软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(11):5356−5385 [doi: 10.13328/j.cnki.jos.007377] [CSTR: 32375.14.jos.007377]

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

Linux 内核定时器并发错误检测*

周多明[1], 马 麟[2], 周亚金[2]

[1](中国人民解放军 93558 部队, 河北 石家庄 050051)

[2](浙江大学 网络空间安全学院, 浙江 杭州 310058)

通信作者: 周亚金, E-mail: yajin_zhou@zju.edu.cn

摘 要: 定时器是操作系统延迟任务调度与执行的驱动器, 具有运行在原子上下文和异步执行的特性, 可以在任何时刻与不同的线程并发执行, 如果开发人员不能考虑到所有多线程交错的场景, 则可能引入多种类型的并发错误, 对操作系统安全产生严重威胁. 定时器并发错误不仅涉及多线程交错, 还涉及定时器处理程序的延迟执行与重复调度, 比普通的并发错误更难发现, 目前还没有工具可以有效地检测此类错误. 总结 3 种定时器并发错误类型, 即定时器睡眠错误、定时器死锁错误和僵尸定时器错误. 为有效地对错误进行检测, 首先通过指针分析, 提取内核中所有与定时器有关的功能模块, 避免对无关代码进行分析提高检测效率; 然后构建上下文敏感、路径敏感和流敏感的过程间控制流图, 为后续分析奠定基础; 最后综合应用函数调用图遍历、锁集分析、指向分析、控制流分析等静态分析技术, 设计针对 3 种定时器并发错误的检测算法. 为评估算法的有效性, 在 Linux 5.15 内核中共发现了 328 个真实定时器并发错误, 向 Linux 内核社区提交了 56 个补丁, 截至目前, 49 个补丁已经合并到 Linux 内核主线, 295 个错误被确认和修复, 申请了 14 个 CVE 编号, 说明了所提方法的有效性. 最后通过对比实验对算法的性能、漏报与误报情况进行了系统分析, 并总结 3 种定时器并发错误的修复方法.

关键词: 静态分析; Linux 内核定时器; 并发错误; 操作系统安全

中图法分类号: TP306

中文引用格式: 周多明, 马麟, 周亚金. Linux内核定时器并发错误检测. 软件学报, 2025, 36(11): 5356–5385. http://www.jos.org.cn/1000-9825/7377.htm

英文引用格式: Zhou DM, Ma L, Zhou YJ. Detection of Timer Concurrency Bug in Linux Kernel. Ruan Jian Xue Bao/Journal of Software, 2025, 36(11): 5356–5385 (in Chinese). http://www.jos.org.cn/1000-9825/7377.htm

Detection of Timer Concurrency Bug in Linux Kernel

ZHOU Duo-Ming[1], MA Lin[2], ZHOU Ya-Jin[2]

[1](The 93558 Unit of Chinese People's Liberation Army, Shijiazhuang 050051, China)

[2](School of Cyber Science and Technology, Zhejiang University, Hangzhou 310058, China)

Abstract: A timer is used to schedule and execute delayed tasks in an operating system. It operates asynchronously in an atomic context and can execute concurrently with different threads at any time. If developers fail to account for all possible scenarios of multithread interleaving, various types of concurrency bugs may be introduced, posing a serious threat to the security of the operating system. Timer concurrency bugs are more difficult to detect than typical concurrency bugs because they involve not only multithread interleaving but also the delayed and repeated scheduling of timer handlers. Currently, there are no tools that can effectively detect such bugs. In this study, three types of timer concurrency bugs are summarized: sleeping timer bugs, timer deadlock bugs, and zombie timer bugs. To enhance detection efficiency, firstly, all timer-related code is extracted through pointer analysis, reducing unnecessary analysis overhead. A context-sensitive, path-sensitive, and flow-sensitive interprocedural control flow graph is then constructed to provide a foundation for subsequence analysis. Based on static analysis techniques, including call graph traversal, lockset analysis, points-to analysis, and control flow analysis,

* 基金项目: 国家重点研发计划 (2022YFE0113200); 国家自然科学基金 (U21A20464)
收稿时间: 2023-08-16; 修改时间: 2024-03-04; 采用时间: 2024-12-18; jos 在线出版时间: 2025-05-14
CNKI 网络首发时间: 2025-05-15

three detection algorithms are designed to identify the different types of timer concurrency bugs. To evaluate the effectiveness of the proposed algorithm, they are applied to the Linux 5.15 kernel, where 328 real-world timer concurrency bugs are detected. A total of 56 patches are submitted to the Linux kernel community, with 49 patches merged into the mainline kernel, 295 bugs confirmed and fixed, and 14 CVE identifiers assigned. These results demonstrate the effectiveness of the proposed method. Finally, a systematic analysis of performance, false positives, and false negatives is conducted through comparative experiments, and methods for repairing the three types of bugs are summarized.

Key words: static analysis; Linux kernel timer; concurrency bug; operating system security

随着多处理器的流行, Linux 内核也提供了多种的并发机制. 而在多线程并发执行的场景下, 如果线程间没有进行充分的同步, 很容易产生并发错误. 其中, 定时器是并发错误的重要来源. 具体来说, 定时器并发错误是指在多线程交错执行时, 参与并发执行的一方为定时器处理程序, 并且各线程之间没有采用正确的同步机制而导致的并发错误. 定时器处理程序具有运行在原子上下文、延迟执行和重复调度的特性. 运行在原子上下文是指: 在定时器处理程序调度时, 会关闭本地 CPU 中断, 为避免 CPU 发生阻塞甚至产生死锁, 在定时器处理程序中不允许执行可能导致睡眠的操作. 延迟执行是指: 当定时器激活后, 不会立刻执行其处理程序, 而是先将定时器相关线程挂起, 使其处于睡眠状态, 当计时结束后才会以内核线程的方式执行处理程序. 重复调度是指: 定时器在删除之后, 仍可再次通过其他线程重新激活, 进而再次调度定时器处理程序. 此外, 定时器的激活、调度和删除方式较其他对象更为多样. 如果开发者没有针对定时器处理程序的特性和定时器多样的操作方式来采取正确的同步操作, 则会导致定时器并发错误. 与此同时, 定时器又具有广泛的应用场景, 计算机中的很多任务的调度与执行都是由定时器驱动的, 例如: 监控任务是否超时、延迟执行任务等. 这导致内核中定时器并发错误会直接影响到内核中的多个核心功能. 定时器并发错误具有以下 3 个特点.

(1) 错误种类多样. 由于定时器多样的操作方式和定时器程序处理程序多样的特性, 其产生的并发错误种类也多种多样, 主要包括: 原子上下文睡眠 (sleep-in-atomic-context)、死锁 (deadlock)、释放后使用 (use-after-free)、空指针解引用 (null-pointer-dereference) 等.

(2) 部分错误对系统危害严重. 漏洞 CVE-2016-8655[1]是在 packet 网络协议中由于多个线程并发使用定时器时没有进行较好的同步, 导致定时器 timer_list 结构体中函数指针产生 UAF 错误, 可以实现本地提权. 本文发现的漏洞 CVE-2022-1734[2]是在 NFC 固件下载过程中超时导致的高危漏洞, 攻击者通过在用户态模拟 NFC 设备, 而且无需 root 权限, 触发 UAF 或 double-free 错误, 可导致敏感信息泄露、拒绝服务甚至本地提权. 本文发现的漏洞 CVE-2022-2318[3]是网络栈中 rose 协议由于定时器删除不当, 使定时器处理程序与其他线程产生竞争, 可以在没有任何权限的情况下造成拒绝服务.

(3) 错误检测困难. 由于定时器并发错误不仅涉及多线程交错, 还涉及定时器处理程序的延迟执行与重复调度, 相比其他并发错误更难检测. 此外, 定时器在内核中分布广泛, 经统计, 在 Linux 5.15 内核中共使用了 1 073 个定时器, 分布在网络协议、设备驱动、内存管理、文件系统、虚拟化等百余个子系统与协议中, 现有的内核测试工具无法全面系统地检测定时器并发错误. 例如: 模糊测试工具 Syzkaller[4]由于代码覆盖率有限, 只可以检测到部分定时器相关错误; 静态分析工具 DSAC[5]只针对原子上下文睡眠错误进行了检测.

正因为定时器错误具有错误种类多样性、危害严重性以及检测困难性的特点, 急需有效的应对方法. 针对这个问题, 本文提出了一种可以有效检测定时器并发错误的方法, 对于提升操作系统的机密性、完整性与可用性具有重要意义. 本文基于定时器并发错误的特征, 总结了 3 种定时器并发错误类型, 并针对每种错误类型进行了精确的建模. 具体来说, 定时器检测面临着定时器在内核中分布广泛、错误类型多样并且只有在特定多线程交错和延迟机制参与的场景下才能触发的难题. 为了解决上述难题, 本文首先提取与定时器有关的功能模块, 避免对无关代码进行分析, 提高检测效率; 然后应用间接调用识别算法、路径约束识别算法和无效边过滤算法构建了较为精确的函数调用图; 基于函数调用图和过程内控制流图, 构建了上下文敏感、路径敏感、流敏感的过程间控制流图; 最后综合应用函数调用图遍历、锁集分析、指向分析、控制流分析等静态分析技术, 实现了自动化的定时器并发错误检测方法. 本文实现了一个原型系统, 并通过实验对检测算法的性能、误报和漏报情况进行了评估. 实验结果表

明本文提出的检测方法能够有效地发现定时器并发错误.

本文的主要贡献如下.

(1) 对定时器并发错误进行了精确地分类与建模, 总结了 3 种定时器并发错误类型: 定时器睡眠错误、定时器死锁和僵尸定时器错误.

(2) 设计并且实现了一套基于静态分析的定时器并发错误检测方法和系统, 具有代码覆盖率高、误报率与漏报率较低的特点.

(3) 通过对 Linux 5.15 内核的持续检测, 系统共发现了 328 个新的定时器并发错误, 提交了 56 个补丁进行修复, 有 49 个补丁已经合并到 Linux 主线, 修复了 295 个漏洞. 并系统地总结了针对 3 种定时器并发错误的修复策略和预防方法.

本文第 1 节对国内外内核并发错误检测的研究现状进行阐述. 第 2 节介绍本文研究工作涉及的相关理论技术. 第 3 节阐述 3 种定时器并发错误类型及检测方法. 第 4 节通过对比实验对本文提出的定时器并发错误检测方法的性能、漏报与误报情况等进行评估, 并阐述定时器并发错误详情与修复方法. 第 5 节对本文检测方法的可扩展性与局限性做阐述. 最后, 第 6 节对本文工作进行总结.

1 内核并发错误检测研究现状

目前针对内核并发错误检测的方法包括静态分析、动态分析、混合分析和形式化方法这 4 种类型.

1.1 静态分析

静态分析是指在不执行程序的前提下通过模式匹配对代码进行审计与分析[6], 主要优点有: 可扩展性强、代码覆盖率高、能够在开发阶段的早期发现错误, 从而降低发现错误的成本, 但如果匹配规则定义不当, 容易产生误报和漏报. 目前, 针对内核并发错误检测主要采用了以下静态分析方法: 数据流分析 (data flow analysis)、控制流分析 (control flow analysis)、污点分析 (taint analysis)、锁集分析 (lockset analysis)、指针分析 (pointer analysis)、Happens-before 关系分析等. DCUAF[7]使用 local-global 策略提取内核中可并发执行的接口函数, 然后通过锁集分析算法检测内核驱动中并发释放后使用错误 (CUAF). RELAY[8]使用相对锁集算法在大型程序中检测数据竞争. DSAC[5]使用混合数据流分析和启发式方法对原子上下文中睡眠 (sleep-in-atomic-context) 错误进行了检测. RacerX[9]采用自顶向下的、流敏感、上下文敏感和过程间的静态分析检测内核竞态条件和死锁. Canary[10]分析框架通过构建值流动图, 对多线程间数据依赖关系进行提取, 同时结合 SMT 约束求解等技术, 将并发错误检测转化为“源点-终点”可达性问题, 可以解决传统静态分析在检测并发错误时复杂度过高的问题. O2[11]应用源敏感分析、Happens-before 关系、锁集分析算法对事件驱动的多线程并发场景下的数据竞争进行了检测, 具有检测快速、误报率低的特点. Kahlon 等人[12]通过数据流分析、上下文敏感的别名分析和锁集分析算法对数据竞争进行快速、精确的检测. Dr. Checker[13]应用流敏感、上下文敏感、域敏感的指针分析和污点分析对内核驱动中的竞态条件等 8 类错误进行了检测. DLOS[14]针对内核中 ABBA 型死锁错误进行了检测, 该工具首先使用基于摘要的锁使用分析技术从内核代码中提取包含不同锁的代码路径, 然后使用基于可达性分析的方法从这些代码路径中检测锁环关系, 最后使用二维过滤策略来降低误报. UACatcher[15]使用上下文敏感、流敏感的过程间静态分析技术和指向分析技术定位可能导致 UAC (清理后使用) 错误的资源释放位置和解引用位置, 然后使用考虑了同步操作和路径约束的例程切换算法检测 UAC 错误.

1.2 动态分析

目前, 针对 Linux 内核并发错误的动态检测方法主要包括二进制插桩、模糊测试、系统化调度等.

二进制插桩是指为了检测错误, 在确保原有程序完整性的基础上添加一些探针代码, 来获取程序的执行时的状态信息. KCSAN[16]和 KTSAN[17]是 Google 开发的用于检测内核数据竞争和死锁的消杀器, KCSAN 使用编译时插桩、运行时监测、随机注入延迟的方法检测数据竞争. KTSAN 则是通过编译时插桩、运行时基于 Happens-before 关系检测数据竞争. 二进制插桩的缺点是只能对能够动态执行到的代码进行检测, 并且在内核运行时会导

致较大性能开销.

模糊测试是一种动态软件测试技术, 是指通过自动或半自动的方式构造大量的非预期的随机数据输入到目标程序中, 通过监控程序的异常, 达到检测目标程序中安全漏洞的目的. 使用模糊测试技术不要求使用者对目标程序有很深入的了解, 却是能够发现漏洞最有效和实际的方式, 大部分模糊测试工作都是基于覆盖率引导的. Syzkaller、kAFL[18]等工具使用了基于覆盖率引导的测试方法来提高代码覆盖率. Syzkaller 采用了多线程并行测试的模式通过不断构造系统调用序列对内核进行测试; kAFL 使用了 Intel CPU 的 Intel VT、Intel PML 和 Intel PT 特性加速测试过程. 为了提高测试用例生成效率, Godefroid 等人[19]使用基于神经网络的机器学习方法能够自动生成符合语法规则的输入数据, 对机器学习与模糊测试间的冲突进行了分析, 并提出了 Learn&Fuzz 算法能够智能地引导测试提高覆盖率. SegFuzz[20]将原有较大的线程交错空间分解为较小的交错片段, 使用交错片段覆盖率来引导模糊测试, 通过改变每个交错片段中指令的执行顺序, 对交错片段进行突变发现新的交错场景. 模糊测试的缺点是代码覆盖率有限, 尤其是对于内核驱动程序, 大部分都是依赖于硬件的, 没有硬件设备或虚拟设备的支持则无法对驱动进行测试. 经统计, 被 Syzkaller 测试过的 Linux 内核代码不足总代码量的 20%, 未被测试到的代码仍然占大多数.

系统化调度是指通过控制线程调度实现多线程交错执行, 此方法的优点是可以充分探索多线程交错执行的空间, 快速地复现错误. SKI[21]通过以下两个步骤控制线程调度: ① 将不同的线程固定到一个特定的 CPU 上, 确保同一个线程不会在不同 CPU 间迁移. ② 设计合理的用户态系统化调度算法通过访存指令、HALT、PAUSE 指令暂停或唤醒 CPU 来进行线程调度, 以达到预期的交错效果. SKI 的缺点是只考虑了用户进程上下文中的线程调度, 没有考虑与执行在后台的内核线程之间的交错, 此外 SKI 测试的代码空间有限, 有大量漏报.

1.3 混合分析

混合分析综合了静态分析和动态分析两种技术的优点, 相比单一的测试方法往往能取得更好的效果. Razzer[22]首先通过静态分析获取潜在的存在竞态条件的代码位置, 然后先后通过单线程模糊测试和多线程模糊测试两阶段来寻找数据竞争. Krace[23]首先通过别名分析找到潜在的别名指令对, 然后基于别名覆盖率和代码覆盖率引导的模糊测试, 同时应用 Happens-before 关系、锁集分析算法针对内核文件系统数据竞争进行检测. HFL[24]是基于 Syzkaller 和符号执行工具 S2E[25]提出的基于覆盖率引导的模糊测试工具, 同时使用了模糊测试和符号执行两种技术, 如果模糊测试被严格的条件约束所阻塞则可以使用符号执行继续测试; 如果符号执行遇到状态爆炸的问题则可以使用模糊测试来避免该问题. PLA (probabilistic lockset analysis)[26]是一种基于动态模糊测试, 将语料库中的种子依次与其他随机选取的种子并发执行, 提取具有较高概率并发访问内存场景, 再通过锁集分析与 Happens-before 关系检测数据竞争的方法. 由于是基于动态测试的结果进行静态分析, 该方法较传统锁集分析具有更高的精度, 但存在一定的漏报.

1.4 形式化方法

形式化方法是指针对给定的模型和代码采用基于数学的方法判定其是否满足相关性质, 可分为数学描述和形式推理两个部分, 能够确保所验证的模型和代码一定满足验证得到的结果. 这种方法多用于对可靠性要求非常高的系统进行证明, 如: 航空航天、医疗设备、交通调度等系统. 应用形式化方法进行内核并发错误检测主要包括两种方法: 基于定理证明的方法和基于模型检测的方法[27]. 定理证明是通过数学推理的方法判断目标对象是否满足某一条件约束. 自动化静态分析工具 WHOOP[28]提出了基于定理证明的符号对锁集分析算法来检测内核设备驱动中的数据竞争. 模型检测是指通过形式化的概念 (如操作、状态、事件等) 来描述系统的行为, 进而通过数学的方法检测系统的错误, 它对于验证具有复杂交互式、并发的系统具有较好的精度, 但对于复杂的对象容易产生状态爆炸问题[29]. MOKERT[30]应用模型检测的方法对 Linux 文件系统的并发错误进行了检测.

通过国内外研究现状进行分析可知, 目前还没有专门的分析工具对定时器并发错误进行检测.

2 基础知识

2.1 Linux 内核定时器

Linux 内核定时器是一种软件功能, 它在内核中定时器被广泛使用, 使用定时器的目的有两种[31]: 一是监测超

时, 即判断某事件是否会在预期时间内发生. 二是延迟执行任务, 一旦定时器到期某项工作必须停止或开始. 定时器属于软中断 (softirq) 的一种, 在被激活后需等待一段时间方可执行. 每个定时器都对应着一个或多个定时器处理程序, 同一定时器在不同的使用场景下, 会调用不同的处理程序. 定时器处理程序以内核线程的方式执行, 它不会中断当前 CPU 任务而是等待 CPU 调度器主动调度. 在定时器处理程序执行过程中会关闭本地 CPU 中断, 除非定时器处理程序错误地主动调度, 否则其他线程无法抢占, 从而保证原子性地执行, 即运行在原子上下文中. 定时器处理程序很容易和其他线程产生竞态条件, 任何可能被定时器处理程序和其他线程并发访问的资源都需要通过自旋锁、引用计数等机制进行同步或定义成 atomic_t 类型的原子变量[32].

Linux 内核定时器分为低精度定时器和高精度定时器两种. 对内核定时器的常用操作包括: 定时器初始化、定时器激活、定时器删除、修改定时器到期时间、获取定时器状态等.

(1) 定时器初始化

对低精度定时器初始化的流程包括: 设置定时器处理程序、到期时间、定时器标志、将定时器插入到合适的链表中等. 主要初始化函数及方法包括: ① init_timer(); ② timer_setup(); ③ DEFINE_TIMER(); ④ 直接对 timer_list 结构体成员赋值.

对高精度定时器进行初始化的流程包括: 设置定时器处理程序、到期时间、绑定时钟设备、将定时器加入到红黑树中等. 主要初始化函数为: hrtimer_init(), 但其定时器处理程序需要额外进行初始化, hrtimer_init() 函数无法赋值.

(2) 定时器激活

对低精度定时器激活的流程包括: 开始计时并将定时器从链表中删除. 主要激活函数包括: add_timer()、add_timer_on(), 其中 add_timer_on() 在特定 CPU 上激活定时器.

对高精度定时器激活的流程包括: 开始计时并将定时器从红黑树中删除, 主要激活函数包括: hrtimer_start()、hrtimer_start_range_ns(), 其中 hrtimer_start_range_ns() 可以指定到期范围再运行定时器.

(3) 定时器删除

删除低精度定时器的方法包括: del_timer_sync()、del_timer()、timer_shutdown_sync()、timer_shutdown().

del_timer_sync() 会删除定时器并阻塞等待定时器处理程序执行完成, 由于其阻塞等待的特性, 很容易产生死锁问题. del_timer() 不会阻塞等待定时器处理程序执行结束, 因此使用该函数删除定时器后, 若定时器处理程序正在执行, 很容易与其他线程产生竞态条件. 使用以上两个函数对定时器删除后, 都无法防止定时器从其他位置再次被激活.

timer_shutdown[_sync]() 系列函数是为了删除与其他函数具有循环调用关系的定时器, 防止定时器再次被激活而创建的 API, 往往用于设备移除或驱动模块卸载等场景下. timer_downdown_sync() 会阻塞等待定时器处理程序运行结束并将 timer_list 结构体中指向定时器处理程序的函数指针置 NULL, 从而使得定时器不能再次被激活, 若想再次使用定时器, 必须重新初始化. timer_shutdown() 函数与 timer_shutdown_sync() 函数功能相同, 但不会阻塞等待定时器处理程序运行结束, 往往在 timer_shutdown_sync() 由于锁或上下文限制无法使用的情况下使用.

删除高精度定时器方法包括: hrtimer_cancel()、hrtimer_try_to_cancel(). hrtimer_cancel() 会删除定时器并阻塞等待定时器处理程序执行完成. hrtimer_try_to_cancel() 则不会阻塞等待.

(4) 修改定时器到期时间

修改低精度定时器到期时间的方法为: mod_timer() 或 timer_reduce(). mod_timer() 会先删除定时器、然后修改定时器到期时间、最后激活定时器. timer_reduce() 类似于 mod_timer(), 如果定时器没有激活则它会激活定时器; 如果定时器已经激活则它会重新设置到期时间为更小的值. 高精度定时器无此类操作.

(5) 获取定时器状态

在低精度定时器中, 可通过 timer_pending() 函数判断定时器是否仍在链表中, 如果定时器不在链表中则说明已经激活或已经被删除, 如果定时器在链表中, 则说明其未被激活.

获取高精度定时器状态的函数有 3 个, 分别为 hrtimer_active()、hrtimer_is_queued() 和 hrtimer_callback_

running(). hrtimer_active() 可用于判断定时器是否在红黑树中、定时器处理程序是否正在运行、定时器是否正在迁移至其他 CPU 这 3 种状态; hrtimer_is_queued() 用于判断定时器是否在红黑树上; hrtimer_callback_running() 用于判断定时器处理程序是否正在运行.

2.2 CodeQL 静态分析工具

CodeQL[33]是一款由 GitHub 安全团队开发的静态分析引擎, 可以用于辅助开发者自动检测安全问题, 主要用于变种分析 (variant analysis, 使用一个已知漏洞作为种子, 发现代码中的同类问题). 在 CodeQL 中, 源代码的语法、语义、数据类型、数据流图、控制流图等相关信息被提取到数据库中, 使代码可以被当作数据一样进行查询. CodeQL 支持对多种类型的语言进行查询, 包括: JavaScript、Python、C、C++、C#、Go 等. 图 1 是 CodeQL 的工作流程.

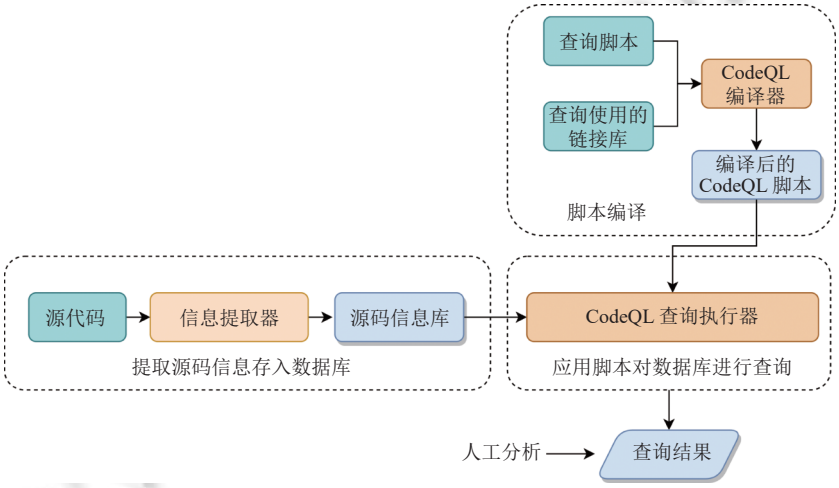


图 1 CodeQL 工作流程

CodeQL 框架由 3 个部分组成.

(1) 数据库创建: 通过信息提取器将源代码相关信息提取到源码信息库中.

(2) 查询脚本编译: 对编写的 QL 查询脚本与脚本使用的链接库进行编译.

(3) 查询脚本执行: 应用 QL 脚本对信息库进行查询, 最后得到查询结果, 得出结果后需要进一步人工分析.

3 定时器并发错误检测

3.1 定时器并发错误种类

本文总结的定时器并发错误包括: 定时器睡眠错误、定时器死锁错误、僵尸定时器错误这 3 种类型. 这些错误可以导致原子上下文睡眠 (sleep-in-atomic-context)、死锁 (deadlock)、释放后使用 (use-after-free) 等问题, 对操作系统安全产生严重威胁.

3.1.1 定时器睡眠错误

定时器睡眠错误是原子上下文睡眠错误 (sleep-in-atomic-context bug) 的一种, 特指在定时器处理程序中调用了可能睡眠的操作. 后文图 2 展示了定时器睡眠错误的形成原因.

定时器睡眠错误在开发阶段不易避免并且在系统实际使用中不易被发现, 主要原因包括: ① 以定时器处理程序为起点的函数调用链往往会很长, 各函数间的调用关系较为复杂, 需要对定时器处理程序能够调用到的每个函数有着深入的理解, 确保其不能睡眠. ② 可能睡眠或线程调度函数种类多样、数量众多, 并且存在大量的函数封装与宏定义, 需要全面地掌握与睡眠相关的函数特征才能避免此类错误. ③ 在系统实际运行过程中, 定时器睡眠错误不一定每次都会出现, 较难观测到, 例如: 带有 GFP_KERNEL 标志的内存分配函数, 只有在系统内存不足的

情况下才会睡眠.

3.1.2 定时器死锁错误

死锁是指在多线程并发执行过程中, 不同线程在竞争有限资源时, 每一个线程都在等待其他线程释放资源而陷入僵持状态. 出现死锁需要满足 4 个必要条件: 互斥访问、持有并等待、资源非抢占、循环等待[34]. 定时器死锁错误是死锁错误的一种, 特指定时器处理程序与具有同步功能的定时器阻塞删除函数竞争同一资源造成的死锁. 图 3 展示了定时器死锁错误的形成原因.

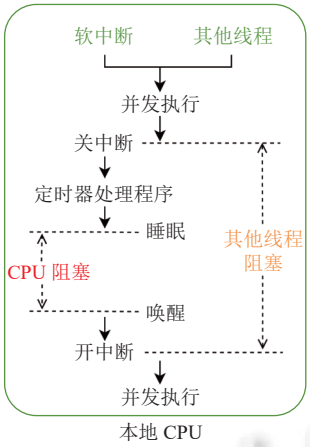


图 2 定时器睡眠错误产生的根本原因

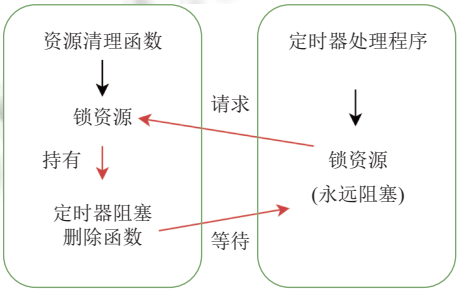


图 3 定时器死锁错误产生的根本原因

定时器阻塞删除函数 (如: del_timer_sync() 等) 会阻塞等待定时器处理程序运行结束. 如果定时器阻塞删除函数持有锁并等待定时器处理程序结束, 而恰好定时器处理程序中也要获取同样的锁则会发生死锁. 定时器死锁错误会使定时器处理程序和资源清理线程都无法正常执行, 占有的资源得不到释放, 使系统中资源利用率降低. 如果得不到释放的资源过多, 可能影响其他线程的正常执行, 最后导致系统崩溃.

3.1.3 僵尸定时器错误

僵尸定时器问题是并发释放后使用错误的一种. 特是指本该被删除的定时器因忘记删除、误用删除操作导致定时器处于僵尸状态, 从而使定时器处理程序与其他线程产生竞态条件. 图 4 展示了僵尸定时器产生的根本原因.

在资源清理函数中, 路径 (1) 表示调用路径上未对定时器进行清理; 路径 (2) 表示调用路径上误用了定时器删除操作, 误用的情况主要包括: 定时器非阻塞删除函数误用和定时器阻塞删除函数误用两种情况. 路径 (3) 表示对定时器进行删除后, 定时器又被重新激活. 在时刻 t 以后, 系统认为定时器已经删除并且定时器处理程序已运行结束, 但实际定时器处理程序仍在运行, 变为僵尸状态. 在资源清理函数中释放资源后, 又在定时器处理程序中对资源进行了解引用, 产生释放后使用错误. 但并非所有的僵尸定时器都会导致内存错误, 在部分情况下, 如果采用了正确的同步操作 (例如: 锁、条件判断等), 即使处于僵尸状态, 也不会产生问题.

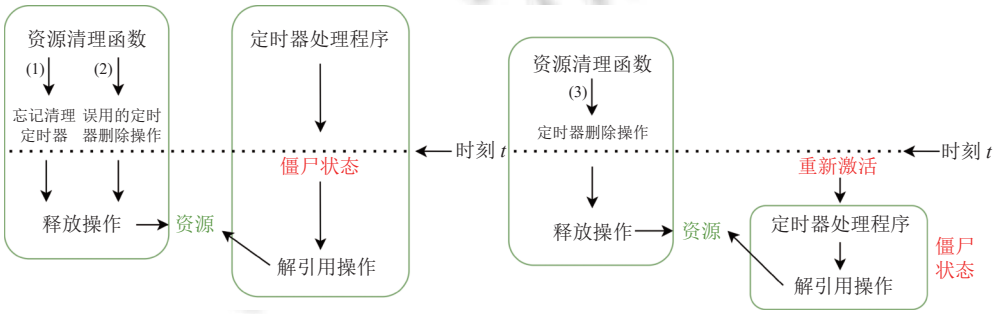


图 4 僵尸定时器错误产生的根本原因

3.2 面临的挑战

对上述 3 类定时器并发错误进行检测面临着以下挑战.

挑战 1: 内核代码数量巨大、各模块间的关系复杂, 并非每一个模块都使用了定时器, 如何提取有价值的目标构建数据库, 同时保证数据的完备性, 提高检测效率是一个挑战.

挑战 2: 内核中存在大量的间接函数调用、复杂的条件判断和大量重名函数, 如何建立相对精确的函数调用图与过程间控制流图, 提高静态分析的精度是一个挑战.

挑战 3: 内核定时器种类多、分布范围广, 如何准确识别内核中所有定时器, 并建立定时器与定时器处理程序之间的关系为后续分析奠定基础是一个挑战.

挑战 4: 本文总结了 3 种定时器并发错误, 如何针对每种错误的特点设计合理的检测算法, 全面地对错误进行检测, 并降低漏报与误报是一个挑战.

3.3 检测方法总体框架

为解决以上 4 个挑战, 本文分别采取以下 4 个方法来解决.

(1) 为提高数据检索效率与精度, 依据定时器位置信息制定信息库构建策略, 提取内核中所有和定时器相关的功能模块, 编译生成 CodeQL 数据库, 提取语法、语义、数据类型、抽象语法树、数据流、控制流等信息, 从而避免对无用的目标对象进行分析, 提高分析效率与精度. 此过程对应图 5 中的信息库构建阶段.

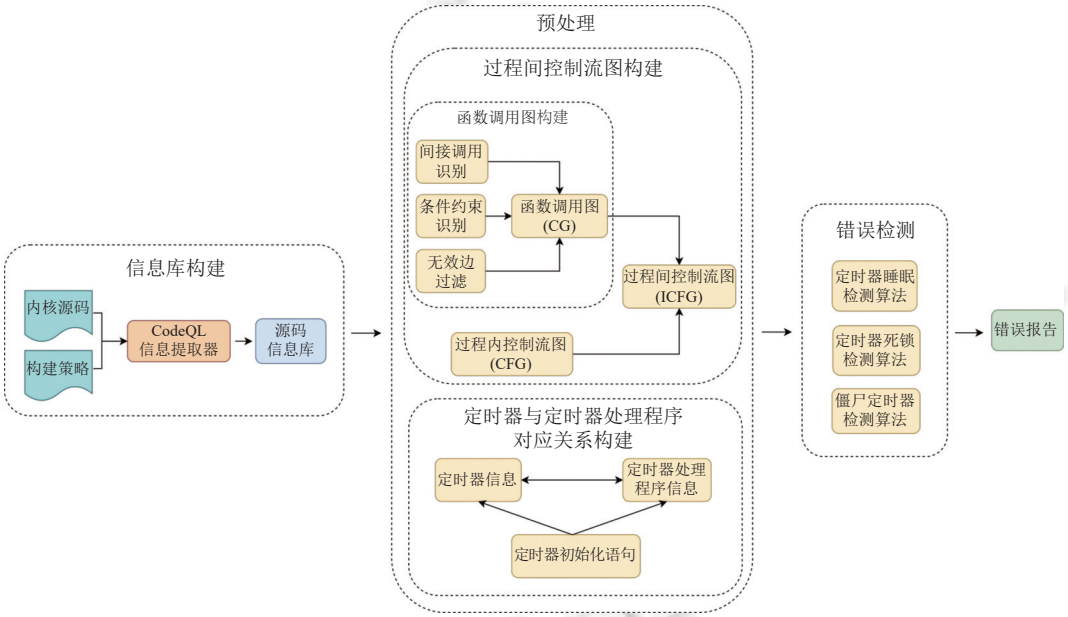


图 5 定时器并发错误检测总体框架

(2) 为构建精确的函数调用图, 采用数据流分析与指针类型匹配相结合的方式进行间接函数调用识别, 以提高间接调用识别精度; 通过控制流分析对条件判断语句等路径约束进行识别, 实现函数调用图的上下文敏感与路径敏感; 通过无效边过滤算法对静态分析建立的但实际执行不到的边进行过滤, 进一步提升函数调用图的精度; 以上过程对应预处理阶段函数调用图构建模块. 为构建过程间控制流图, 本文以基本块为单位, 采用深度优先遍历的方式, 基于过程内控制流图与函数调用图生成上下文敏感、路径敏感与流敏感的过程间控制流图, 此过程对应预处理阶段的过程间控制流图构建模块.

(3) 为识别定时器并建立定时器与处理程序之间的关系, 通过模式匹配的方法, 提取内核中所有的定时器初始化语句, 本文通过初始化语句来识别所有的定时器、并建立定时器与定时器处理程序对应关系, 为后续分析提供

基础. 此过程对应图 5 中预处理阶段的定时器与定时器处理程序对应关系构建模块.

(4) 为精确地对错误进行全面地检测, 本文依次通过定时器睡眠检测算法、定时器死锁检测算法、僵尸定时器检测算法对定时器 3 类并发错误进行检测, 最后生成报告. 此过程对应图 5 中错误检测阶段.

3.4 信息库构建

经调研, 内核中有很多功能模块未使用任何定时器, 例如: 加密与认证模块、部分设备驱动程序等. 如果基于全内核进行定时器并发错误检测, 不仅效率较低而且会因间接调用识别不精确等因素产生较多的误报.

图 6 显示了 Linux 5.15 中内核定时器分布情况. 由统计结果可知 Linux 5.15 内核 x86 架构下共使用了 1 073 个定时器, 大部分都分布在有限的几个模块中. 因此, 我们可以过滤掉无用的目标对象, 提取和定时器有关的功能模块进行分析, 对提高静态分析效率与精度具有重要意义. 图 7 是构建目标数据库的主要流程.

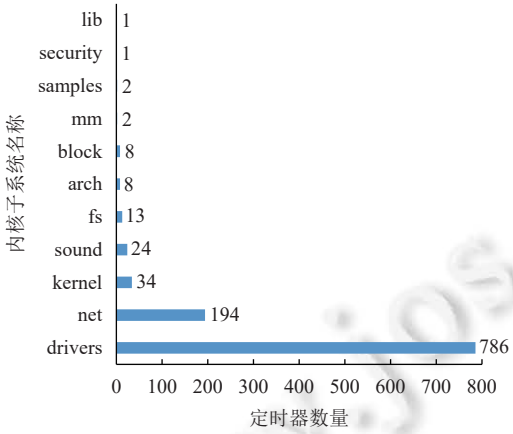


图 6 Linux 5.15 内核定时器分布情况

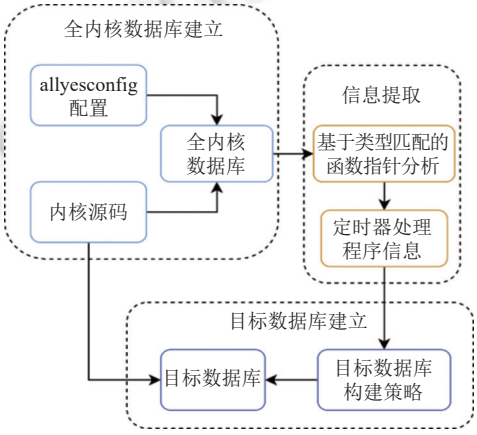


图 7 CodeQL 目标数据库构建流程

① 首先配置内核选项为 allyesconfig, 建立全内核数据库, 确保不遗漏任何和定时器有关的功能模块.

② 在信息提取阶段, 通过基于类型匹配的方法对低精度定时器 timer_list 结构体和高精度定时器 hrtimer 结构体中的函数指针进行分析, 匹配函数返回值、参数类型与参数个数和定时器结构体中函数指针成员相同的为定时器处理程序. 低精度定时器函数指针返回值为 void 类型, 有 1 个 timer_list 类型的参数; 高精度定时器函数指针返回值为 enum hrtimer_restart 类型, 有 1 个 hrtimer 类型的参数. 类型匹配过程如图 8 所示.

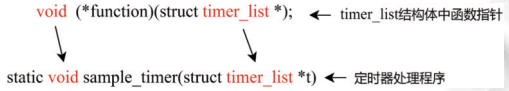


图 8 通过指针类型匹配提取定时器处理程序示例

通过类型匹配得出在 Linux 5.15 内核 x86 架构下共使用了 1 073 个定时器, 由于参数类型较为特殊, 经人工审查, 不存在误报.

③ 在目标数据库建立阶段, 数据库构建策略为: 依据提取的定时器处理程序信息, 确定定时器所在功能模块, 依据功能模块不同, 分别编译源码生成多个数据库. 通过此种方式不仅可以降低静态分析检索范围提高结果的准确率, 而且可以并行检索多个数据库来提高静态分析的效率.

3.5 预处理

预处理阶段主要目的是建立各类数据之间的关系, 为后续错误检测阶段提供基础. 主要包括以下 3 个部分: ① 函数调用图构建; ② 过程间控制流图构建; ③ 定时器与定时器处理程序关系构建.

3.5.1 函数调用图构建

为构建精确的函数调用图, 本文基于数据流分析与指针类型匹配的间接调用识别算法、基于控制流分析的路

径约束识别算法与基于程序局部性原理的无效边过滤算法建立函数之间的调用关系.

(1) 间接函数调用识别

本文采用数据流分析与基于文件位置信息辅助的多级指针类型匹配相结合的算法识别间接函数调用, 如图 9 所示.

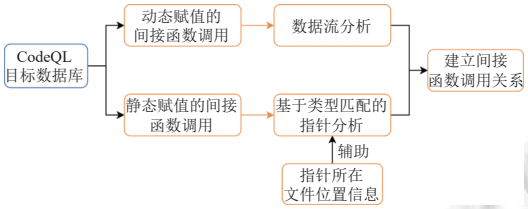


图 9 间接函数调用识别与调用关系建立

当函数指针作为函数参数时, 其指向的值是在程序运行过程中通过参数传递的方式动态赋予的, 因此采用数据流分析的方法进行识别. 当函数指针位于结构体中时, 其指向的值大部分情况下是通过结构体初始化语句以静态的方式赋予的, 本文通过多级指针类型匹配同时结合指针所在文件位置信息识别间接调用.

如图 10 所示, 为寻找函数指针 sk->sk_state_change 指向的函数, 通过匹配函数指针名称和所在结构体类型, 寻找所有对该指针进行初始化的语句, 但同一个函数指针在不同的场景下会被赋予不同的值. 为进一步提高间接调用识别的精度, 本文通过提取函数指针所在文件位置与各函数指针初始化语句所在文件位置, 在所有可能的初始化语句中, 选择所在文件位置和当前函数指针文件位置具有最长公共前缀的语句作为当前函数指针的初始化语句, 因此函数指针指向的函数为 vcc_def_wakeup().

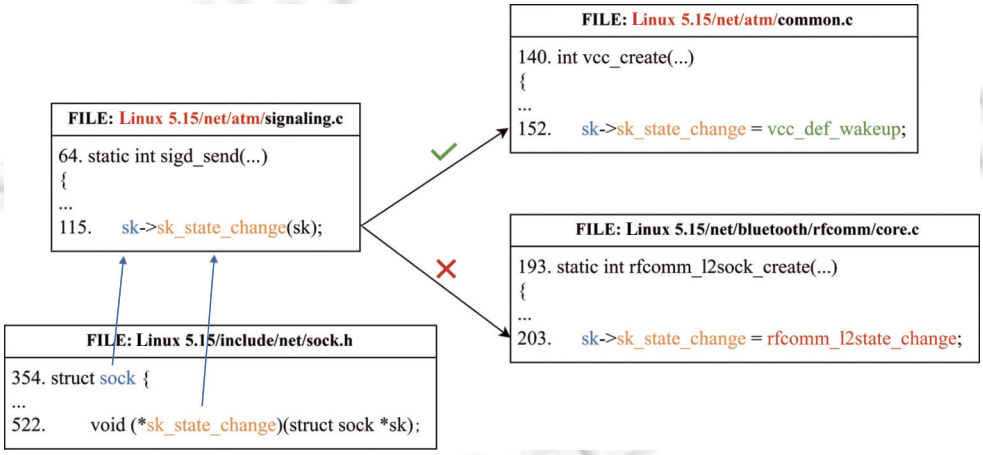


图 10 通过指针类型匹配及文件位置信息识别间接调用

(2) 路径约束识别

在内核代码中有大量 if 与 switch 条件判断语句, 在特定的函数调用上下文中, 条件判断语句能走到的分支是确定的. 图 11 分别展示了 if 和 switch 语句对控制流的影响, 图 11(a) 中函数 B 在被函数 A 调用时, 仅能调用到 D 函数; 图 11(b) 中函数 B 在被函数 A 调用时, 仅能调用到 C 函数.

为构建上下文敏感与路径敏感的函数调用图, 本文应用 CodeQL 提供的 Guards 模块[35]对函数参数中具有常量, 并且函数内有 if 或 switch 语句使用了该常量参数作为判断条件的情况进行了控制流分析.

如果函数调用的常量参数满足其内部 if 判断条件, 则在当前上下文中, 被调用者函数到 if 所在分支函数具有调用关系; 如果不满足, 则到 else 所在分支函数具有调用关系.

如果函数调用的常量参数满足其内部某个 switch-case 判断值, 则在当前上下文中, 被调用者函数到 case 所在分支函数具有调用关系.

(3) 基于局部性原理过滤无效边

由于内核不同文件中具有大量重名的函数, 通过上述方法建立的某些函数调用关系在程序实际执行过程中却执行不到, 如图 12 所示.

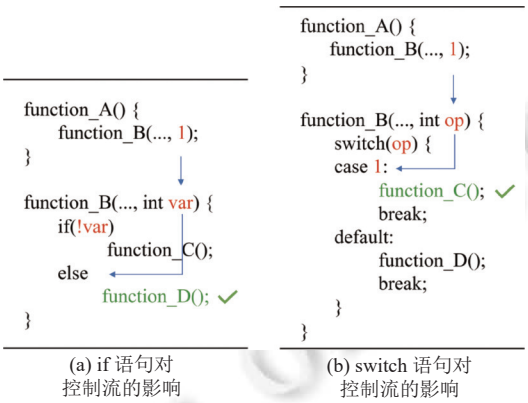


图 11 if 与 switch 语句对控制流的影响

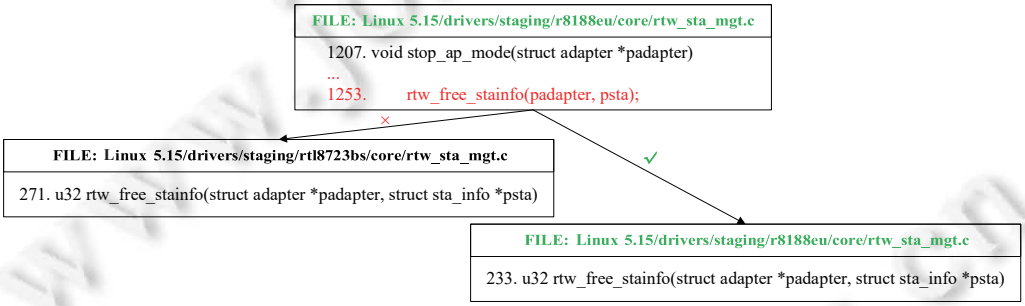


图 12 因函数重名建立的无效函数调用关系示例

无效边过滤算法的核心思想为: 基于程序局部性原理, 若被调用者函数在多个文件中具有相同的名称, 根据当前调用者函数所在文件位置, 寻找被调用者函数位置与当前调用者函数位置具有最长公共前缀的函数作为被调用者函数. 图 12 中右侧函数 rtw_free_stainfo() 与调用者函数 stop_ap_mode() 在同一文件中, 因此为正确的被调用者函数.

3.5.2 过程间控制流图构建

本文基于 CodeQL 提供的过程内控制流图和第 3.5.1 节构建的函数调用图, 以基本块为粒度, 采用深度优先遍历的方式, 生成过程间控制流图. 算法将没有被其他函数调用的函数作为起始函数, 由于本文仅针对定时器进行分析, 因此仅提取与定时器相关的起始函数即可. 本文依次从定时器删除函数、定时器激活函数出发, 通过反向遍历函数调用图的方式, 提取与定时器相关的起始函数. 然后依次以提取的起始函数为起点, 以基本块为粒度, 进行深度优先遍历. 在遍历过程中, 分为以下 3 种情况.

① 若当前基本块包含函数调用语句, 则将当前基本块的后继基本块入栈; 然后根据函数调用关系获取被调用者函数, 并获取被调用者函数起始基本块 Entry, 添加从当前基本块到 Entry 之间的边; 若 Entry 没有被访问过, 则从 Entry 继续遍历.

② 若当前基本块为函数中最后一个退出基本块, 并且栈不为空则将下一个要执行的基本块 successor 出栈, 添

加从退出基本块到 successor 的边; 若 successor 没有被访问过, 则从其继续遍历.

③ 若当前基本块为普通的基本块, 则获取其所有后继基本块, 添加从当前基本块到后继基本块之间的边; 如果后继基本块没有被访问过, 则从后继基本块继续遍历.

在当前起始函数遍历结束后, 获取下一个起始函数继续遍历. 当所有起始函数均遍历结束后, 过程间控制流图构建完毕.

3.5.3 定时器与定时器处理程序关系构建

建立定时器与定时器处理程序的关系可分为两步: 首先需要对定时器和定时器处理程序进标识; 然后通过定时器初始化语句建立两者之间的关系.

(1) 标识定时器

当定时器作为全局变量时, 会声明为 static 类型, 即在本文中是唯一的, 因此可通过定时器名称与定时器所在文件位置唯一标识一个全局变量定时器.

当定时器作为结构体成员时, 由于同一结构体可能在多个不同的模块中使用, 本文通过定时器名称、父结构体类型与所在模块信息唯一标识一个定时器. 其中, 所在模块信息是指定时器所在文件的上一级目录, 例如: net/ax25/af_ax25.c 文件中使用了 ax25->t1timer, 则该定时器所在模块为 net/ax25.

(2) 标识定时器处理程序

由于不同文件中定时器处理程序可能具有相同的名字, 本文通过定时器处理程序名称与定时器处理程序所在文件位置唯一标识一个定时器处理程序.

(3) 定时器与定时器处理程序关系构建

本文通过定时器初始化语句建立定时器与定时器处理程序之间关系. 内核定时器的初始化方法包括 3 种: ① 通过宏 timer_setup() 进行赋值. ② 通过宏 DEFINE_TIMER 进行赋值. ③ 通过赋值语句进行赋值. 图 13 为 3 种初始化方法的示例.

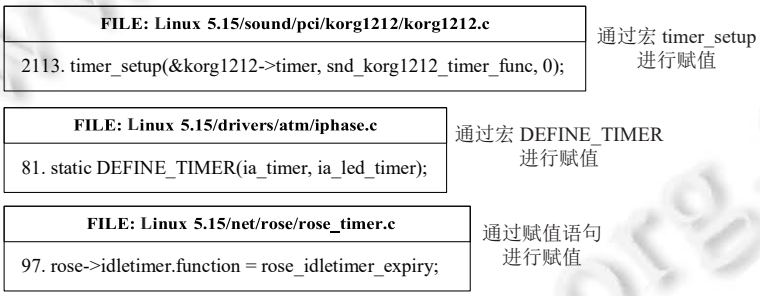


图 13 定时器 3 种初始化方法

对于 timer_setup() 宏和 DEFINE_TIMER() 宏, 第 1 个参数为定时器, 第 2 个参数为定时器处理程序; 对于赋值语句, 左值为定时器, 右值为定时器处理程序.

经调研发现, 少部分作为结构体成员的定时器在初始化时会首先赋值给一个局部指针变量, 然后对局部指针变量进行初始化, 此类情况无法通过定时器初始化语句建立二者之间的关系, 如图 14 所示.

```
FILE: Linux 5.15/drivers/scsi/hisi_sas/hisi_sas_v1_hw.c
807. static void phys_init_v1_hw(struct hisi_hba *hisi_hba)
...
810.        struct timer_list *timer = &hisi_hba->timer;
...
817.        timer_setup(timer, start_phys_v1_hw, 0);
...
```

图 14 通过局部指针变量对定时器进行初始化

为全面建立定时器与处理程序之间的关系, 对于通过 timer_setup() 进行初始化的情况, 首先提取 timer_setup() 宏的第 1 个参数, 判断是否为局部变量; 如果是局部变量, 则获取局部变量赋值表达式的右值, 提取定时器名称、父结构体类型和所在模块信息, 并建立与定时器处理程序的关系; 否则, 根据 timer_setup() 语句直接建立定时器与定时器处理程序之间的关系.

3.6 检测算法

3.6.1 定时器睡眠错误检测算法

本文通过函数调用图遍历的方式检测定时器睡眠错误. 首先需要对原子上下文中可能睡眠函数进行准确识别, 然后以定时器处理程序为起点遍历函数调用图, 如果定时器处理程序能调用到可能或一定睡眠的函数则认为存在定时器睡眠错误.

(1) 可能睡眠的函数提取

可能睡眠的函数包括两类: ① 内存管理相关的函数. 对于连续物理内存分配函数 (如 kmalloc() 等), 如果其参数使用了原子上下文内存分配标志 (如: GFP_ATOMIC 和 GFP_NOWAIT), 不论是单独使用或与其他内存分配标志组合使用, 本文认为该内存分配函数不会睡眠, 否则认为该内存分配函数可能睡眠. 对于虚拟内存分配函数 (如: vmalloc()、vzalloc() 等), 在系统可用内存空间不足时可能睡眠. ② 同步相关操作. 对于互斥锁 (如: mutex_lock() 等), 如果一个进程试图获取一个被占用的 mutex 锁时, 该进程会进入睡眠状态, 直到 mutex 锁释放. 对于信号量操作 (如: down_killable()、down_interruptible() 等), 如果进程试图获取一个已经被占用的信号量时, 该进程会进入睡眠状态, 直到该信号量释放. 对于阻塞等待函数 (如: cancel_work_sync()、synchronize_rcu()、synchronize_irq() 等), 如果该函数等待的事件没有结束, 则会发生调度或睡眠.

(2) 一定睡眠的函数提取

一定睡眠的函数包括两类: ① 线程调度函数 (如: schedule()、schedule_timeout() 等). ② 睡眠等待函数 (如: msleep()、usleep_range() 等). 以上两类函数会使当前线程直接睡眠.

(3) 函数调用图遍历

算法中以定时器处理程序为起点, 采用深度优先搜索遍历函数调用图, 寻找可能或一定睡眠的函数, 如果路径上遇到内核线程调度函数 (如: 工作队列等) 则终止在当前路径上进行搜索, 因为内核线程运行在进程上下文中允许睡眠操作. 收集所有从定时器处理程序到可能睡眠函数之间的路径.

(4) 错误过滤

本文通过实例发现, 如果调用路径上存在以下两类情况则不会导致错误: ① 存在中断上下文判断语句 in_interrupt()、in_softirq() 等, 会检测当前是否处于中断上下文中, 从而避免调用可能睡眠函数; ② 存在开中断语句 disable_irq_nosync(), 则会改变当前上下文类型, 在其后调用可能睡眠的函数则不会产生错误.

为进一步降低误报, 本文基于检测出的可能睡眠的调用路径, 以可能睡眠的函数为起点, 反向遍历控制流图. 如果调用路径上存在中断上下文判断语句或开中断语句则认为是误报, 将其进行过滤.

3.6.2 定时器死锁错误检测算法

本文通过锁集分析 (lockset analysis) 算法检测定时器死锁错误, 如果定时器阻塞删除函数当前持有的锁和定时器处理程序中要获取的锁有交集则会产生定时器死锁错误. 首先通过模式匹配提取定时器阻塞删除函数及其封装函数; 然后对以上函数进行指向分析 (pointsto analysis) 提取定时器信息, 进而获取其处理程序信息. 为确保定时器删除函数所在线程与定时器处理程序所在线程是可并发的, 需进一步进行可并发性过滤. 最后通过锁集分析算法对定时器死锁错误进行检测. 定时器死锁错误检测流程如图 15 所示.

(1) 定时器阻塞删除函数及其封装函数提取

本文总结了 3 类定时器阻塞删除函数: del_timer_sync()、hrtimer_cancel() 和 timer_shutdown_sync(). 此外, 一些函数对上述函数中进行了封装, 通过定时器阻塞删除函数无法识别正在删除的定时器, 需对封装函数的参数进行指向分析才能识别正在删除的定时器, 如图 16 所示.

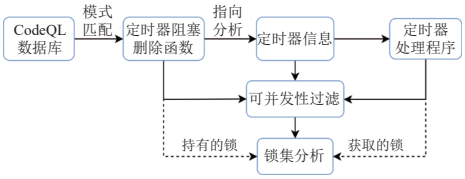


图 15 定时器死锁错误检测流程

FILE: Linux 5.15/net/core/sock.c

```
void sk_stop_timer_sync(struct sock *sk, struct timer_list *timer)
{
    if (del_timer_sync(timer))
        __sock_put(sk);
}
```

图 16 定时器阻塞删除函数封装函数示例

(2) 提取删除函数对应的处理程序

本文通过对定时器阻塞删除函数的参数进行指向分析, 提取阻塞删除函数删除的定时器信息. 然后依据预处理阶段建立的定时器与定时器处理程序关系, 获取对应的定时器处理程序信息.

(3) 可并发性过滤

为确保定时器删除函数所在线程与定时器处理程序是可并发的, 本文通过遍历函数调用图的方式进行可达性分析, 查看定时器处理程序能否调用到定时器删除函数, 如果可以调用到, 则认为二者同属一个线程, 是不可并发的, 将其进行过滤. 否则认为是可并发执行的, 需进一步进行锁集分析.

(4) 锁集分析

分别提取定时器阻塞删除函数持有的锁的集合和定时器处理程序要获取的锁的集合, 若二者锁集有交集则可能产生死锁. 进行锁集分析首先需要确定要分析的锁的类型. 由于定时器处理程序内只允许调用自旋锁 (spin_lock) 和读写锁 (rwlock), 因此锁集分析只关心这两种类型的锁即可. 此外, 内核中很多锁是以宏的方式定义的 (如: raw_spin_lock、write_lock 等), 因此进行锁集分析时需要以宏展开后的锁形式为准. 算法 1 展示了锁集分析的流程.

算法 1. 锁集分析算法.

Input: 定时器处理程序 *handler*, 定时器阻塞删除函数 *blockFunc*;
Output: 当前输入是否会产生死锁 True/False.

```
1. Function GetLock(handler)
2.     Lock = 从 handler 开始, 遍历控制流图, 获取 handler 中加锁操作语句
3.     foreach item in Lock do
4.         Set_Lock = Set_Lock ∪ item.PointsTo()
5.     end
6.     return Set_Lock
7. end
8. Function HoldLock(blockFunc)
9.     Lock = ∅, UnLock = ∅, Set_Lock = ∅, result = ∅
10.    Lock = 从 blockFunc 开始, 反向遍历控制流图, 获取加锁操作语句
11.    foreach item in Lock do
12.        hold[item.PointsTo()] = True
13.        Set_Lock = Set_Lock ∪ item.PointsTo()
14.    end
15.    UnLock = 从 blockFunc 开始, 反向遍历控制流图, 获取解锁操作语句
16.    foreach item in UnLock do
17.        if item.PointsTo() in Set_Lock then
18.            hold[item.PointsTo()] = False
19.        end
```

```
    end
    foreach item in Set_Lock do
        if hold[item] == True then
            result = result ∪ item
        end
    end
    return result
end
Function DetectDeadLock()
    result = GetLock(handler)∩HoldLock(blockFunc)
    if result ≠ ∅ then
        return True
    else
        return False
    end
end
```

GetLock() 函数用于获取定时器处理程序要获取的锁的集合. 从定时器处理程序开始自顶向下遍历过程间控制流图提取加锁操作语句; 然后通过指向分析提取对应的自旋锁和读写锁.

HoldLock() 函数用于获取定时器阻塞删除函数持有的锁. 以定时器阻塞删除函数为起点反向自底向上遍历控制流图, 提取阻塞删除函数获取过的锁和解除的锁, 没有被解除的锁即为当前函数持有的锁.

在遍历过程间控制流图过程中, 会存在路径爆炸问题. 本文通过实践发现, 当函数遍历的层数为 2 层以上时, 二者的锁集不再有新增的交集. 因此本文限制遍历的层数为 2 来避免路径爆炸问题.

DetectDeadLock() 函数用于查看同一定时器对应的阻塞删除函数持有的锁集与处理程序要获取的锁集是否有交集, 如果有交集则可能产生死锁问题.

3.6.3 僵尸定时器错误检测算法

僵尸定时器错误分为定时器删除函数误用、定时器重新激活和忘记清理定时器这 3 类, 针对 3 类不同的错误, 本文分别设计了 3 个算法进行检测.

(1) 定时器删除函数误用

定时器删除函数误用包括两种情况: ① 定时器阻塞删除函数前存在错误的定时器状态判断语句, 使阻塞删除函数被绕过, 导致资源释放后又在处理程序中解引用; ② 使用非阻塞删除函数无法删除正在运行的定时器, 从而导致资源释放后又在处理程序中解引用. 为检测以上两类错误, 检测算法流程如图 17 所示.

算法首先通过控制流图遍历、模式匹配和指向分析技术, 提取可能产生问题的定时器删除函数和对应的定时器处理程序, 然后通过控制流图遍历提取定时器删除函数后资源释放操作和定时器处理程序中解引用操作, 通过指向分析提取可能产生问题的对象, 进一步通过引用计数对象过滤和锁集分析生成错误报告.

● 提取可能产生问题的定时器删除函数与对应的处理程序

算法通过模式匹配和反向遍历控制流图的方法分别提取可能产生问题的定时器阻塞删除函数集合 S1 和定时器非阻塞删除函数集合 S2. 然后通过 CodeQL 提供的 PointsTo 模块进行指向分析提取被删除的定时器, 并根据二者之间的关系获取对应的定时器处理程序.

● 解引用操作和资源释放操作提取

本文采用控制流图遍历和模式匹配的方法提取指针解引用操作和可能产生问题的定时器删除函数后所有的资源释放操作, 本文总结的内核中常用的资源释放操作包括: kfree()、kvfree()、kfree_const()、kfree_skb()、

kfree_sensitive()、kmem_cache_free()、devlink_free()、free_netdev()、release_firmware()、sk_free()、release_region() 这 11 种.

首先对定时器处理程序中所有指针解引用操作进行指向分析, 提取解引用的对象. 然后对资源释放操作中的参数进行指向分析, 提取释放的对象. 最后选取既进行了释放操作又进行了解引用操作的对象作为可能产生问题的对象.

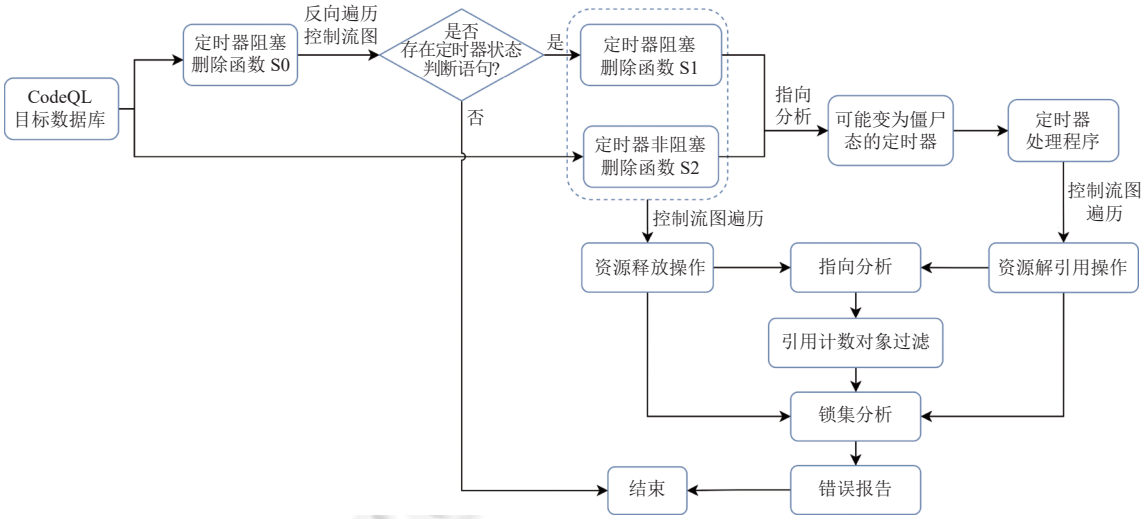


图 17 定时器删除函数误用检测流程

● 引用计数对象过滤

在 Linux 内核中引用计数通过以下 5 类结构体实现: refcount_t、atomic_t、atomic64_t、atomic_long_t 和 kref, 若在结构体中嵌入了以上 5 类结构体之一, 则认为资源使用了引用计数进行管理. 本文通过调研内核中大量实例发现: 对于使用引用计数管理的资源, 若在定时器处理程序中存在对该资源的解引用操作, 但没有对引用计数增加, 则可能会导致问题. 因为在定时器处理程序中解引用该资源之前, 资源的引用计数可能降为 0 并释放, 之后在定时器处理程序中解引用就会产生 UAF 错误. 因此, 若使用引用计数管理的资源在定时器处理程序中进行了使用, 并且在定时器处理程序中没有对该资源引用计数的增加操作, 则本文将其作为可能产生问题的对象.

● 锁集分析

本文通过锁集分析判断对同一资源的释放操作持有的锁和解引用操作持有的锁是否有交集来判断是否会产生错误. 本文分析的锁的类型包括互斥锁、自旋锁、读写锁. 为了获取二者持有的锁, 本文分别从资源释放操作和资源解引用操作出发, 反向遍历控制流图, 获取持有的锁; 若解引用操作有多个, 则只从第 1 个解引用操作出发进行反向遍历, 用第 1 个解引用操作持有的锁代表所有解引用操作持有的锁. 如果资源清理操作持有的锁和资源解引用操作持有的锁没有交集, 则认为会产生错误; 如果有交集, 则认为不会产生错误.

(2) 定时器重新激活

本文首先通过模式匹配方式提取定时器阻塞删除函数和定时器激活函数, 并通过指向分析提取对同一定时器的删除与激活函数对. 然后通过控制流图遍历的方式收集从定时器阻塞删除函数 (del_timer_sync() 和 hrtimer_cancel() 两类函数. 本文默认 timer_shutdown[_sync] 系列函数不会产生重新激活问题) 至资源清理函数之间获取的锁、从定时器激活函数至边界函数之间获取的锁; 最后判断两个锁集是否存在交集, 若不存在交集, 则认为定时器可被重新激活. 最后, 进一步对可被重新激活的定时器进行过滤, 筛选出因定时器重新激活而产生的僵尸定时器错误. 检测流程如图 18 所示.

● 资源清理函数提取

本文所指的资源清理函数包括设备移除函数和驱动模块卸载函数两种. 本文仅关注设备移除函数和驱动模块

卸载函数的原因为: 当设备移除或驱动模块卸载时, 需要将定时器从系统中彻底删除, 并且两种函数内部具有大量的资源释放操作, 如果再次激活定时器则很可能会产生竞态条件. 而在其他场景下, 定时器往往无需彻底删除, 而是经常与其他函数配合实现某些功能.

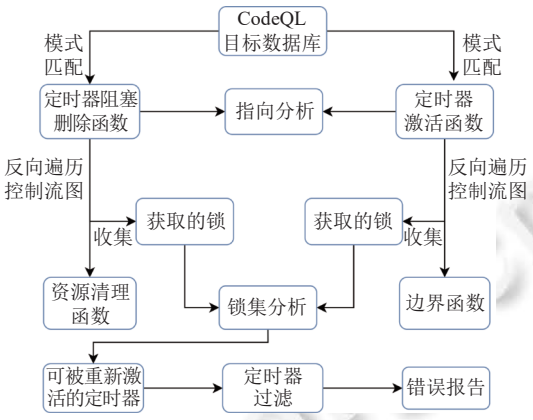


图 18 定时器重新激活错误检测流程

经过大量实例调研, 本文总结出设备移除函数具有以下 3 个特点: ① 一般带有 unregister、kill、remove、close、disconnect、detach、shutdown、stop、power_off 等关键字; ② 函数的返回值类型为 void 类型; ③ 只具有一个参数, 此参数为与设备相关的结构体. 例如: static void ax25_kill_by_device(struct net_device *dev)、void nfc_unregister_device(struct nfc_dev *dev) 等.

Linux 内核中通过 device 结构体描述了所有设备的核心特性, device 结构体中包含了对设备进行建模所需要的核心信息. 基于 device 结构体, 内核会实例化不同功能的设备, 例如: nfc_dev (NFC 设备)、hci_dev (蓝牙设备)、usb_device (usb 设备)、pci_dev (PCI 设备) 等, 在这些结构体中都对 device 结构体进行了封装, 本文将所有结构体成员 (包括多级子结构体成员) 中包含 device 的结构体作为与设备相关的结构体, 如图 19 所示.

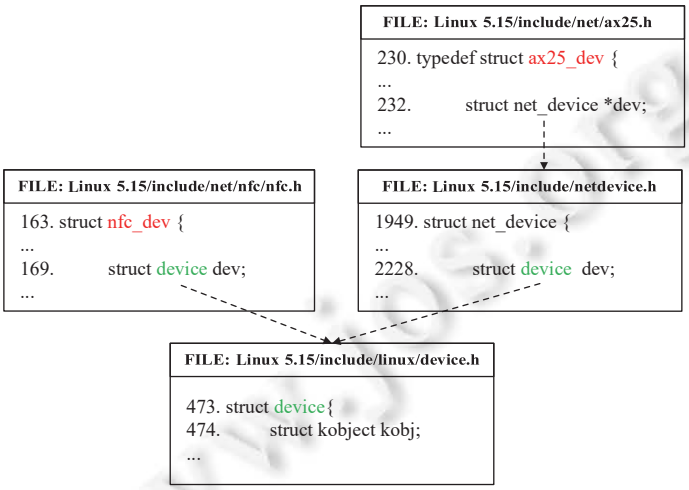


图 19 内核中与设备相关的结构体示例

对于驱动模块卸载函数, 一般具有以下 3 个特点: ① 返回值一般为 void 类型; ② 参数个数为 0 (用 void 表示); ③ 通过__exit 宏进行修饰. 例如: static void __exit idt77252_exit(void) 等. 本文将符合以上 3 个特点的函数作为驱动模块卸载函数.

● 边界函数提取

本文通过调研内核中大量实例发现, 定时器被重新激活主要分为两种情况, 一是用户通过系统调用进行重新激活; 二是一些异步机制 (如工作队列、tasklet) 与定时器具有循环调用关系, 已经删除的定时器可以从这些异步机制中再次激活.

因此, 本文所指的边界函数包括两类: ① 实现系统调用具体功能的内核接口函数; ② 与定时器具有循环调用关系的工作者线程、tasklet 回调函数. 其中, 内核接口函数为系统调用在不同场景下使用的具体实现, 与系统调用的关系如图 20 所示.

对于与定时器具有循环调用关系的工作者线程和 tasklet 回调函数, 本文通过集合运算的方式进行提取. 如图 21 所示, 集合 S1 为能重新激活定时器的工作者线程或 tasklet 回调函数, 集合 S2 为能被定时器激活的工作者线程或 tasklet 回调函数, S1∩S2 为与定时器具有循环调用关系的工作者线程或 tasklet 回调函数.

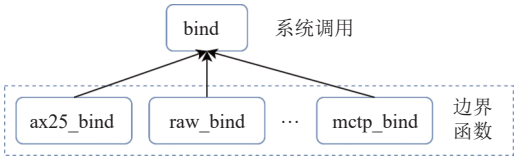


图 20 边界函数与系统调用的关系

图 21 与定时器具有循环依赖关系的边界函数

● 锁集分析

分别从对同一定时器的阻塞删除函数和激活函数出发, 反向遍历控制流图, 收集从定时器阻塞删除函数到资源清理函数之间所有获取的锁、从定时器激活函数到边界函数之间所有获取的锁, 本文关注的锁包括: 读写锁、互斥锁、自旋锁这 3 种情况. 反向遍历控制流图的原因为: 正向遍历搜索空间较大, 不确定通过哪一个边界函数才能调用到定时器激活函数, 需要将所有边界函数全部遍历才能确定, 相比反向遍历要耗费更多的时间和资源. 为了提取获取的锁, 本文首先提取路径上加锁操作语句, 然后通过指向分析提取获取的锁. 最后判断提取的两个锁集是否存在交集, 若不存在交集, 则认为定时器会被重新激活.

● 定时器过滤

基于可被重新激活的定时器集合, 分别提取对应的定时器处理程序和定时器阻塞删除函数, 采用与定时器删除函数误用相同的检测流程, 依次通过控制流图遍历、指向分析、引用计数对象过滤与锁集分析的方式提取可能产生错误的定时器, 最后生成错误报告.

(3) 忘记清理定时器

本文通过差集运算的方式提取忘记清理的定时器. 首先提取内核中所有的定时器, 记为集合 S1; 然后提取内核中所有定时器删除函数及删除函数的封装函数 (以参数传递的方式将定时器结构体传递给定时器删除函数, 如图 16 所示), 并对定时器删除函数及封装函数的参数进行指向分析, 提取所有执行过删除操作的定时器, 记为集合 S2. 最后通过差集运算 S1–S2 得到的结果集合即为忘记清理的定时器集合. 如图 22 所示.

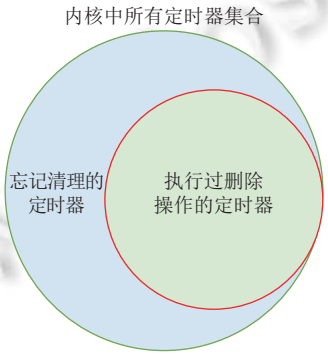


图 22 忘记清理的定时器示意图

并非所有的忘记清理的定时器都会导致错误, 如果采用了较好的同步机制则不会产生问题. 为检测可能产生错误的定时器, 本文通过定时器信息提取对应的处理程序和所在功能模块内资源清理函数 (设备移除函数和驱动模块卸载函数), 分别提取定时器处理程序中资源解引用操作和资源清理函数中资源释放操作. 然后采用与定时器删除函数误用相同的检测流程, 依次通过指向分析、引用计数对象过滤与锁集分析的方式提取可能产生错误的定时器, 最后生成错误报告.

4 实验分析

本文实验主机的内存大小为 120 GB、CPU 为 Intel(R) Xeon(R) Silver 4114 CPU、主频 2.20 GHz、20 核; CodeQL 工具版本为 v2.9.4; Linux 内核配置为 allyesconfig; GCC 版本为 9.4.0.

4.1 错误检测能力评估

为评估算法发现新错误的能力, 本文基于 Linux 5.15 内核 (long-term support 版本), 将内核分别编译为 x86、arm、sh 和 xtensa 这 4 种架构, 进行实验. 通过对 Linux 5.15 内核的持续检测, 表 1–表 3 为本文新发现的并且已得到内核开发者确认的定时器并发错误信息. 表中一个补丁修复一个或多个错误, 一个 CVE 对应一个或多个补丁.

表 1 新发现的定时器睡眠错误信息

已合并到Linux内核主线的补丁名称	错误数量	功能模块名称	CVE编号	错误详情
NFC: netlink: fix sleep in atomic bug when firmware download timeout	2	nfc	CVE-2022-1975	原子上下文中调用可能睡眠的内存分配函数或可能睡眠的锁函数
NFC: nci: fix sleep in atomic context bugs caused by nci_skb_alloc	3		无	
NFC: hci: fix sleep in atomic context bugs in nfc_hci_hcp_message_tx	5			
staging: rtl8192u: Fix sleep in atomic context bug in dm_fsync_timer_callback	4	staging		
sctp: fix sleep in atomic context bug in timer handlers	14	sctp		
ethernet: rocker: fix sleep in atomic context bug in neigh_timer_handler	1	ethernet		
tty: n_gsm: fix sleep-in-atomic-context bug in gsm_control_send	1	tty		
mwifiex: fix sleep in atomic context bugs caused by dev_coredumpv	4	wireless		
qlcnic: fix sleep-in-atomic-context bugs caused by msleep	2	qlcnic		原子上下文中调用msleep()等睡眠函数
drivers: staging: r8188eu: Fix sleep-in-atomic-context bug in rtw_join_timeout_handler	1	staging		
Revert “char: pcmcia: cm4000_cs: Replace mdelay with usleep_range in set_protocol”	2	char		

表 2 新发现的定时器死锁错误信息

已合并到Linux内核主线的补丁名称	错误数量	功能模块名称	CVE编号	错误详情
drivers: staging: rtl8723bs: Fix deadlock in rtw_surveydone_event_callback()	1	staging	无	低精度定时器死锁
drivers: staging: rtl8192bs: Fix deadlock in rtw_joinbss_event_prehandle()	1			
drivers: staging: rtl8192eu: Fix deadlock in rtw_joinbss_event_prehandle	1			
drivers: staging: rtl8192u: Fix deadlock in ieee80211_beacons_stop()	1			
drivers: staging: rtl8192e: Fix deadlock in rtllib_beacons_stop()	1			
drivers: net: hippi: Fix deadlock in rr_close()	1	hippi		
drivers: tty: serial: Fix deadlock in sa1100_set_termios()	1	tty		
drivers: usb: host: Fix deadlock in oxu_bus_suspend()	1	usb		
can: grcan: grcan_close(): fix deadlock	1	can		
RDMA/irdma: Fix deadlock in irdma_cleanup_cm_core()	1	idrma		
xtensa: iss: clean up per-device locking in network driver	1	xtensa		
arch: xtensa: platforms: Fix deadlock in rs_close()	1			
usb: chipidea: fix deadlock in ci_otg_del_timer	1	usb		高精度定时器死锁

表 3 新发现的僵尸定时器错误信息

已合并到Linux内核主线的补丁名称	错误数量	功能模块名称	CVE编号	错误详情
ax25: Fix NULL pointer dereferences in ax25 timers	10	ax25	CVE-2022-1205	定时器非阻塞删除函数误用
ax25: Fix UAF bugs in ax25 timers	40			
ax25: Fix ax25 session cleanup problems	6			
ax25: fix use-after-free bugs caused by ax25_ds_del_timer	5		无	
net: rose: fix UAF bugs caused by timer handler	35	rose	CVE-2022-2318	
net: rose: fix UAF bug caused by rose_t0timer_expiry	12		无	
scsi: libsas: fix use-after-free bug in smp_execute_task_sg	6	scsi		
watchdog: cpu5wdt.c: Fix use-after-free bug caused by cpu5wdt_trigger	1	watchdog	CVE-2024-38630	
ALSA: sh: aica: reorder cleanup operations to avoid UAF bugs	1	sound	CVE-2024-26654	
media: netup_unidvb: fix use-after-free at del_timer()	1	media	无	
mISDN: fix use-after-free bugs in l1oip timer handlers	5	mISDN		
mISDN: hfcpci: Fix use-after-free bug in hfcpci_softirq	1			定时器阻塞删除函数误用
nfc: replace improper check device_is_registered() in netlink related functions	1	nfc	CVE-2022-1974	
nfc: nfcmrvl: main: reorder destructive operations in nfcmrvl_nci_unregister_dev to avoid bugs	3		CVE-2022-1734	
nfc: pn533: Fix use-after-free bugs caused by pn532_cmd_timeout	7		无	定时器忘记清理
atm: idt77252: fix use-after-free bugs caused by tst_timer	30	atm	CVE-2022-3635	
media: rc: Fix use-after-free bugs caused by ene_tx_irqsim()	3	media	CVE-2023-1118	
drivers: hamradio: 6pack: fix UAF bug caused by mod_timer()	1	6pack	CVE-2022-1198	定时器重新激活
Input: cyttsp4_core - change del_timer_sync() to timer_shutdown_sync()	1	ethernet	CVE-2023-4134	
cxgb4: fix use after free bugs caused by circular dependency problem	9	touchscreen	CVE-2023-4133	
net: usb: lan78xx: reorder cleanup operations to avoid UAF bugs	35	lan78xx	CVE-2023-6039	
NFC: add NCI_UNREG flag to eliminate the race	2	nfc	CVE-2021-4202	
nfc: nci: add flush_workqueue to prevent uaf	2		无	
nfc: pn533: Fix buggy cleanup order	1			
ASoC: rt5645: Fix errorenous cleanup order	25	rt5645		

本文新发现的定时器睡眠错误共有 39 个得到了确认, 主要分布在网络协议与设备驱动程序中.

本文新发现的定时器死锁错误共有 13 个得到了确认. 其中 10 个为 x86 构架下的死锁, 1 个为 arm 架构下死锁, 2 个为 xtensa 架构下的死锁, 主要分布在 staging、usb 等设备驱动程序中.

本文新发现的僵尸定时器错误共有 243 个得到了确认, 主要分布在网络协议与设备驱动中, 其中 1 个为 sh 架构下的僵尸定时器问题. 综上所述, 算法可有效地检测出新的定时器并发错误.

为评估算法能否检测出已知错误, 本文选择了较旧的内核版本 Linux 5.0 (发布于 2019 年 3 月, git commit 1c163f4c7b3f) 进行实验. 检测出的已知的定时器并发错误分布及补丁数量情况如后文表 4 所示. 由表 4 可知, 算法检测出的已知的定时器并发错误主要分布在驱动程序与网络协议中.

为评估本文错误检测能力在业界的水平, 本文基于 Linux 主线内核[36], 通过提取“timer”“sleep”“deadlock”“use-after-free”等关键字、阅读补丁提交说明与代码更改记录的方式, 提取了近 3 年 (2021 年 1 月–2024 年 3 月) 与本文所提 3 类定时器并发错误相关的补丁, 共计 69 个, 其中 49 个是本文提交的, 占比 71%. 具体来说, 定时器睡眠错误相关补丁 13 个, 有 11 个补丁是本文提交的, 占比 84.6%; 定时器死锁错误相关补丁 14 个, 有 13 个补丁是本文提交的, 占比 92.8%; 僵尸定时器错误相关补丁 42 个, 有 25 个是本文提交的, 占比 59.5%. 由此可知, 本文错误检测能力在业界处于领先水平.

4.2 误报与漏报评估

为评估本文算法的误报, 本文基于 Linux 5.15 内核进行实验. 在检测定时器睡眠错误时, 共检测出错误 69 个, 其中误报 18 个, 误报率为 26.1%. 在检测定时器死锁错误时, 共检测出错误 16 个, 其中误报 1 个, 误报率为 5.9%.

在检测因定时器删除函数误用而导致的僵尸定时器错误时, 共检测出错误 183 个, 其中误报 52 个, 误报率为 28.4%. 在检测因定时器重新激活而导致的僵尸定时器错误时, 共检测出可能产生错误的定时器 20 个, 其中误报 7 个, 误报率为 25.9%. 在检测因忘记清理定时器而导致的僵尸定时器错误时, 共检测出可能产生错误的定时器 4 个, 其中误报 1 个, 误报率为 25%.

表 4　检测出的已知的定时器并发错误分布

错误类型	产生错误的位置	Linux内核主线补丁数量
定时器睡眠错误	drivers/net/ethernet	1
	net/ipv6	1
	sound	1
	samples/ftrace	1
	drivers/char	1
定时器死锁错误	drivers/s390	1
僵尸定时器错误	drivers/iommu	1
	drivers/isdn	1
	drivers/atm	2
	driver/bus/mhi	1
	drivers/watchdog	3
	drivers/hte	1
	sound	1
	drivers/hid	1
	drivers/net/wireless	1

本文总结了误报的原因主要有以下 8 点.

① 在检测定时器睡眠错误时, 一些可能产生问题的路径仅管理论可达, 但实际应用场景下却无法通过真实硬件或系统调用进行触发.

② 本文在识别间接函数调用时, 尽管采用了多层指针类型配置与指针所在文件位置信息进行辅助, 但如果各级指针类型相同并且文件位置信息也相同则会产生误报.

③ 内核中存在重名的函数, 尽管本文通过无效边过滤算法可以提高检测的准确率, 但如果两个重名函数位置和调用者函数位置的公共前缀相同, 则无法进一步精确识别, 从而导致误报.

④ 本文路径约束识别算法只能识别函数调用的参数为常量, 并且该常量参数作为被调用函数中 if 或 switch 语句条件判断的情况. 当条件判断的值为函数的返回值、动态赋值的变量等情况时, 路径约束识别算法无法识别造成误报.

⑤ 本文在识别作为结构体成员的定时器时, 是通过定时器名称、定时器父结构体类型与所在模块信息进行识别的. 但内核中存在对结构体的强制类型转换操作, 导致无法准确识别正在操作的定时器, 导致误报.

⑥ 本文使用的指向分析算法是 CodeQL 提供的基于 steensgaard 算法实现的, 该算法虽然效率较高但为流不敏感算法, 会丢失一定精度, 导致识别出的释放的资源与解引用的资源实际为不同的对象, 从而产生误报.

⑦ 本文在检测因定时器重新激活而产生的僵尸定时器错误时, 采用启发式的方法提取设备移除函数, 不够精确, 导致误报产生.

⑧ 本文在检测僵尸定时器错误时, 仅通过锁集分析来判断是否会产生错误, 没有考虑信号量、与错误位置相关的其他层次中的同步机制等, 从而导致误报.

本文总结了漏报的原因主要有以下 3 点.

① 本文采取启发式的方法提取可能睡眠的函数、锁资源和资源清理操作, 提取的可能不够全面, 从而产生漏报.

② 本文算法主要通过锁集分析进行错误检测, 没有考虑条件判断语句对同步关系的影响. 即使两个目标对象的锁集不为空, 但由于关键的条件判断语句并没有用锁进行保护, 同样可能产生错误, 从而产生漏报.

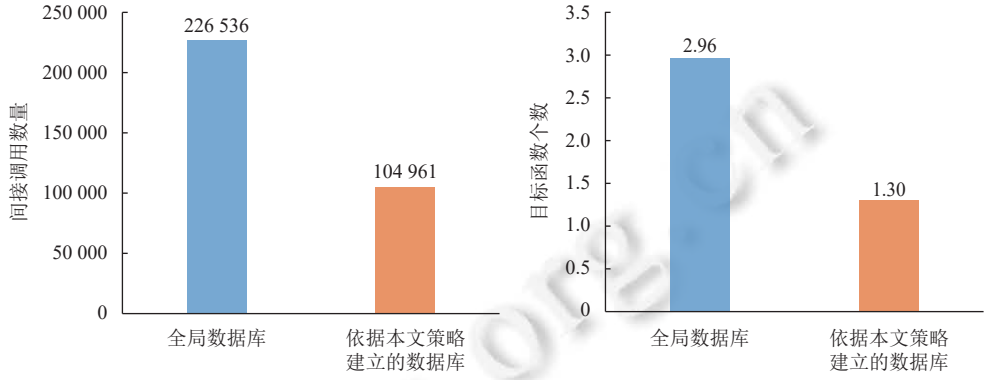
③ 在检测僵尸定时器错误时, 本文未能对所有应该执行定时器删除操作的函数进行精准定位, 如果在这些函

数中未执行定时器删除操作, 则可能导致僵尸定时器错误, 从而产生漏报.

4.3 信息库构建与预处理阶段策略评估

(1) 数据库构建策略有效性评估

为评估数据库构建策略的有效性, 本文分别构建了将所有代码合并在一起的全局数据库和采用数据库构建策略构建的目标数据库. 二者虽然构建的形式不同, 但其中包含的信息是相同的. 实验结果如图 23 所示.



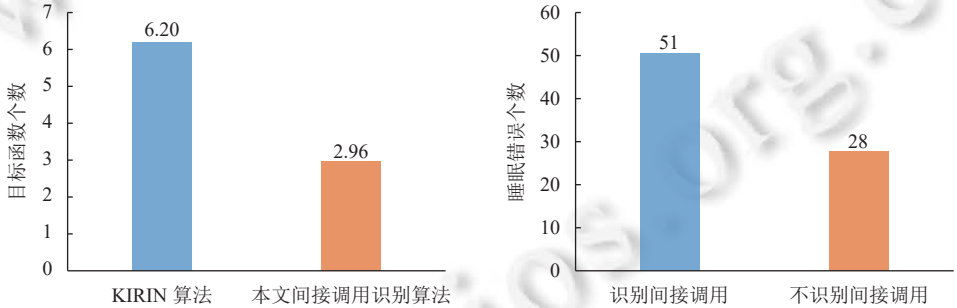
(a) 识别出的间接调用数量 (b) 平均每个间接调用对应的目标函数个数

图 23 数据库构建策略评估结果

基于全局数据库进行间接调用识别时, 共建立间接调用关系 226 536 个, 平均每个间接调用对应的目标函数个数为 2.96 个. 基于构建策略构建的数据库进行间接调用识别时, 共建立间接调用关系 104 961 个, 平均每个间接调用对应的目标函数个数为 1.3 个. 由此可知数据库构建策略可有效地提升间接调用识别准确率.

(2) 间接调用识别算法评估

为评估间接调用识别算法的性能, 本文进行了两组实验. 首先与现有间接调用识别方法进行了对比; 然后评估了算法在检测定时器睡眠错误时的效果. 实验结果如图 24 所示.



(a) 平均每个间接调用对应的目标函数个数 (实验 1) (b) 检测出的定时器睡眠错误个数 (实验 2)

图 24 间接调用识别算法评估结果

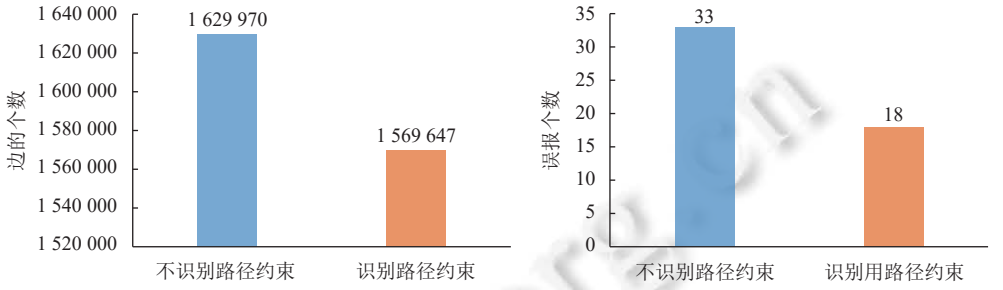
实验 1: 与现有间接调用识别算法进行对比. KIRIN[37]同样采用了基于指针类型匹配和数据流分析的方式进行间接调用识别. 在 allyesconfig 配置下 KIRIN 平均每个间接调用对应的目标函数个数为 6.2, 而本文算法作用在全局数据库下, 平均每个间接函数调用对应的目标函数个数为 2.96. 此外, KIRIN 的作用域仅为内核接口中的函数指针, 而本文算法关注了所有间接函数调用的情况. 本文间接调用识别算法较优越的原因为: 本文算法属于多层指针类型识别算法, 并且通过间接调用所在文件位置信息对间接调用识别进行辅助, 过滤掉了大量无用的目标函数.

实验 2: 基于构建策略构建的数据库, 保持其他条件不变, 分别统计采用间接调用识别算法与不采用该算法两

种情况下定时器睡眠错误的检测效果. 当采用间接调用识别算法时, 共检测出真实定时器睡眠错误 51 个; 当不采用该算法时, 共检测出真实定时器睡眠错误 28 个. 采用间接调用识别算法时比不采用该算法多检测出了 23 个真实错误; 由此可知, 通过本文间接调用识别算法可以更全面地检测出错误.

(3) 路径约束识别算法评估

为评估路径约束识别算法的性能, 本文进行了两组实验, 分别评估算法对构建函数调用图的影响和对检测定时器睡眠错误的影响. 实验结果如图 25 所示.



(a) 函数调用图中边的个数 (实验 1) (b) 检测定时器睡眠错误时产生的误报 (实验 2)

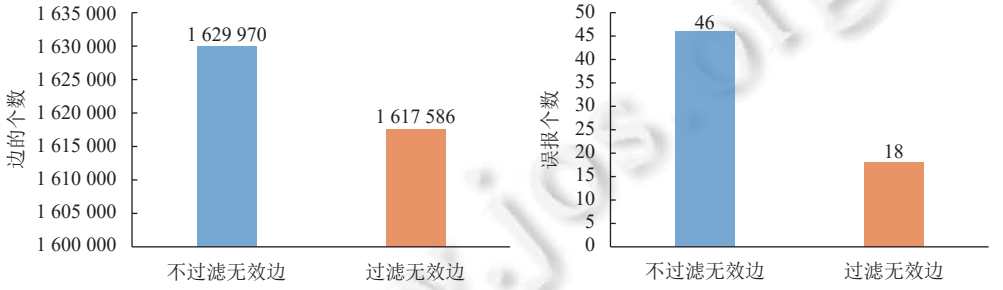
图 25 路径约束识别算法评估结果

实验 1: 基于构建策略构建的目标数据库, 不对间接调用进行识别, 不对无效边过滤, 分别统计采用路径约束算法和不采用路径约束算法时函数调用图的边数. 当不采用路径约束识别算法时, 函数调用图中具有的边 1 629 970 个; 当采用路径约束识别算法时, 函数调用图中具有边 1 569 647 个; 可将函数调用图的准确率提升 3.7%. 由实验结果可知, 算法可有效地提升函数调用图精度.

实验 2: 基于构建策略构建的目标数据库, 保持其他条件不变, 分别统计不采用路径约束算法和采用路径约束算法在检测定时器睡眠错误时的效果. 当采用路径约束算法对定时器睡眠错误进行检测时, 产生的误报为 18 个; 当不采用该算法时, 产生的误报为 33 个; 不采用路径约束算法时比采用该算法多产生了 15 个误报. 由此可知, 算法可有效地降低误报.

(4) 无效边过滤算法评估

为评估无效边过滤算法的性能, 本文进行了两组实验, 分别评估算法对构建函数调用图的影响和对检测定时器睡眠错误的影响. 实验结果如图 26 所示.



(a) 函数调用图中边的个数 (实验 1) (b) 检测定时器睡眠错误时产生的误报 (实验 2)

图 26 无效边过滤算法评估结果

实验 1: 基于本文构建的目标数据库, 不对间接调用和路径约束进行识别, 分别统计采用无效边过滤算法和不采用无效边过滤算法两种情况下函数调用图的边数. 当不采用无效边过滤算法时, 函数调用图中具有边 1 629 970 个; 当采用了无效边过滤算法时, 函数调用图中具有边 1 617 586 个; 共减少了 12 384 个无效边, 将函数调用图的准确率提升了 0.76%. 由实验结果可知, 算法可有效地提升函数调用图的精度.

实验 2: 基于本文构建的目标数据库, 保持其他条件不变, 分别统计采用无效边过滤算法和不采用该算法时检测定时器睡眠错误的情况. 当采用无效边过滤算法时, 产生的误报为 18 个; 当不采用该算法时, 产生的误报为 46 个; 不采用无效边过滤算法时比采用该算法多产生了 28 个误报. 由实验结果可知, 算法可有效地降低误报.

4.4 3 种检测算法与现有工具对比

4.4.1 定时器睡眠错误检测算法与 DSAC 工具对比

DSAC[38]作为原子上下文睡眠错误检测工具, 考虑了在自旋锁 spin_lock() 和中断处理程序中睡眠两种情况, 因此本文选择与其进行对比. 通过收集分析 Linux 内核提交日志中 DSAC 发现的定时器睡眠错误, 并未发现与定时器睡眠错误, 原因为 DSAC 并没有将定时器处理程序作为检测的对象.

本文检测算法相对 DSAC 的优势在于以下两个方面: ① DSAC 是基于所有内核代码进行检测, 而本文在信息库构建阶段应用数据库构建策略避免了对大量无关代码进行分析, 提高了检测效率和准确率. ② 本文和 DSAC 都考虑了间接函数调用和路径约束对错误检测效果的影响. 此外, 本文针对内核中存在函数同名的情况, 使用基于程序局部性原理的无效边过滤算法对无效的调用路径进行了过滤, 提高了函数调用图的精度. 而 DSAC 则没有考虑此类情况.

4.4.2 定时器死锁错误检测算法与 DLOS 工具对比

通过调研, 现阶段通过静态分析的方式检测内核死锁错误的工具仅有 RacerX[9]和 DLOS[14]两种. DLOS 相比 RacerX 在检测效果方面具有较大提高. 因此, 本文选择与其进行对比. DLOS 首先通过基于摘要的锁使用分析提取每个目标路径上锁约束; 然后通过可达性分析检测具有循环请求的锁关系; 进一步通过路径可达性分析与路径可并发性检测来过滤误报.

通过收集分析 Linux 内核提交日志中 DLOS 发现的死锁错误, 并未发现与定时器死锁错误, 原因为 DLOS 主要用于检测 ABBA 型具有循环等待关系的死锁错误, 而定时器死锁错误是由具有同步功能的阻塞等待函数造成的. DLOS 检测方法并不完全适用于检测定时器死锁错误.

本文相比 DLOS 的优势体现在以下几个方面: ① DLOS 没有考虑到间接函数调用对检测结果带来的影响, 而本文则进行了考虑. ② 在提取锁约束时需遍历各条路径, 为避免路径爆炸的问题, DLOS 通过自底向上的数据流分析随机选取路径进行分析, 会产生一定漏报. 本文则通过限制遍历的层数来避免路径爆炸问题, 当函数遍历层数达到 2 层以上时, 新增锁交集个数始终为 0. 因此本文限制函数遍历的层数为 2, 从而更全面地检测错误.

4.4.3 僵尸定时器错误检测算法与 UACatcher 工具对比

UACatcher[15]是检测内核因设备移除而产生的并发释放后使用错误的工具. 本文部分僵尸定时器错误同样是与设备移除操作相关的并发释放后使用错误, 因此选择与该工具进行对比.

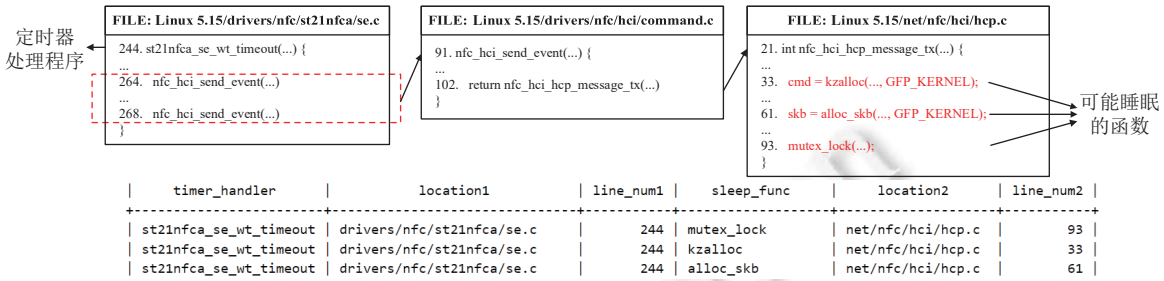
通过收集分析 Linux 内核提交日志中 UACatcher 发现的错误, 仅发现 1 个与僵尸定时器错误相关的补丁, 因为 UACatcher 检测的为设备清理线程与系统调用线程之间的并发错误, 并未针对定时器相关线程进行建模. 本文算法与 UACatcher 相比, 优势主要体现在以下 3 个方面: ① 在信息库构建阶段, 本文是以功能模块为粒度建立数据库, 而 UACatcher 是以层次为粒度建立数据库进行检测. 在检测因定时器重新激活产生的僵尸定时器错误时, 需要判断能否从边界函数调用到定时器激活函数, 以层次为粒度的数据库由于只包含特定层次的代码信息, 会遗漏很多可以激活定时器的边界函数, 从而导致漏报. ② 在预处理阶段, 本文考虑了调用路径上存在间接函数调用的情况, 而 UACatcher 则未对间接函数调用进行处理. 因此, 本文算法可检测出更多错误. ③ 在错误检测阶段, 本文算法首先确定可能成为僵尸状态的定时器, 缩小了检测范围, 然后再应用锁集分析等算法进行检测. 而 UACatcher 则是从资源释放操作和解引用操作出发, 应用锁集分析同时结合条件判断进行检测, 并没有考虑 del_timer()、del_timer_sync() 等同步操作语句, 也没有考虑定时器与工作队列等异步机制循环调用的情况. 此外, 对于引用计数管理的资源, UACatcher 默认其不会产生错误, 从而导致其相比本文算法会有一定的漏报.

4.5 错误详情

本节展示了系统在 Linux 5.15 内核中检测到的定时器并发错误, 所有错误均得到了内核开发人员的确认.

4.5.1 定时器睡眠错误详情

图 27 中上侧为定时器睡眠错误示例, 下侧为检测报告. 在 se.c 文件中定时器处理程序 st21nfca_se_wt_timeout() 可以调用到 hcp.c 文件中第 33、61 和 93 行的可能睡眠的函数 kzalloc(…, GFP_KERNEL)、alloc_skb(…, GFP_KERNEL) 和 mutex_lock(), 从而产生定时器死锁错误.

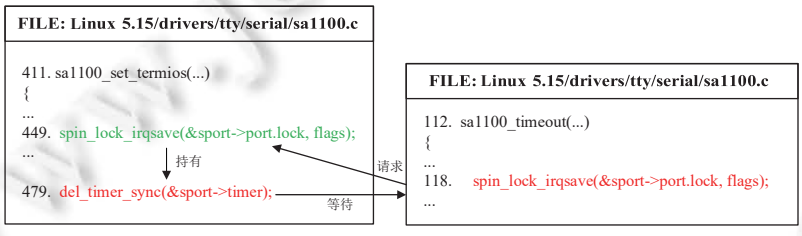


timer_handler	location1	line_num1	sleep_func	location2	line_num2
st21nfca_se_wt_timeout	drivers/nfc/st21nfca/se.c	244	mutex_lock	net/nfc/hci/hcp.c	93
st21nfca_se_wt_timeout	drivers/nfc/st21nfca/se.c	244	kzalloc	net/nfc/hci/hcp.c	33
st21nfca_se_wt_timeout	drivers/nfc/st21nfca/se.c	244	alloc_skb	net/nfc/hci/hcp.c	61

图 27 nfc 协议定时器睡眠错误示例及检测报告

4.5.2 定时器死锁错误详情

图 28 中上侧为定时器死锁错误示例, 下侧为检测报告 (需将内核编译成 arm 架构进行检测). sa1100.c 文件中第 479 行定时器阻塞删除函数 del_timer_sync() 持有锁 sport->port.lock 并等待第 112 行的定时器处理程序 sa1100_timeout() 结束, 而定时器处理程序也需获取同样的锁, 从而产生死锁.

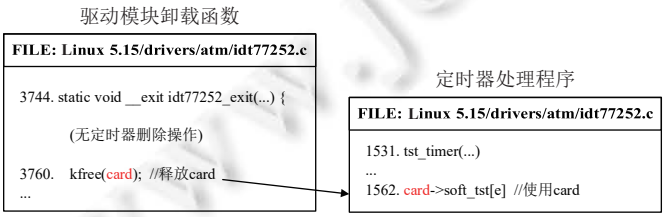


del_name	del_location	line_num1	handler_name	handler_location	line_num2	lock_name
del_timer_sync	drivers/tty/serial/sa1100.c	479	sa1100_timeout	drivers/tty/serial/sa1100.c	112	sport->port.lock

图 28 tty 驱动定时器死锁错误示例及检测报告

4.5.3 僵尸定时器错误详情

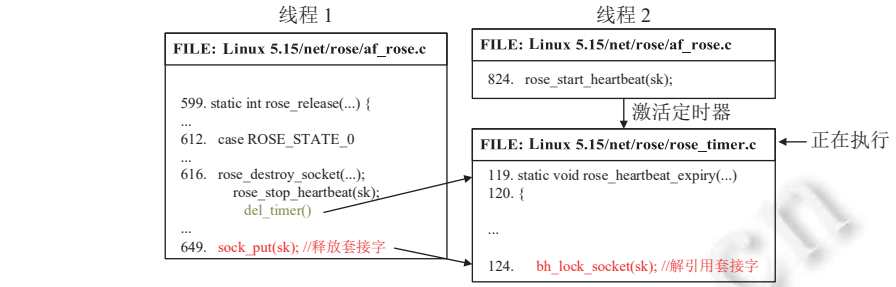
图 29 中上侧为忘记清理定时器错误示例, 下侧为错误检测报告. 在驱动模块卸载函数 idt77252_exit() 中没有对定时器 tst_timer (所在结构体为 idt77252_dev) 进行删除, 导致资源 card 在释放后, 又在定时器处理程序中再次使用, 产生释放后使用错误.



handler_name	file1	line1	cleanup_func_name	file2	line2	timer_name	parent_struct	resource
tst_timer	drivers/atm/idt77252.c	1531	idt77252_exit	drivers/atm/idt77252.c	3744	tst_timer	idt77252_dev	card

图 29 忘记清理定时器示例及检测报告

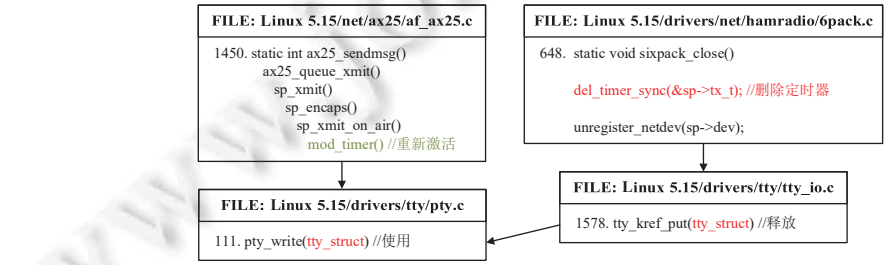
图 30 中上侧为定时器删除函数误用示例, 下侧为错误检测报告. rose_release() 函数中 del_timer() 无法删除正在执行的定时器处理程序, 使定时器 sk_timer (所在结构体为 sock) 变为僵尸状态, 导致套接字 sk 释放后又在定时器处理程序中再次使用, 产生释放后使用错误.



handler_name	file1	line1	cleanup_func_name	file2	line2	timer_name	parent_struct	resource
rose_heartbeat_expiry	net/rose/rose_timer.c	119	rose_release	net/rose/af_rose.c	599	sk_timer	sock	sk

图 30 定时器删除函数误用示例及检测报告

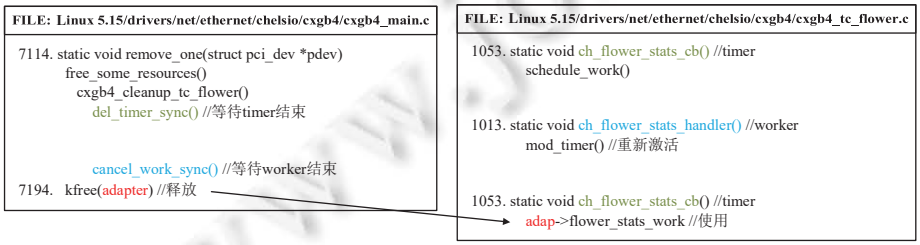
图 31 上侧为从系统调用重新激活定时器的示例, 下侧为错误检测报告. 在设备移除函数 sixpack_close() 中对定时器 tx_t (所在结构体为 sixpack) 进行删除之后, 又可以通过 send 系统调用执行内核接口函数 ax25_sendmsg() 重新激活定时器, 使定时器处理程序与设备移除函数产生竞态条件, 导致资源 tty_struct 产生释放后使用错误.



interface	file1	line1	cleanup_func_name	file2	line2	timer_name	parent_struct	resource
ax25_sendmsg	net/ax25/af_ax25.c	1450	sixpack_close	drivers/net/hamradio/6pack.c	648	tx_t	sixpack	tty_struct

图 31 从系统调用重新激活定时器示例及检测报告

图 32 中上侧为从工作队列中重新激活定时器的示例, 下侧为错误检测报告. 在 cxgb4_tc_flower.c 文件中第 1053 行定时器处理程序 ch_flower_stats_cb 与 1013 行工作者线程 ch_flower_stats_handler 存在循环调用关系. 在设备移除函数 remove_one() 中对定时器 flower_stats_timer (所在结构体为 adapter) 执行删除操作后, 定时器又可从工作者线程重新激活后, 变为僵尸状态, 使资源 adapter 产生释放后使用错误.



file1	timer_handler	line1	worker_callback	line2	cleanup_func_name	file2	line3	timer_name	parent_struct	resource
cxgb4_tc_flower.c	ch_flower_stats_cb	1053	ch_flower_stats_handler	1013	remove_one	cxgb4_main.c	7114	flower_stats_timer	adapter	adapter

图 32 从工作队列中重新激活定时器示例及检测报告

4.6 错误修复方法

为修复定时器睡眠错误可采用以下两种方法: ① 将睡眠操作放入工作队列、延迟队列等进程上下文中. ② 将可睡眠操作替换为不可睡眠操作. 图 33 分别为两种修复方法的示例 (详见内核提交日志: ① NFC: hci: fix sleep in atomic context bugs in nfc_hci_hcp_message_tx; ② qlcnic: fix sleep-in-atomic-context bugs caused by msleep).

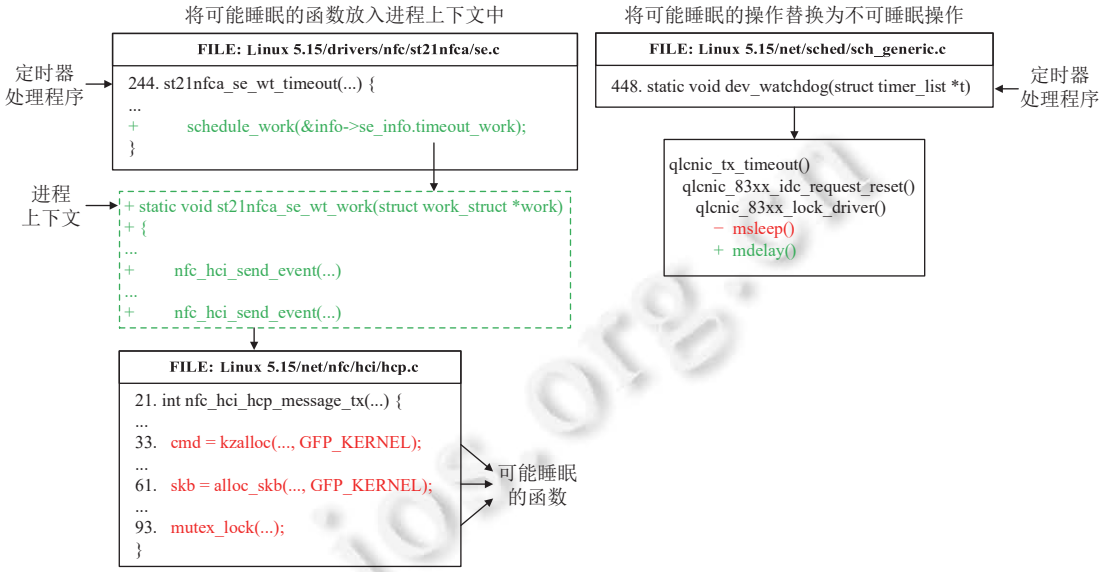


图 33 修复定时器睡眠错误的两种方法

为修复定时器死锁错误可采用破坏“持有并等待”条件来避免死锁, 即确保不发生错误的条件下, 通过减少阻塞删除函数持有的锁资源避免死锁. 示例如图 34 所示 (详见内核提交日志: drivers: tty: serial: Fix deadlock in sa1100_set_termios()).

僵尸定时器错误分为 3 种. 为修复因忘记清理定时器而产生的错误时, 可采用添加定时器删除函数进行修复. 示例如图 35 所示 (详见内核提交日志: atm: idt77252: fix use-after-free bugs caused by tst_timer).

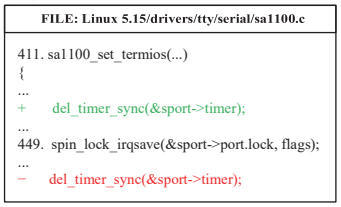


图 34 定时器死锁错误修复示例

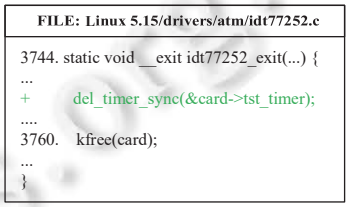


图 35 忘记清理定时器错误修复

为修复定时器删除函数误用而产生的错误时, 可采用将定时器非阻塞删除函数变为阻塞删除函数、去除误用的定时器状态判断语句、为产生错误的资源添加引用计数的方法进行修复. 示例分别如图 36 所示 (详见内核提交日志: ① net: rose: fix UAF bug caused by rose_t0timer_expiry; ② mISDN: hfcpci: Fix use-after-free bug in hfcpci_softirq; ③ net: rose: fix UAF bugs caused by timer handler).

为修复因定时器重新激活而产生的错误时, 可采用调整同步操作语句顺序、添加新的同步操作、使用 timer_shutdown[_sync]() 系列函数进行修复. 示例如图 37 所示 (详见内核提交日志: ① drivers: hamradio: 6pack: fix UAF bug caused by mod_timer(); ② nfc: nci: add flush_workqueue to prevent uaf; ③ cxgb4: fix use after free bugs caused by circular dependency problem).

FILE: Linux 5.15/net/rose/rose_route.c

```
226. rose_remove_neigh(struct rose_neigh *rose_neigh)
227. {
...
−       rose_stop_ftimer(rose_neigh);
−       rose_stop_t0timer(rose_neigh);
+       del_timer_sync(&rose_neigh->ftimer);
+       del_timer_sync(&rose_neigh->t0timer);
...
240.    kfree(rose_neigh);
```

(a) 将非阻塞删除函数变为阻塞删除函数

FILE: Linux 5.15/drivers/isdn/hardware/mISDN/hfcpci.c

```
2346.  HFC_cleanup(void)
{
−    if (timer_pending(&hfc_tl))
−        del_timer_sync(&hfc_tl);
+    del_timer_sync(&hfc_tl);
```

(b) 去除误用的定时器状态判断语句

FILE: Linux 5.15/net/rose/timer.c

```
32. void rose_start_heartbeat(struct sock *sk)
{
−    del_timer(&sk->sk_timer);
+    sk_stop_timer(sk, &sk->sk_timer);
...
−    add_timer(&sk->sk_timer);
+    sk_reset_timer(sk, &sk->sk_timer, sk->sk_timer.expires);
}
```

(c) 为产生错误的资源添加引用计数

图 36　定时器删除函数误用修复

FILE: Linux 5.15/drivers/net/hamradio/6pack.c

```
648. static void sixpack_close(...)
{
...
+    unregister_netdev(sp->dev);
...
     del_timer_sync(&sp->tx_t);
     del_timer_sync(&sp->resync_t);
...
−    unregister_netdev(sp->dev);
```

(a) 调整同步语句顺序进行修复

FILE: Linux 5.15/drivers/nfc/hci/command.c

```
553. static int nci_close_device(struct nci_dev *ndev)
{
...
+        /* Need to flush the cmd wq in case
+         * there is a queued/running cmd_work
+         */
+        flush_workqueue(ndev->cmd_wq);
         del_timer_sync(&ndev->cmd_timer);
         del_timer_sync(&ndev->data_timer);
```

(b) 添加新的同步操作进行修复

FILE: Linux 5.15/drivers/net/ethernet/chelsio/cxgb4/cxgb4_tc_flower.c

```
1132. void cxgb4_cleanup_tc_flower(struct adapter *adap)
{
...
      if (adap->flower_stats_timer.function)
−         del_timer_sync(&adap->flower_stats_timer);
+         timer_shutdown_sync(&adap->flower_stats_timer);
```

(c) 使用 timer_shutdown[_sync]() 系列函数进行修复

图 37　定时器重新激活错误修复

综上所述, 为预防定时器并发错误可以采用以下方式: 首先, 定时器处理程序运行在原子上下文中, 不能调用可能睡眠的函数. 其次, 在使用定时器阻塞删除函数时要注意其持有的锁不能在定时器处理程序中出现. 最后, 使用定时器时要注意使用引用计数、锁、条件判断等同步机制与其他线程进行同步, 避免出现并发错误.

5 讨　论

本文检测算法的可扩展性. 本文提出的错误检测算法具有较好的可扩展性, 可以用于检测工作队列、延迟队列、tasklet 等延迟机制并发错误. 对于工作队列或延迟队列来说, 其回调函数运行在进程上下文中, 不存在原子上下文睡眠问题; 但二者同样存在阻塞删除函数 (如: cancel_work_sync()、cancel_delayed_work_sync() 等), 如果阻塞删除函数持有的锁也是其回调函数中要获取的锁则会产生死锁; 如果工作队列或延迟队列忘记清理或误用删除操作, 则会变为僵尸状态, 从而可能导致僵尸工作队列或僵尸延迟队列问题. 对于 tasklet 来说, 其特性与定时器类似, 其回调函数运行在原子上下文中; 存在阻塞删除函数 (如: tasklet_kill() 等); 如果 tasklet 忘记清理或误用删除操作则会变为僵尸状态, 因此, 本文 3 种检测算法均适用.

本文检测算法的局限性. 定时器并发错误的种类不仅为本文总结的 3 类, 也包括其他类型, 例如: 没有变为僵尸状态的定时器处理程序与其他线程竞争产生的错误、由定时器并发错误导致的内存泄漏、空指针解引用等问题. 由于定时器并发错误的种类过多, 本文未能对所有的定时器并发错误类型进行全面建模与检测是本文的一个局限性.

6 总　结

定时器是一种软中断, 也是一种异步机制, 是竞态条件的重要来源. 本文总结了 3 种类型的定时器并发错误: 定时器睡眠错误、定时器死锁错误和僵尸定时器错误. 围绕如何进行定时器并发错误检测, 本文检测流程分为以下 3 个部分. 在信息库构建阶段, 通过指针分析的方式, 提取所有与定时器相关的功能模块. 避免对大量无关的代码进行分析, 提高分析效率. 在信息预处理阶段, 分别构建了函数调用图、过程间控制流图、定时器与定时器处理程序之间的关系为后续错误检测算法提供了基础. 在错误检测阶段, 依据 3 种不同的并发错误类型, 综合应用模式

匹配、控制流分析、指向分析和锁集分析等方法提出了 3 种不同的错误检测算法. 应用以上算法本文共检测出定时器并发错误 328 个, 截至目前, 共有 49 个补丁被接收并合并到 Linux 主线, 295 个错误得到了修复, 申请了 14 个 CVE 编号, 说明了本文所提出方法的有效性.

References:

[1] CVE-2016-8655 Linux af_packet.c race condition (local root). 2022. https://seclists.org/oss-sec/2016/q4/607

[2] CVE-2022-1734 Detail. 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-1734

[3] CVE-2022-2318 Detail. 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-2318

[4] Google/Syzkaller. GitHub. 2021. https://github.com/google/syzkaller/tree/master

[5] Bai JJ, Wang YP, Lawall J, Hu SM. DSAC: Effective static analysis of sleep-in-atomic-context bugs in kernel modules. In: Proc. of the 2018 USENIX Annual Technical Conf. Boston: USENIX Association, 2018. 587–599.

[6] Møller A, Schwartzbach MI. Static program analysis. 2025. https://cs.au.dk/~amoeller/spa/spa.pdf#:~:text=Static%20program%20analysis

[7] Bai JJ, Lawall J, Chen QL, Hu SM. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In: Proc. of the 2019 USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 255–268.

[8] Voung JW, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Dubrovnik: ACM, 2007. 205–214. [doi: 10.1145/1287624.1287654]

[9] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. ACM SIGOPS Operating Systems Review, 2003, 37(5): 237–252. [doi: 10.1145/1165389.945468]

[10] Cai YD, Yao PS, Zhang C. Canary: Practical static detection of inter-thread value-flow bugs. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. New York: ACM, 2021. 1126–1140. [doi: 10.1145/3453483.3454099]

[11] Liu BZ, Liu PM, Li YZ, *et al*. When threads meet events: Efficient and precise static race detection with origins. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. New York: ACM, 2021. 725–739. [doi: 10.1145/3453483.3454073]

[12] Kahlon V, Yang Y, Sankaranarayanan S, Gupta A. Fast and accurate static data-race detection for concurrent programs. In: Proc. of the 19th Int'l Conf. on Computer Aided Verification. Berlin: Springer, 2007. 226–239.

[13] Machiry A, Spensky C, Corina J, Stephens N, Kruegel C, Vigna G. Dr. Checker: A soundy analysis for Linux kernel drivers. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 1007–1024.

[14] Bai JJ, Li T, Hu SM. DLOS: Effective static detection of deadlocks in OS kernels. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 367–382.

[15] Ma L, Zhou DM, Wu HJ, Zhou YJ, Chang R, Xiong H. When top-down meets bottom-up: Detecting and exploiting use-after-cleanup bugs in Linux kernel. In: Proc. of the 2023 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2023. 2138–2154. [doi: 10.1109/SP46215.2023.10179356]

[16] The kernel concurrency sanitizer (KCSAN)—The Linux kernel documentation. 2022. https://docs.kernel.org/dev-tools/kcsan.html

[17] Kernel thread sanitizer (KTSAN). 2022. https://github.com/google/kernel-sanitizers/blob/master/KTSAN.md

[18] Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In: Proc. of the 26th USENIX Conf. on Security Symp. Vancouver: USENIX Association, 2017. 167–182.

[19] Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: IEEE, 2017. 50–59. [doi: 10.1109/ASE.2017.8115618]

[20] Jeong DR, Lee B, Shin I, Kwon Y. SegFuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In: Proc. of the 2023 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2023. 2104–2121. [doi: 10.1109/SP46215.2023.10179398]

[21] Fonseca P, Rodrigues R, Brandenburg BB. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 415–431.

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

[23] Xu M, Kashyap S, Zhao HQ, Kim T. Krace: Data race fuzzing for kernel file systems. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1643–1660. [doi: 10.1109/SP40000.2020.00078]

[24] Kim K, Jeong DR, Kim CH, Jang Y, Shin I, Lee B. HFL: Hybrid fuzzing on the Linux kernel. In: Proc. of the 27th Annual Network and

Distributed System Security Symp. San Diego: The Internet Societ, 2020. 1–17.
[25] S2E: The selective symbolic execution platform—S2E 2.0 documentation. 2022. http://s2e.systems/docs/
[26] Ryan G, Shah A, She DD, Jana S. Precise detection of kernel data races with probabilistic lockset analysis. In: Proc. of the 2023 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2023. 2086–2103. [doi: 10.1109/SP46215.2023.10179366]
[27] Shi JJ, Ji WX, Shi F. Recent progress of concurrency bug detection in operating system kernels. Ruan Jian Xue Bao/Journal of Software, 2021, 32(7): 2016–2038 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6265.htm [doi: 10.13328/j.cnki.jos.006265]
[28] Deligiannis P, Donaldson AF, Rakamarić Z. Fast and precise symbolic analysis of concurrency bugs in device drivers (T). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 166–177. [doi: 10.1109/ASE.2015.30]
[29] Souri A, Rahmani AM, Navimipour NJ, Rezaei R. A symbolic model checking approach in formal verification of distributed systems. Human-centric Computing and Information Sciences, 2019, 9(1): 4. [doi: 10.1186/s13673-019-0165-x]
[30] Kim M, Hong S, Hong C, Kim T. Model-based kernel testing for concurrency bugs through counter example replay. Electronic Notes in Theoretical Computer Science, 2009, 253(2): 21–36. [doi: 10.1016/j.entcs.2009.09.049]
[31] Guo YD, Yin Q, Dong WY. Linux Principles and Structures. Xi'an: Xidian University Press, 2012. 118–126 (in Chinese).
[32] Corbet J, Rubini A, Kroah-Hartman G. Linux Device Drivers. 3rd ed., Sebastopol: O'Reilly Media, Inc., 2005. 197–200.
[33] About CodeQL—CodeQL. 2022. https://codeql.github.com/docs/codeql-overview/about-codeql/
[34] Chen HB, Xia YB. Modern Operating Systems: Principle and Implementation. Beijing: China Machine Press, 2020. 239–240 (in Chinese).
[35] Class GuardCondition. GitHub. 2022. https://codeql.github.com/docs/codeql-language-guides/using-the-guards-library-in-cpp/
[36] Linux. GitHub. 2023. https://github.com/torvalds/linux
[37] Zhang T, Shen WB, Lee D, Jung C, Azab AM, Wang RW. PeX: A permission check analysis framework for Linux kernel. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 1205–1220.
[38] Bai JJ, Lawall J, Hu SM. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. ACM Trans. on Computer Systems (TOCS), 2018, 36(4): 10. [doi: 10.1145/3381990]

附中文参考文献:

[27] 石剑君, 计卫星, 石峰. 操作系统内核并发错误检测研究进展. 软件学报, 2021, 32(7): 2016–2038. http://www.jos.org.cn/1000-9825/6265.htm [doi: 10.13328/j.cnki.jos.006265]
[31] 郭玉东, 尹青, 董卫宇. Linux 原理与结构. 西安: 西安电子科技大学出版社, 2012. 118–126.
[34] 陈海波, 夏虞斌. 现代操作系统: 原理与实现. 北京: 机械工业出版社, 2020. 239–240.

周多明(1993－), 男, 硕士, 主要研究领域为操作系统安全.

周亚金(1982－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为操作系统安全, 嵌入式系统安全, 软件安全, 二进制分析, 区块链安全.

马麟(1998－), 男, 博士生, 主要研究领域为操作系统安全.