行延时等待时,会同时设置一个观测点,当另一线程执行时会与va_x处设置的观测点进行访问比对,比对条件包括判断访问内存地址是否相等$addr(va_x)=addr(va_y)$,是否为至少一写$\{write(va_x), write(va_y)\} \neq \varnothing$.由于此时已经确保了两个变量在不同的线程触发,且两个变量同时访问到观测点说明满足短时间内访问,总体上满足Lamport定义的并发冲突判定准则[41].因此只需满足地址相等及至少一读两个条件时,就能说明真实触发了数据竞争.

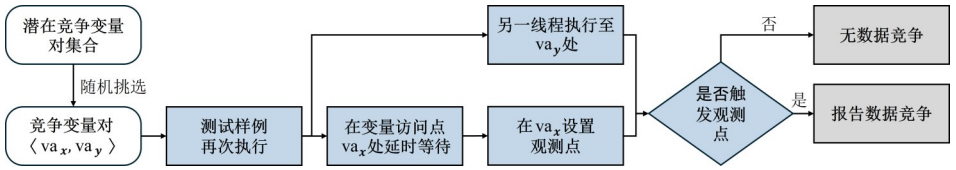


图5 竞争变量对验证流程

3.4 RIV创新性说明

本文提出的RIV方法核心创新点在于引入了“推断-验证”的新检测模式. 该范式解耦了高开销的在线监控与高精度竞争确认过程,通过一轮运行推断的方式寻找高价值的潜在竞争目标,并结合二轮运行验证来确认竞争的可复现性,从而在性能和检测精度方面体现出了优越性.

与静态分析相比,RIV方法作为动态分析具有无误报的优点. 静态分析难以处理复杂内核代码带来的函数指针、指针别名、复杂同步原语等问题,例如RacerX[9]需要处理复杂的指针别名分析和控制流,LRMiner[17]也需要挖掘大量代码样本从而归纳出可能的锁保护规则,这都导致静态分析方法可能会报告出实际永远无法执行的路径上的竞争从而产生误报. 而RIV从一个真实可达的执行路径出发进行推断,其分析的候选竞争变量对都源于实际发生过的事件,从而在根本上剔除了静态分析因路径不可达导致的大量误报问题.

与动态分析相比,RIV方法摆脱了对锁模型、发生序关系建模的依赖,并基于竞争变量对的推断结果,通过延时注入的方法定向验证了竞争在特定线程调度下能够真实发生. 锁集分析如SDILP[40]与PLA[22]都是在对锁规则的建模与追踪的基础上进行检测,但是无法检测不被锁保护的同步机制发生的竞争. 而RIV无需对锁规则进行建模,无论是锁、RCU、序列锁还是原子变量,只要满足RIV定义的推断规则,RIV就能将其纳入竞争变量对并进行定向验证. 另外RIV无须依赖严格的发生序关系进行建模,而是构建了一个更宽松的推断模型,与FastTrack[24]等发生序分析方法对比牺牲了推断阶段的准确性,将准确检测移动到二轮定向验证阶段,确保了检测的效率以及更广的检测覆盖面. 与观测点采样分析相比,RIV方法不依赖于随机采样,而是基于推断之后的定向验证进行检测. KCSAN[31]等检测工具通过对变量随机插入观测点,在冲突发生时进行检测. 然而这一过程十分依赖对观测点的采样,即需要恰好在冲突访问发生时对两侧的变量点都进行了采样才能够准确检测到数据竞争. RIV方法则基于推断的结果,使用延时注入的方式操控调度,定向验证目标能否发生竞争,保证了触发竞争的稳定性. 与模糊测试相比,RIV方法使用推断结果作为指导,定向验证每个测试用例的竞争变量对. Conzzer[35]与Krace[34]等模糊测试方法都是通过定义新的覆盖率来指导检测并发漏洞,而这一过程本质上依旧是基于输入间接达到某种内核状态以期触发数据竞争. 而RIV方法验证阶段的延时注入方法则是精确地对变量点进行测试,更加精确地利用了每个测试用例来检测数据竞争.

4 实验与评估

本节将从实验设置、数据竞争检测、现有方法对比实验等方面详细介绍RIV方法在真实内核模块中的实验评估情况,并给出详细的实验结果,以分析RIV方法的数据竞争检测有效性和效率.

4.1 实验设置

本文将Linux 6.14内核作为实验对象,并选取4款文件系统(XFS、BTRFS、JFS、F2FS)和2款网卡驱动程序(e1000e、ne2k_pci)作为测试目标. 在并发场景产生方面,本文使用文件读写基准测试工具FIO和IOZONE作为文件系统并发测试工具,并使用网络通信基准测试工具NETPERF作为网卡驱动并发测试工具. 在实验环境方面,本文选取的主机配置为AMD Ryzen 7 3700X处理器与16 GB物理内存,并利用QEMU创建内核仿真测试环境(配置为4 GB内存,2核CPU). 表1展示了测试目标的详细信息,其中源代码行数采用CLOC工具统计获得.

表1 被测程序基本信息

测试目标	测试目标描述	测试目标代码行数/K行
btrfs	btrfs文件系统	111.3
xfs	xfs文件系统	146.1
jfs	jfs文件系统	16.9
f2fs	f2fs文件系统	37.9
e1000e	e1000驱动程序	19.4
ne2k_pci	net2k_pci驱动程序	9.1

在对比实验方面,本文选取两款典型的内核数据竞争检测工具KCSAN[31]、DILP[20]以及DILP的扩展工具SDILP[40]作为对比对象. 其中,KCSAN是Linux内核集成的数据竞争检测器,版本为对应测试内核版本即Linux 6.14,使用观测点采样分析和随机延时注入方法;DILP与SDILP是学术界典型的内核数据竞争检测方法,使用动态锁集分析和变量访问监控方法,其中

SDILP为DILP在静态插桩以及静态分析部分扩展后的工具. 这三种方法均可对本文的6款内核模块进行实际测试,以实现与RIV方法的效果对比.

4.2　插桩优化及数据竞争检测结果

本文使用RIV方法在相同的实验环境下分别对6个内核模块进行了24 h的持续性测试,结果如表2所示,其中第2列表示RIV方法在不同内核模块的编译时插桩的优化情况,第3、4、5列表示数据竞争检测结果.

在代码编译与插桩阶段,RIV会使用静态污点分析的方法优化所需插桩变量的数量. 从表2可以看出,RIV分别在btrfs、xfs、jfs、f2fs文件系统中减少了57.17%、50.15%、59.70%、58.98%的插桩点数,平均将文件系统的监控桩点数量降低了56.50%.在网卡驱动e1000e与ne2k_pci上减少了71.53%与64.86%的插桩点数. 需要指出的是,RIV考虑到动态分析的漏报可能,采取了较为宽松的变量筛选策略. 另外,静态污点分析并非本文的核心创新点,因此在插桩上仍有可优化的空间. 例如与SDILP在e1000e网卡驱动上将变量点数优化为739(↓94.87%)相比,本文的污点分析方法并未表现出突出优化效果,不过在后续整体的数据竞争检测与性能测试上,RIV依然表现出了比SDILP更优秀的结果.

在数据竞争检测方面,RIV方法共推断出1 287个潜在竞争变量对,经过定向验证共有31个数据竞争能够真实触发,且没有误报情况产生. 经过手动分析内核模块代码,发现其中有12个数据竞争可导致数据不一致等有害行为,19个数据竞争是良性的,不会造成有害行为或仅会在极端情况下造成较小影响. 本文将发现的12个有害数据竞争报告给了相应的内核开发者,并得到了确认.

表2　插桩优化及数据竞争检测结果　　单位:个

测试模块	优化前插桩变量点数/优化后插桩变量点数	潜在竞争变量对	真实数据竞争	有害数据竞争
btrfs	37 927/16 245(↓57.17%)	318	14	6
xfs	63 955/31 879(↓50.15%)	209	4	2
jfs	6 125/2 468(↓59.70%)	112	2	1
f2fs	18 581/7 622(↓58.98%)	154	6	1
e1000e	14 414/4 104(↓71.53%)	195	3	1
ne2k_pci	1 309/460(↓64.86%)	299	2	1
总计	—	1 287	31	12

图6(*a*)展示了一个由RIV方法在btrfs文件系统中发现的有害数据竞争示例. 线程1在执行函数btrfs_update_block_group时,在第3 678行会对变量cache->used进行赋值操作;同时,线程2在执行函数btrfs_discard_queue_work时,在第366行会对变量block_group->used进行读取操作. 在运行过程中,两个变量cache->used和block_group->used会访问相同的内存地址,赋值和读取操作会并发执行,因此发生了数据竞争. 由于文件系统在第366行对变量block_group->used进行读取操作后进行条件判断,影响文件系统元数据添加的功能函数add_to_discard_unused_list的执行,因此可能导致文件系统元数据被破坏.

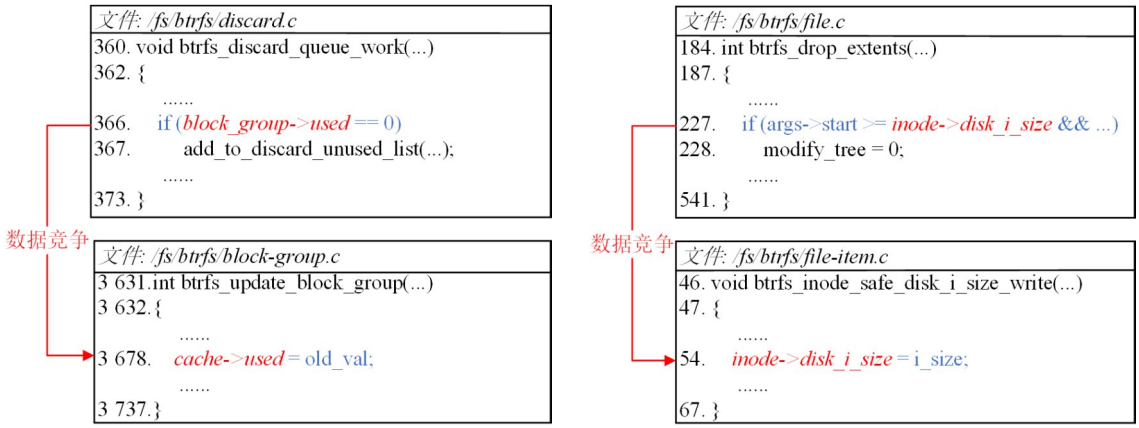
图6(*b*)展示了一个由RIV方法在btrfs文件系统中发现的良性数据竞争示例. 当线程1在执行函数btrfs_inode_safe_disk_i_size_write时,会在第54行对变量inode->disk_i_size进行赋值操作. 同时线程2在执行函数btrfs_drop_extents时,会在第227行对变量inode->disk_i_size进行读取操作. 在运行过程中,两个变量inode->disk_i_size会访问相同的内存地址,赋值与读取操作会并发执行,因此发生了数据竞争. 而文件系统在第228行对变量modify_tree进行了修改,该变量会影响后续的b+树合并,导致b+树优化失败,从而造成效率上的影响.

本文提出的RIV方法能够针对Linux全内核代码进行检测. 具体而言,在代码编译与插桩阶段将内核的编译工具指定为RIV的编译插桩工具,再正常启动内核即可进行测试. 除了上述6个内核模块,RIV还在针对全Linux内核进行插桩的情况下使用iozone与Syzkaller生成的固定测试用例进行了24 h测试,分别在内存管理子系统、虚拟文件系统、蓝牙模块检测出了1个、2个与6个数据竞争.

本文提出的RIV方法在数据竞争检测方面依然存在一些局限性. 其一,RIV方法检测出的大部分数据竞争为良性数据竞争,缺少对有害数据竞争的自动识别检测能力;其二,RIV方法在检测覆盖度方面依然依赖于测试负载和用户输入. 因此可以考虑将RIV与Razzer和Syzkaller等内核模糊测试工具相结合,提升内核测试覆盖度,以发现更多复杂数据竞争.

4.3　现有方法对比实验

本文在相同的实验环境下,使用RIV方法与两款现有内核数据竞争检测工具KCSAN、DILP和SDILP进行实验对比,每款内核模块的测试时间为24 h. 同时对



(*a*) 有害数据竞争示例 (*b*) 良性数据竞争示例

图6 数据竞争示例

于发现的数据竞争,本文对相关内核模块代码进行手动分析,以识别其中的有害数据竞争. 本文选取数据竞争检测和检测性能的结果作为对比实验的关键指标进行详细分析.

4.3.1 数据竞争检测

表3展示了KCSAN、DILP、SDILP和RIV这4种方法的数据竞争检测结果. 具体来讲,KCSAN共发现了15个真实数据竞争,其中8个为有害数据竞争;DILP共发现了10个真实数据竞争,其中7个为有害数据竞争;SDILP共发现了13个真实数据竞争,其中8个为有害数据竞争(包括DILP发现的全部10个数据竞争以及7个有害数据竞争). 相比之下,RIV方法不但发现了KCSAN、DILP和SDILP发现的所有真实数据竞争和有害数据竞争,并多发现了16个真实数据竞争(KCSAN与SDILP发现9个相同的数据竞争,各自发现6个与4个不同的数据竞争),包括4个有害数据竞争. 实验结果表明,RIV在数据竞争检测方面的效果优于KCSAN、DILP与SDILP这3种现有方法.

表3 插桩优化及数据竞争测试结果 单位:个

测试模块	DILP		SDILP		KCSAN		RIV	
	真实数据竞争	有害数据竞争	真实数据竞争	有害数据竞争	真实数据竞争	有害数据竞争	真实数据竞争	有害数据竞争
btrfs	2	2	3	3	6	4	14	6
xfs	1	1	1	1	2	2	4	2
jfs	1	0	1	0	1	0	2	1
f2fs	2	1	2	1	2	1	6	1
e1000e	2	2	3	2	3	1	3	1
ne2k_pci	2	1	3	1	1	0	2	1
总计	10	7	13	8	15	8	31	12

RIV表现出更优秀的数据竞争检测能力的核心原因在于使用了推断-验证的检测机制. 具体而言,RIV在推断阶段找出了常规线程调度下难以产生竞争的变量对,并通过延时注入的方式定向验证,从而找到了其他工具难以在常规测试用例下发现的数据竞争. 以SDILP为例,SDILP的检测算法要求对并发访问同一内存的变量点进行持锁的交集检测,然而在24 h的实验测试中,依然有21个数据竞争(SDILP在动态测试中发现了10个数据竞争,在之后的静态扩展分析中多发现了3个数据竞争,因此动态测试过程只发现了10个数据竞争,相较RIV的31个数据竞争少发现了21个数据竞争)没有在常规测试调度下被发现. 经过对未被SDILP检测出的竞争变量的分析,发现大部分变量对相应地址的访问时间都较短,因此在真正发生竞争之前就已经改变了变量地址,导致无法被SDILP检测到;而RIV通过延时注入的方式,让变量长时间持有对应的地址,因此成功触发了竞争.

实际上,KCSAN、DILP及SDILP需要程序在指令或者函数级别出现真实的并发情况时才能进行数据竞争检测,该检测条件较为苛刻且受到内核线程调度随机性的严重影响. 相比之下,RIV方法通过运行时信息进行潜在竞争变量对的推断,可放宽数据竞争的检测条件,并减少检测过程受内核线程调度随机性的影响,从而提升复杂数据竞争的发现能力和增强数据竞争的检

测范围. 同时,RIV方法的验证阶段通过潜在竞争变量对的定向复现,从而保障该方法与KCSAN、DILP及SDILP一样实现零误报的效果.

除了上述6个内核模块,本文还将RIV在蓝牙模块、以太网卡、声卡模块等其他内核模块上实现了测试,验证了RIV方法的可扩展性.

4.3.2　检测性能分析

为了评估KCSAN、DILP、SDILP和RIV的数据竞争检测性能,本文利用IOZONE和NETPERF分别对4款文件系统和2款网卡驱动进行性能评测. 文件系统测试方案主要针对随机写入,通过调整块大小(4~256 KB)、队列深度(1~128)与并发线程数(2~8)构建测试;网卡驱动测试方案主要针对TCP流的吞吐能力,通过设定60 s测试时长以及1 500 B数据包测试吞吐能力.

表4展示了原始内核运行和4种方法数据竞争检测的性能结果. 由于RIV包括推断阶段和验证阶段,所以表4中分别对其性能结果进行详细展示,其中括号中的百分数是相较原始内核运行的性能损失百分数. 与KCSAN、DILP和SDILP相比,RIV推断阶段的平均性能分别提升了1.5倍、6.7倍和1.8倍,RIV验证阶段的平均性能分别提升了418倍、1 878倍与376倍,RIV验证阶段相较原始内核仅有23.15%左右的性能下降. 需要注意的是,RIV验证阶段只对潜在竞争变量对相关的变量访问进行运行时拦截分析,而RIV推断阶段需要对所有潜在共享变量访问进行运行时拦截分析,因此后者所需要收集处理的数据更多,其运行时开销也更大.

表4　RIV与对比工具的文件读写/网络吞吐效率对比　　单位:B/s

被测对象	原始内核	KCSAN	DILP	SDILP	RIV(推断阶段)	RIV(验证阶段)
xfs	275 M	123 K(↓99.96%)	27 K(↓99.99%)	140 K(↓99.95%)	188 K(↓99.93%)	223 M(↓18.91%)
btrfs	87 M	316 K(↓99.65%)	77 K(↓99.91%)	380 K(↓99.57%)	480 K(↓99.46%)	62 M(↓28.74%)
jfs	229 M	799 K(↓99.66%)	171 K(↓99.93%)	749 K(↓99.68%)	1 M(↓99.55%)	176 M(↓23.14%)
f2fs	141 M	487 K(↓99.66%)	118 K(↓99.92%)	515 K(↓99.64%)	789 K(↓99.45%)	103 M(↓26.95%)
e1000e	855 K	172 K(↓79.88%)	43 K(↓94.97%)	260 K(↓69.59%)	286 K(↓66.55%)	666 K(↓22.11%)
ne2k_pci	892 K	181 K(↓79.71%)	35 K(↓96.08%)	63 K(↓92.94%)	263 K(↓70.52%)	722 K(↓19.06%)

实际上,KCSAN、DILP和SDILP需要对所有变量访问进行运行时拦截分析,并利用串行方式进行信息记录和处理. 相比之下,RIV推断阶段通过静态污点分析仅对潜在共享变量访问进行运行时拦截分析,并利用并行机制和信号控制来提升运行时信息采集性能,因此RIV推断阶段可实现更快的运行效率.

4.3.3　性能开销评估

为了评估KCSAN、DILP、SDILP和RIV在数据竞争检测过程中的性能开销,本文将四者在CPU和内存资源消耗方面进行了详细对比,实验结果如表5所示. 该评估覆盖了文件系统(xfs、btrfs、jfs、f2fs)和网络设备驱动(e1000e、ne2k_pci)两类典型的内核并发场景. 实验数据清晰地表明,RIV在实现高精度竞争检测的同时,并未引入显著的额外性能负担,其资源消耗与现有主流工具处于同一水平,甚至在特定阶段表现更优.

表5　RIV与对比工具的CPU使用率和内存使用率对比

被测对象	KCSAN		DILP		SDILP		RIV(推断阶段)		RIV(验证阶段)	
	CPU/%	内存/MB	CPU/%	内存/MB	CPU/%	内存/MB	CPU/%	内存/MB	CPU/%	内存/MB
xfs	194.48	16.13	208.33	24.11	199.58	20.85	196.41	18.66	192.36	12.85
btrfs	195.62	17.22	199.64	25.62	198.15	22.63	195.44	19.32	194.82	15.51
jfs	192.74	14.85	196.92	23.18	195.96	18.75	193.65	13.98	189.44	13.91
f2fs	192.85	15.62	197.45	24.45	198.12	19.83	192.88	15.66	190.14	14.35
e1000e	55.43	4.32	72.62	10.30	68.74	8.96	55.87	8.15	54.36	2.36
ne2k_pci	54.34	3.26	71.55	9.85	67.15	8.34	52.83	6.13	52.36	2.44

在CPU利用率方面,RIV的推断阶段和验证阶段均表现出与轻量级基准工具KCSAN高度相当的水平. 例如,在对xfs文件系统进行测试时,RIV推断阶段的CPU使用率(196.41%)与KCSAN(194.48%)几乎持平,而验证阶段(192.36%)甚至略有降低. 这一趋势在所有被测对象上保持了一致性. 更重要的是,与DILP(208.33%)和SDILP(199.58%)等需要进行更复杂运行时分析的工具相比,RIV的CPU开销具有明显优势,这得益于其将复杂的分析任务移至离线阶段,从而减轻了在线监控的压力.

在内存消耗方面,RIV的优势更为显著,体现了其两阶段设计的精巧之处. 其推断阶段由于需要缓存运行时访问日志,内存占用(如在xfs上的18.66 MB)略高于KCSAN(16.13 MB),但依然显著低于需要维护复杂状态的DILP(24.11 MB)和SDILP(20.85 MB),这证明了

RIV的日志记录机制是高效且内存友好的. 尤为关键的是,在验证阶段,RIV的内存开销是所有对比工具中最低的. 例如,在xfs测试中,其内存占用仅为12.85 MB. 这是因为验证阶段的目标极其明确,仅需为特定的、被推断出的候选竞争对设置观测点和维持最小化状态,而无需进行全局性的监控,从而极大地节约了存储资源.

5 相关工作

5.1 基于静态分析的数据竞争检测技术

基于静态分析的数据竞争检测技术是指以静态方式在编译阶段识别潜在的数据竞争点. 静态分析不需要实际运行程序,而是通过控制流分析、数据流分析等手段推断可能在多线程环境下被无保护访问的变量点. 2003年,Engler等人[9]设计了RacerX,使用流敏感的过程间分析来检测数据竞争和死锁漏洞. 其从源文件中提取控制流图和变量信息,利用过程间、流敏感和上下文敏感的分析来计算代码路径中的锁集合并检测数据竞争,最后对结果进行后处理和排名以生成数据竞争报告. 2007年,Voung等人[10]设计了RELAY,采用推测分析结合上下文敏感和路径敏感分析的方式实现了可扩展的竞争检测分析. 2011年,Pratikakis等人[12]设计了LOCKSMITH,使用简单的无工作列表策略产生过程间数据流分析,精确构建C结构体和void指针的精度,实现了针对Linux内核设备驱动的数据竞争检测. 2016年Vojdani等人[13]结合了不同的指针分析与类型分析,在Goblint静态分析框架实现了针对Linux设备驱动的锁集分析,实现数据竞争的检测. 2021年,Andrianov等人[14]提出了一种将线程转换为包含共享数据与同步原语的投影,通过分析投影来实现数据竞争检测的方案. 2023年,Chen等人[15]提出了SGXracer工具,使用锁集分析的方法在可信执行环境中检测数据竞争. 同年Cai等人[16]提出了Lockpick,通过类型敏感的类型状态分析与并发感知两个阶段结合,分析可能存在的错误加锁状态,以检测数据竞争. 2025年,Li等人[17]提出了LR-Miner,通过挖掘内核代码的锁规则构建锁与变量之间的结构字段关系,以检测数据竞争.

尽管静态分析能够在数据竞争检测过程中能够覆盖大范围代码,但是由于缺少运行时的内存访问等信息,从而表现出了高误报率的特征. 本文提出的RIV方法主要采用动态分析的方式,以“零误报”的方式检测数据竞争. 但必须指出,一方面本文提出的RIV动态检测方法存在漏报的可能性,需要与模糊测试等输入用例生成工具结合测试更大范围的代码;另一方面可以通过静态分析的方式优化过滤一些不常见的变量点位,以进一步优化RIV检测方法.

5.2 基于动态分析的数据竞争检测技术

基于动态分析的数据竞争检测技术是在程序运行时进行检测,监控线程间的内存访问操作,利用程序运行记录的读写时间戳或追踪锁机制来判断是否发生了数据竞争. 动态分析依赖于实际执行路径,因此能提供更精确的检测结果,但动态检测数据竞争的检测范围受限于线程间的交错程度. 动态检测数据竞争有三种主流手段,分别是动态锁集分析、发生序分析和观测点采样分析. 动态锁集分析是通过追踪程序运行过程中的锁函数,来识别变量的受保护情况,以找到缺少锁保护的潜在竞争变量[19~23]. 其中在2019年,Chen等人[20]提出了一种名为DILP的动态锁集分析方法,首次将动态锁集分析应用到内核驱动检测,在Linux内核驱动中检测出了大量数据竞争. 发生序分析是通过记录程序运行发生事件的时间戳,推断变量可能的并发情况,以找到潜在的竞争变量[24~28]. 然而由于记录时序关系的开销过大,内核态的数据竞争检测很少采用追踪发生序关系的方法. 观测点采样是通过在变量执行时插入观测点并暂停线程后续操作,通过观察另一线程的内存访问情况来找到竞争的变量[29~31]. 其中在2020年,Linux内核开发者基于DataCollider的思想实现了KCSAN[31],在Linux内核中检测出了大量数据竞争. 2022年Bai等人[40]在DILP的基础上扩展实现了SDILP,优化了插桩方案并添加了静态分析扩展,发现了相同锁集保护下的更多数据竞争.

动态分析方法需要覆盖不同的线程交错情况以发现更多问题,通常动态分析会通过随机注入延时的方式来覆盖更多的线程交错. 而这种随机注入方法经常性能开销大的问题,同时也难以发现不常见的数据竞争. 而本文提出的RIV方法通过推断-验证的数据竞争检测模式,先在运行时收集潜在的竞争变量对信息,再通过延时注入的方式对其进行定向验证,发现了常规条件下难以检测的数据竞争.

5.3 基于模糊测试的数据竞争检测技术

基于模糊测试的数据竞争检测技术结合了动态分析和自动化测试生成,通过模糊测试生成多样化的输入来覆盖尽可能多的程序执行路径,进而触发潜在的数据竞争. 在2019年,Jeong等人[32]提出了Razzer,Razzer通过静态分析的方式来识别潜在的数据竞争点,并通过两轮模糊测试,定向探索可能存在数据竞争的点. 2020年,Xu等人[34]提出了工具Krace,通过别名覆盖度指导系统调用变异,结合发生序分析和锁集分析的方法检测竞争. Chen等人[33]在同年提出了MUZZ工具,以灰盒的方式检测多线程程序中的数据竞争. 2022年,Jiang等人[35]提出了CONZZER工具,提出了上下文敏感的并发调用对的概念,并以此为覆盖率指导定向

探索线程交错,发现了大量真实数据竞争.2023年,Jeong等人[36]提出SegFuzz工具,提出了一种描述线程交错空间的描述方式,并使用模糊测试的方式在内核中发现了大量并发缺陷.2024年,Wolff等人[37]使用与灰盒测试类似的方式,指导测试暴露出新的并发情况,以挖掘并发缺陷.

模糊测试的方法主要通过描述并发情况下的代码覆盖度,来生成多样化的输入用例以期覆盖更多的并发执行情况,但是对于生成的测试用例,单纯的模糊测试方法无法充分发挥其在数据竞争检测方面的能力.而本文提出的RIV方法可以通过与模糊测试的输入用例生成相结合,将模糊测试工具作为RIV的输入引擎,对每个输入测试用例执行推断-验证模式检测,以充分挖掘每个用例里面潜在的数据竞争.

5.4　基于污点分析的数据竞争检测技术

基于污点分析的技术主要可以分为静态污点分析和动态污点分析.静态污点分析受限于高误报率制约了其广泛应用,动态污点分析通过标记外部输入或敏感数据为污点,追踪其在程序中的流动以检测是否触发安全漏洞.在数据竞争检测上,污点分析可以通过标记共享内存地址作为污点源,动态追踪线程间的读写操作,以检测未同步的访问事件.在2008年,Nightingale等人[42]提出了Speck方法,通过推测执行将检测任务从主线程剥离单独检测,并通过进程级重放系统保证状态一致性以并行执行污点分析检测.2019年,Banerjee等人[43]提出了一种乐观混合分析框架Iodine,其通过提取高概率运行时不变式构建污点传播模型,大幅提升精度的同时减少了冗余监控点.同年Davanian等人[44]提出了一种弹性的全系统动态污点分析模型DECAF++,基于污点数据的实时传播分析以及传播路径熔断等方法避免冗余监控,使用纯软件的方法优化污点分析.2023年,Ouyang等人[45]提出了MirrorTaint,通过在微服务字节码中注入轻量级指令,设计针对JVM指令集的细粒度传播策略实现了微服务场景下的污点分析.2024年,Zhang等人[46]提出了HardTaint,结合静态分析,选择性硬件跟踪和并行图处理技术优化污点分析的效率.

污点分析技术通过追踪不可信源数据的传播路径以检测漏洞,动态污点分析通常采用编译时插桩运行时收集信息,并对信息进行分析检测的方式捕捉数据竞争.然而,现有的动态污点分析尽管能够在运行时追踪潜在的竞争变量访问操作,却缺乏对数据竞争的定向验证及复现支持.

6　结论

针对当前数据竞争检测工具的检测效率低与有效性不足的问题,本文提出了一种基于推断-验证模式的内核数据竞争检测方法RIV,针对操作系统内核中复杂的并发场景下的数据竞争检测问题,实现了精度与效率的协同优化,通过引入“潜在竞争变量对”的概念,进行数据竞争的精确检测与复现.具体来讲,RIV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对进行定向验证,以实现数据竞争的精确检测与复现.同时,RIV利用静态污点分析识别潜在共享变量,以减少被插桩代码量和降低运行性能开销;并通过采集变量访问的内存地址和发生时间,确保数据竞争检测准确性.在实验评估中,本文利用RIV在6款常用的Linux内核模块(包括4款文件系统和2款网卡驱动)进行了实际测试,发现了31个真实的数据竞争且没有误报,其中12个被Linux内核开发者确认.相比现有内核数据竞争检测方法KCSAN、DILP和SDILP,RIV分别提升了1.5倍、6.7倍与1.8倍性能,并发现了更多真实的数据竞争.在未来的工作中,本文将首先关注如何结合Syzkaller等内核模糊测试方法,覆盖更广的内核代码以收集更多的竞争变量对以进行验证;其次将关注如何将竞争变量对推广到原子性违反、顺序性违反等其他类型的并发漏洞检测上,以提升方法的通用性,发现更多严重内核漏洞.

参考文献

[1] CHEN H G, MAO Y D, WANG X, et al. Linux kernel vulnerabilities: State-of-the-art defenses and open problems[C]//Proceedings of the Second Asia-Pacific Workshop on Systems. New York: ACM, 2011: 1-5.

[2] LU S, PARK S, SEO E, et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics[J]. ACM SIGOPS Operating Systems Review, 2008, 42(2): 329-339.

[3] RYZHYK L, CHUBB P, KUZ I, et al. Dingo: Taming device drivers[C]//Proceedings of the 4th ACM European Conference on Computer Systems. New York: ACM, 2009: 275-288.

[4] TAN L, LIU C, LI Z M, et al. Bug characteristics in open source software[J]. Empirical Software Engineering, 2014, 19(6): 1665-1705.

[5] LINUS T. CVE-2016-5195[EB/OL]. (2016-10-18) [2025-05-26]. https://nvd.nist.gov/vuln/detail/CVE-2016-5195.

[6] KELLERMANN M. CVE-2022-0847[EB/OL]. (2022-02-21)[2025-05-26]. https://nvd.nist.gov/vuln/detail/CVE-2022-0847.

[7] LINUS T. CVE-2022-0185[EB/OL]. (2022-01-18) [2025-05-26]. https://nvd.nist.gov/vuln/detail/CVE-2022-0185.

[8] GREG K H. CVE-2025-21998[EB/OL]. (2025-03-28)[2025-

05-26]. https://nvd.nist.gov/vuln/detail/CVE-2025-21998.

[9] ENGLER D, ASHCRAFT K. RacerX: Effective, static detection of race conditions and deadlocks[J]. ACM SIGOPS Operating Systems Review, 2003, 37(5): 237-252.

[10] VOUNG J W, JHALA R, LERNER S. RELAY: Static race detection on millions of lines of code[C]//Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2007: 205-214.

[11] KAHLON V, SINHA N, KRUUS E, et al. Static data race detection for concurrent programs with asynchronous calls[C]//Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2009: 13-22.

[12] PRATIKAKIS P, FOSTER J S, HICKS M. LOCKSMITH: Practical static race detection for C[J]. ACM Transactions on Programming Languages and Systems, 2011, 33(1): 1-55.

[13] VOJDANI V, APINIS K, RÕTOV V, et al. Static race detection for device drivers: The Goblint approach[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2016: 391-402.

[14] ANDRIANOV P, MUTILIN V, KHOROSHILOV A. Cpalockator: Thread-modular analysis with projections (competition contribution)[M]//Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer International Publishing, 2021: 423-427.

[15] CHEN S, LIN Z, ZHANG Y. Controlled data races in enclaves: Attacks and detection[C]//32nd USENIX Security Symposium. California: USENIX Association, 2023: 4069-4086.

[16] CAI Y D, YAO P S, YE C F, et al. Place your locks well: Understanding and detecting lock misuse bugs[C]//USENIX Security Symposium. California: USENIX Association, 2023: 1-18.

[17] LI T, BAI J J, HAN G D, et al. LR-Miner: Static race detection in OS kernels by mining locking rules[C]//33rd USENIX Security Symposium. California: USENIX Association, 2024: 6149-6166.

[18] SALES E, INVERSO O, TUOSTO E. Accurate static data race detection for C[M]//Formal Methods. Cham: Springer Nature Switzerland, 2024: 443-462.

[19] CAI Y, ZHANG J, CAO L W, et al. A deployable sampling strategy for data race detection[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2016: 810-821.

[20] CHEN Q L, BAI J J, JIANG Z M, et al. Detecting data races caused by inconsistent lock protection in device drivers[C]//2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. Piscataway: IEEE, 2019: 366-376.

[21] ARAHORI Y. RangeLocker: Adaptive range-sensitive lockset analysis for precise dynamic race detection[C]//2019 IEEE 19th International Symposium on High Assurance Systems Engineering. Piscataway: IEEE, 2019: 184-191.

[22] RYAN G, SHAH A, SHE D D, et al. Precise detection of kernel data races with probabilistic lockset analysis[C]//2023 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2023: 2086-2103.

[23] OLIVEIRA J, GONÇALVES J, MATOS M. HawkSet: Automatic, application-agnostic, and efficient concurrent PM bug detection[C]//Proceedings of the Twentieth European Conference on Computer Systems. New York: ACM, 2025: 1092-1108.

[24] FLANAGAN C, FREUND S N. FastTrack: Efficient and precise dynamic race detection[J]. ACM SIGPLAN Notices, 2009, 44(6): 121-133.

[25] JIANG Y Y, YANG Y, XIAO T, et al. DRDDR: A lightweight method to detect data races in Linux kernel[J]. The Journal of Supercomputing, 2016, 72(4): 1645-1659.

[26] LI G P, LU S, MUSUVATHI M, et al. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM, 2019: 162-180.

[27] YU K P, WANG C X, CAI Y, et al. Detecting concurrency vulnerabilities based on partial orders of memory and thread events[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM, 2021: 280-291.

[28] GORJIARA H, XU G H, DEMSKY B. Yashme: Detecting persistency races[C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2022: 830-845.

[29] BOND M D, COONS K E, MCKINLEY K S. PACER: Proportional detection of data races[J]. ACM SIGPLAN Notices, 2010, 45(6): 255-268.

[30] ERICKSON J, MUSUVATHI M, BURCKHARDT S, et al. Effective data-race detection for the kernel[C]//Proceedings of the 9th USENIX conference on Operating

systems design and implementation. California: USENIX Association, 2010: 151-162.
[31] ELVER M, VYUKOV D. KCSAN[EB/OL]. (2022-09-28)[2025-05-26]. https://github.com/google/kernel-sanitizers/blob/master/KCSAN.md.
[32] JEONG D R, KIM K, SHIVAKUMAR B, et al. Razzer: Finding kernel race bugs through fuzzing[C]//2019 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2019: 754-768.
[33] CHEN H, GUO S, XUE Y, et al. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs[C]//29th USENIX Security Symposium. California: USENIX Association, 2020: 2325-2342.
[34] XU M, KASHYAP S, ZHAO H Q, et al. Krace: Data race fuzzing for kernel file systems[C]//2020 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2020: 1643-1660.
[35] JIANG Z M, BAI J J, LU K J, et al. Context-sensitive and directional concurrency fuzzing for data-race detection[C]//Proceedings 2022 Network and Distributed System Security Symposium. Internet Society, 2022: 1-18.
[36] JEONG D R, LEE B, SHIN I, et al. SegFuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing[C]//2023 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2023: 2104-2121.
[37] WOLFF D, SHI Z, DUCK G J, et al. Greybox fuzzing for concurrency testing[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. New York: ACM, 2024: 482-498.
[38] STERN A. Explanation of the linux-kernel memory consistency model[EB/OL]. (2017-10-01)[2025-05-26]. https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/explanation.txt.
[39] BAI J J. CVE-2023-52855[EB/OL]. (2023-11-20)[2025-05-26]. https://nvd.nist.gov/vuln/detail/CVE-2023-52855.
[40] BAI J J, CHEN Q L, JIANG Z M, et al. Hybrid static-dynamic analysis of data races caused by inconsistent locking discipline in device drivers[J]. IEEE Transactions on Software Engineering, 2022, 48(12): 5120-5135.
[41] LAMPORT L. Time, clocks, and the ordering of events in a distributed system[J]. Communications of the ACM, 1978, 21(7): 558-565.
[42] NIGHTINGALE E B, PEEK D, CHEN P M, et al. Parallelizing security checks on commodity hardware[C]//Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2008: 308-318.
[43] BANERJEE S, DEVECSERY D, CHEN P M, et al. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis[C]//2019 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2019: 490-504.
[44] DAVANIAN A, QI Z, QU Y, et al. DECAF++: Elastic whole-system dynamic taint analysis[C]//22nd International Symposium on Research in Attacks, Intrusions and Defenses. California: USENIX Association, 2019: 31-45.
[45] OUYANG Y C, SHAO K L, CHEN K Q, et al. MirrorTaint: Practical non-intrusive dynamic taint tracking for JVM-based microservice systems[C]//Proceedings of the 45th International Conference on Software Engineering. New York: ACM, 2023: 2514-2526.
[46] ZHANG Y Y, LIU T Y, WANG Y Y, et al. HardTaint: Production-run dynamic taint analysis via selective hardware tracing[J]. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA2): 1615-1640.

作者简介

郑浩然　男,2001年9月出生于河南省郑州市. 现为北京航空航天大学网络空间安全学院硕士研究生. 主要研究方向为操作系统内核安全.
E-mail: zhenghaoran@buaa.edu.cn

张　涔　男,2003年7月出生于福建省三明市. 现为北京航空航天大学博士研究生. 主要研究方向为操作系统内核安全.
E-mail: zzzccc427@buaa.edu.cn

白家驹　男,1990年5月出生于河北省石家庄市. 2018年毕业于清华大学计算机科学与技术系. 现为北京航空航天大学副教授. 主要研究方向为系统软件安全. 中国电子学会会员编号:E190188406M.
E-mail: nymph@uestc.edu.cn

关振宇　男,1984年9月出生于内蒙古自治区赤峰市. 2013年博士毕业于英国帝国理工大学. 现为北京航空航天大学教授、博士生导师,国家杰出青年科学基金获得者、国家级青年人才. 主要研究方向为网络安全、区块链、人工智能. 中国电子学会会员编号:E190051987M.
E-mail: guanzhenyu@buaa.edu.cn