学校代码：10284
分 类 号：TP316.1
密　　级：公开
U D C：004.45
学　　号：DZ1833026

南京大学

博 士 学 位 论 文

论 文 题 目　基于 eBPF 技术的内核漏洞实时防御研究

作 者 姓 名　王子成

专 业 名 称　计算机科学与技术

研 究 方 向　操作系统内核安全

导 师 姓 名　曾庆凯 教授

2024 年 3 月 25 日

答辩委员会主席 王箭 教授

评　　阅　　人 吴礼发 教授

茅兵 教授

陆桑璐 教授

钱柱中 教授

论文答辩日期　2024 年 4 月 29 日

研究生签名:

导师签名:

Research on Real-time Defense Against Kernel Vulnerabilities Based on eBPF

by

Zicheng Wang

Supervised by

Qingkai Zeng, Prof.

A dissertation submitted to

the graduate school of Nanjing University

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Technology

Department of Computer Science and Engineering

Nanjing University

March 25, 2024

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于 eBPF 技术的内核漏洞实时防御研究

计算机科学与技术 专业 2018 级博士生姓名：王子成

指导教师（姓名、职称）：曾庆凯 教授

摘　要

在数字化时代背景下，操作系统已成为支撑现代社会的重要基础设施。内核是操作系统的核心组件，运行在系统的最高特权层，即使微小的内核漏洞也会将安全威胁放大到整个系统。然而当前主流内核漏洞频发，且完整的修复时间超过 66 天，期间攻击者能利用未修复漏洞攻陷整个系统。现有的内核安全缓解和隔离技术虽然做出了应对，但仍存在局限性。一方面，缓解技术聚焦提升漏洞攻击利用难度，但现有方案复杂性不足易被攻击者绕过；另一方面，隔离技术受制于复杂的软硬件机制，难以实现系统内的实时部署。

为提升对已知内核漏洞的实时防御能力，新增的内核防御方案应满足以下三个核心需求：(1) 灵活性强的安全策略，以应对各种成因复杂的内核漏洞；(2) 能够实时集成至内核中，无需重新编译或重启系统；(3) 保证载入策略的安全性和高效性。eBPF 技术具备强大的安全策略表达和实时部署能力，支持将用户定义的程序动态载入内核沙箱执行，为实现内核漏洞实时防御提供了基础。因此本文改进了 eBPF 技术以进一步发掘其潜力，研究在漏洞利用的不同阶段对内核漏洞进行实时防御，主要的研究贡献及创新点如下：

1. **ERA: 基于 eBPF 的内核堆漏洞缓解框架**。本文率先将 eBPF 技术引入系统安全领域，提出了针对内核堆漏洞的实时防御框架 ERA。该框架旨在漏洞利用阶段提升攻击难度，降低内核安全风险。内核堆漏洞由于攻击利用稳定性较高，是目前主要的攻击目标，而内核现阶段采用的缓解机制均能被稳定绕过。为此 ERA 对 eBPF 生态系统进行改造，成功实现了动态内核安全堆分配器功能。攻击者无法操纵被保护的数据对象构造攻击布局，有效增强了系统的安全性。本文使用真实世界漏洞对 ERA 进行评估，实验结果显示 ERA 能有效防御常见类型内核堆漏洞，压力测试情况下仅对系统造成 1% 的性能开销和

微不足道的内存开销。

2. **PET: 基于 eBPF 的防内核漏洞触发框架**。本文提出了在漏洞利用前阶段防止内核漏洞触发的 PET 框架，与 ERA 框架相比，PET 实现了对更广泛类型内核漏洞的实时防御。在无任何修复补丁参考的情况下，PET 能对整数溢出、越界访问、释放后使用、未初始化访问以及数据竞争多种类型内核漏洞实现实时防御。通过将消毒器漏洞报告作为输入，PET 构建了可以通过 eBPF 程序运行时检查的触发条件。如果触发条件得到满足，PET 将采取一系列行动防止漏洞触发并将内核恢复到平稳工作状态。在实验中，本文展示了 PET 针对上述五种最常见消毒器报告漏洞的有效性。PET 能够同时防御多个内核漏洞，仅造成 3% 以内轻量级的性能开销，并且在内核漏洞触发被阻止后，系统还能稳定运行 3 个月以上。此外，PET 支持漏洞类型无关的运行时机制，有潜力实时防御更广泛类型的内核漏洞。
3. **O2C: 基于 eBPF 和机器学习审计的实时内核分隔探索**。本文提出了实时内核分隔框架 O2C，它突破了 ERA 和 PET 对内核漏洞利用、触发方式的限制，采用实时部署的分隔化技术隔离包含漏洞的内核组件。O2C 探索了内核实时分隔化的可行性，它有效避免了现有分隔化技术部署时会中断系统服务的缺点，在漏洞利用后阶段防止不可信组件攻陷整个系统。本文揭示了实时分隔化的主要挑战在于解决状态切换风险，并为此设计并实现了 O2C，一种审计状态切换风险的探索性分隔化解决方案。它在 eBPF 生态系统的协助下实现了两项技术创新：将机器学习模型嵌入内核来提供智能化安全保障和基于动态插桩的分隔化。实验评估显示 O2C 能有效地将安全威胁限制在分隔区内，决策树模型适合用于协助 O2C 审计状态切换风险。隔离超过 255,000 行代码仅对系统造成了小于 4% 的性能开销，具备出色的可扩展性。

综上所述，本研究指出了内核漏洞修复窗口缺乏实时防御技术的核心问题，并敏锐地识别出 eBPF 技术能够为内核实时防御提供支持。因此，通过对 eBPF 生态系统的改进，本文分别针对内核漏洞利用的中、前、后阶段，提出了可靠、高效的实时防御方案。上述实时防御方案不仅在策略层面实现了高实用性的内核漏洞实时防御，而且在机制层面上展现了 eBPF 技术在安全领域应用的全新思路和实践。

关键词： 内核；eBPF；漏洞；系统安全；操作系统；防御

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research on Real-time Defense Against Kernel Vulnerabilities Based on eBPF

SPECIALIZATION: Computer Science and Technology

POSTGRADUATE: Zicheng Wang

MENTOR: Qingkai Zeng, Prof.

ABSTRACT

In the context of the digital age, the operating system has become a crucial infrastructure supporting modern society. The kernel, as the core component of an operating system, operates at the highest privilege level. Even minor kernel vulnerabilities can magnify security threats across the entire system. However, vulnerabilities in mainstream kernels occur frequently, and the comprehensive fixing process takes longer than 66 days, during which attackers can exploit unpatched vulnerabilities to compromise the entire system. Although Existing kernel security mitigation and isolation techniques make responses, still have limitations. On the one hand, mitigation techniques focus on increasing the difficulty of exploitation rather than fundamentally preventing the triggering of vulnerabilities, they can easily be bypassed by attackers due to the insufficient complexity; On the other hand, isolation techniques rely heavily on complex software or hardware mechanisms, making real-time deployment challenging.

To enhance the real-time defense capabilities against kernel vulnerabilities, the newly developed kernel defense schemes should satisfy the following three core requirements: (1) Flexible security policies to address various kernel vulnerabilities with complex root causes; (2) The ability to be integrated into the kernel at runtime without recompilation or rebooting; (3) Ensuring the security and efficiency of the policies. eBPF possesses powerful capabilities in expressing security policies and real-time deployment, that supports running user-defined programs in a kernel space sandbox. This provides foundational support for the real-time defense against kernel vulnerabilities. In this dissertation, we further modify eBPF and explore its potential. We research the

real-time defense against kernel vulnerabilities at various phases of exploitation. The main research contributions and innovations are as follows:

1. **ERA: An eBPF-based Kernel Heap Vulnerability Mitigation Framework.** This research pioneers the introduction of eBPF into the field of system security. It develops the ERA framework, aimed at real-time defense against kernel heap vulnerabilities. By strategically increasing the complexity of attacks during the exploitation phase, the framework significantly mitigates risks to kernel security. Given their high exploit stability, kernel heap vulnerabilities are currently a major target, but existing mitigation mechanisms can be reliably bypassed. ERA modifies the eBPF ecosystem to support a dynamic kernel secure heap allocator. Attackers cannot manufacture the attacking heap layout with protected objects, and the system security is significantly enhanced. We use real-world vulnerabilities to evaluate ERA, the experiment results show its effectiveness in defending against common kernel heap vulnerabilities, with a lightweight 1% performance overhead and negligible memory overhead.
2. **PET: An eBPF-based Framework to Prevent Kernel Vulnerabilities From Being Triggered.** This dissertation introduces the PET, a framework that proactively prevents the triggering of kernel vulnerabilities prior the exploitation. Building upon the foundation established by the ERA framework, PET extends real-time defense capabilities to encompass a wider range of kernel vulnerabilities. It effectively defends against a spectrum of kernel vulnerabilities—including integer overflows, out-of-bounds accesses, use-after-free, uninitialized access, and race conditions—without any reference to patches. PET takes sanitizer bug reports as input, and constructs eBPF programs that monitor the vulnerability triggering condition at runtime. If the triggering condition is met, PET will take actions to prevent the vulnerability from being triggered, and recover the kernel to maintain stability. In the experiments, PET is effective for the vulnerabilities reported by the state-of-the-art kernel sanitizers. PET is lightweight with a performance overhead of less than 3%, it can defend against multiple vulnerabilities simultaneously, and keep functioning for 3 months after vulnerabilities are triggered. Moreover, PET supports a vulnerability-agnostic runtime check mechanism, potentially enabling defense against more types of kernel

vulnerabilities.

3. **O2C: An Exploration of Real-time Compartmentalization Based on eBPF and Machine Learning Auditing.** This dissertation introduces the O2C, a real-time compartmentalization framework, which surpasses the limitations of the ERA and PET frameworks regarding kernel vulnerability exploitation and triggering methods. It isolates kernel components that contain vulnerabilities. O2C explores the feasibility of real-time kernel compartmentalization, effectively avoiding the drawbacks of existing compartmentalization that interrupt system services, preventing vulnerabilities in untrusted components from compromising the entire system in the post-exploitation phase. In this research, we reveal that the key challenge of real-time compartmentalization is solving the transition hazard, and we design and implement the O2C, an exploratory solution for auditing the transition hazard. O2C achieves two technique innovations with the assistance of eBPF: embedding the machine learning model into the kernel for intelligent security guarantee, and dynamic instrumentation-based compartmentalization. Experimental evaluations show that O2C effectively confines security threats in the compartment, and the decision tree model is suitable for assisting O2C in auditing state transition hazards. Even when isolating over 255,000 lines of code, O2C incurs less than a 4% performance overhead on the system, demonstrating excellent scalability.

In summary, this dissertation highlights the critical issue of the lack of real-time defense in the kernel vulnerability fixing window, and keenly identifies the capability of eBPF to support real-time kernel defense. Consequently, through enhancements to the eBPF ecosystem, this dissertation proposes reliable and efficient real-time defense strategies for the mid-, pre-, and post-exploitation phases of kernel vulnerabilities. These strategies not only achieve practical security protection against kernel vulnerabilities at the policy level, but also demonstrate innovative security approaches and practices for the application of eBPF at the mechanism level.

KEYWORDS: Kernel; eBPF; Vulnerability; System Security; Operating System; Defense

目　　录

插图目录

表格目录

第一章　绪论

1.1　选题背景

在数字化时代的大背景下，现代社会各个维度——从民生到国防、从个人的智能设备到远端的云服务器、从日常简单任务到复杂的大语言模型推理，无不依赖于信息系统的强大功能。操作系统，作为这些信息系统运行的基石，已成为当代社会不可或缺的基础设施。内核是操作系统的核心组件，位于系统架构的最高特权层，负责管理复杂的底层硬件资源，向上层应用提供安全和高效的接口。因此，内核的安全性对整个信息系统的稳定运行至关重要，即使微小的内核漏洞也会将安全威胁放大到整个系统。

然而，现阶段主流的操作系统内核安全面临严峻挑战。以 Linux 为例，内核社区每日平均接收超过 7 个不同的崩溃报告[1]，每 4 天就有一个新的内核相关 CVE ID 被登记[2]。这些漏洞来源复杂多样，即便是经验丰富的内核开发者也难以迅速修补并发布补丁。因此内核漏洞的平均修复时间长达 66 天[3]，部分漏洞修复时间甚至超过 1300 天[4]。在这漫长的修复过程中，攻击者有机会利用未修复的内核漏洞攻陷整个系统。

尽管现有的内核安全缓解和隔离技术在一定程度上缓解了这一问题，但它们存在的局限性不容忽视。一方面，种类众多的缓解技术虽然提高了漏洞的利用难度，但并未能根本上阻止漏洞的触发，部分缓解机制因复杂性不足甚至已被证实可以被攻击者稳定绕过。另一方面，隔离技术通常依赖于复杂的软硬件机制，难以在系统内实现实时部署，而突发的安全事件也几乎无法预测，隔离机制的部署意味着要频繁地中断运行在系统上的重要服务。尤其值得注意的是，在漏洞修复窗口不存在可行的修复补丁，因此传统的热补丁方案无从应用。

鉴于此，内核亟需针对未修复漏洞实时防御[5]，新增的内核防御方案需要满足以下条件：(1) 具备更灵活的安全策略，以应对各种成因复杂的内核漏洞；(2) 能够实时集成至内核中，无需重编译或重启系统；(3) 保证载入策略的安全性

和高效性。而 eBPF 技术[6]，以其强大的安全策略表达能力和实时部署能力，能够快速定制实时防御响应安全威胁，为满足这些需求提供了坚实的基础。此外，eBPF 的内置验证器和即时编译 (JIT) 特性确保了代码的安全和高效执行。

基于以上分析，我们认为，利用 eBPF 技术实现对操作系统内核漏洞的实时防御不仅具有较强的可行性，而且有显著的实用价值，能够直接应用于主流的开源操作系统内核中，有效提高系统安全性。

1.2 问题描述

内核漏洞实时防御与传统内核漏洞防御的主要区别在于时效性[5]，要在漏洞公开后修复前的时间窗口中，尽可能快地实现安全、高效的防御，且防御策略无法参考专家给出的专业修复意见。而等待防御的内核漏洞往往成因复杂，种类多样，出现位置和攻击者的攻击方式都无法预测。这导致现有防御研究都很难实现对内核漏洞的实时防御，本文将在第 二 章相关研究工作 §2.1 到 §2.3 三个章节，详细梳理并深入探讨现有研究方案。

现有防御方案根据防御动机可以被分为 (1) 漏洞利用前防止漏洞触发，(2) 漏洞利用中提升利用难度，(3) 漏洞利用后限制攻击范围，它们无法用于实时防御的原因如下：

对于漏洞利用前阶段，尽管热补丁技术[7]能够针对多种漏洞类型实现安全、高效的修复，但该技术的设计依赖于内核修复补丁的存在，而漏洞修复窗口中无补丁可供参考。而内核漏洞预防工作，包括内存漏洞预防[8-9]和消毒器[10]通常开销巨大且仅针对单一类型漏洞，通常更适合作为漏洞探测工具协助开发者修复漏洞，而难以作为运行时内核实时防御方案。

对于漏洞利用阶段，内核缓解机制的原理是提升攻击利用难度[11]，而非从根本上阻止漏洞的触发，因此应该尽可能提升复杂性避免被攻击者绕过。而现有的缓解机制，特别是对于攻击风险最高的堆漏洞缓解机制，由于复杂性不足几乎都可以被攻击者确定性地绕过，无法有效阻止攻击的发生。同理内核裁剪[12]和系统调用过滤[13]也仅基于最小特权原则缩减内核攻击面，提升代码重用攻击难度。

对于漏洞利用后阶段，分隔化技术难以在系统内实现实时部署[14]，而出于

性能考虑，我们只能在漏洞公开后仅选择对漏洞部分部署分隔化，而非在系统启动时对所有内核组件部署分隔化。此外，完整性保护策略通常专注于控制流或数据流的完整性，将多种检测策略叠加使用往往会导致系统性能开销大幅增加[15]。

鉴于此，现有内核亟需开发更加灵活、敏捷的实时漏洞防御机制，以有效应对日益复杂多变的安全威胁，确保操作系统内核的安全稳定。eBPF 技术具备强大的安全策略表达能力和实时部署能力，能够满足内核漏洞实时防御的迫切需求，为其实现提供了坚实基础。因此本文选择改进 eBPF 技术，将创新性的内核漏洞实时防御方案注入内核。本文将在第 二 章相关研究工作的 §2.4 中详细介绍 eBPF 技术的基本原理，讨论 eBPF 的优势和不足，并给出解决内核漏洞实时防御难题的改进思路。

1.3 研究意义和贡献

回顾 §1.1 提出的灵活、实时部署和安全高效性需求，本文由浅入深，分别针对内核漏洞利用的中、前、后三个阶段提出了对应的实时防御方案，对未修复内核漏洞的面对的严峻挑战做出回应。在策略层面，三个工作在安全性上达到或超过了现有漏洞防御方案的水平，同时均针对内核场景进行性能优化和详细的测试。在机制层面，改造和提升了现有的 eBPF 生态系统，实现了防御策略的实时部署。

对比相关研究工作，基于 eBPF 技术的防御方案不仅在宏观上实现了对内核漏洞的实时防御，针对各个利用阶段的防御策略也有显著的能力提升。对于漏洞利用前阶段，eBPF 技术能够针对不同类型内核漏洞定制实时防御策略，防止漏洞触发，不再受限于内存漏洞预防机制难以接受的巨大开销；对于漏洞利用中阶段，eBPF 技术也能够为内核动态载入复杂程度更高的安全缓解策略，提升漏洞的利用难度，而反观现有的内核缓解机制因复杂性不足均能被轻松绕过；对于漏洞利用后阶段，eBPF 技术支持实时部署对分隔区访问控制策略的强制执行，有效防止攻击者利用攻击元语的攻陷整个系统，对比之前的内核沙箱研究工作取得了实时部署这一全新进展。

本文的主要贡献如下：

1. **ERA: 基于 eBPF 的内核堆漏洞缓解框架**。ERA 率先将 eBPF 技术应用在漏洞

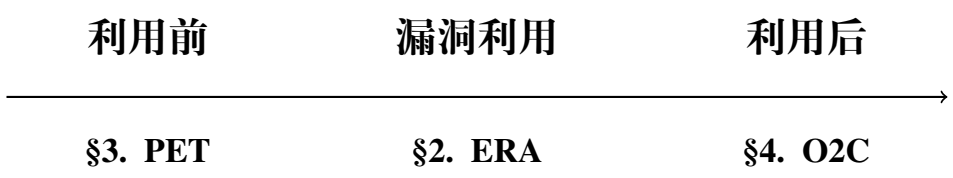


图 1-1 本文主要贡献

防御领域，通过改造 eBPF 生态系统在漏洞利用过程中提升内核堆漏洞的利用难度。内核堆漏洞由于攻击利用稳定性较高，目前公开的漏洞利用案例中有超过 80% 针对内核堆溢出和释放后使用类型漏洞。尽管现有的缓解机制能提高攻击难度，但随着时间的推移，攻击者发现了稳定的绕过方法。ERA 通过随机化技术破坏数据对象布局，阻止攻击者利用堆风水技术制造漏洞数据对象与攻击载荷的重叠，极大提升了攻击难度。ERA 首先实现了数据对象类型敏感的静态分析技术识别出了漏洞数据对象的分配点，随后改造 eBPF 生态系统使其支持内核堆安全分配器功能，并能够替换原有的漏洞数据对象分配点，将分配的数据对象替换为使用空间随机化保护的数据对象。最后 ERA 将构造完成的 eBPF 程序准确装载在漏洞数据对象分配点上，提升堆漏洞的攻击难度。经过试验评估，ERA 能有效应对真实世界的漏洞，压力测试下性能和内存开销均微不足道。

2. **PET: 基于 eBPF 的防内核漏洞触发框架**。 PET 的主要贡献是在漏洞利用前阶段，在没有修复补丁参考的情况下，通过动态检查漏洞触发条件防止整数溢出、越界访问、释放后使用、未初始化访问以及数据竞争多种类型的内核漏洞触发。对比 ERA，PET 将实时防御扩展到更多内核漏洞类型。由于在漏洞触发前阶段，现有工作无法在没有修复补丁的情况下实现对多类型漏洞触发的阻拦。因此 PET 以消毒器漏洞报告为输入，构建了可以通过 eBPF 程序运行时检查的触发条件。一旦漏洞触发条件满足，PET 将首先控制内核跳过漏洞触发点，其次处理内核触发点所在函数的内存分配、锁等配对操作，设置漏洞触发点调用者函数能够妥善处理的错误码，最后在退出内核前给触发漏洞的恶意进程发送 `SIGKILL` 终止危害。PET 深入分析上述漏洞类型相关的消毒器报告和实时防御策略，实现了对 KASAN 报告的访问越界和释放后使用类型漏洞，KMSAN 报告的未初始化访问，UBSAN 报告的整数溢出，以及 KCSAN 报告的数据竞争类型漏洞的实时防御方案。在实验中，PET 有效地阻

止内核漏洞的触发，在同时防御多个漏洞的情况下也能保持 3% 以内的轻量级性能开销。此外，PET 通过改造 eBPF 生态系统支持漏洞类型无关运行时检查机制，有潜力在未来实现更多类型内核漏洞的实时防御。

3. **O2C: 基于 eBPF 和机器学习审计的实时内核分隔探索**。O2C 的主要贡献是在漏洞利用后阶段提供一种探索性的内核分隔化方案，以应对现有内核分隔化的强制执行需要中断系统服务的缺点。O2C 突破了 ERA 和 PET 对漏洞利用、触发方式的限制，通过为不可信内核组件创建隔离区域，分隔化技术可以有效防止这些组件的漏洞威胁到整个系统。然而，现有的分隔化需要重启系统进行强制执行，在此条件下想要实现对突发的内核漏洞进行分隔化，只能预先分隔化所有内核组件或频繁地重启系统分隔化内核组件，前者可能造成系统运行时开销过大，而后者不可避免地中断了系统上运行的关键服务。O2C 采用了与传统方法截然不同的策略，探索了实时内核分隔的可行性，并揭示了其中主要挑战在于解决状态切换风险。O2C 在 eBPF 的协助下通过两项技术创新实现了其目标：首先，通过将机器学习模型直接嵌入内核对状态切换风险进行审计，为内核提供了智能化的安全保障。其次，利用动态插桩技术实现了基于 eBPF 的分隔化，无需系统重启或中断关键服务。实验评估显示 O2C 能有效地将安全威胁限制在分隔区内，决策树模型因其对表格型数据的优势和可解释性，更适合协助 O2C 审计状态切换风险，O2C 在同时分隔化 3 个组件超过 255,000 行代码时也 O2C 仅对系统造成了小于 4% 的性能开销，具备出色的可扩展性。

第二章　相关研究工作

操作系统内核的漏洞防御是系统安全领域的重要研究方向，相关的防御方案涵盖的动机和应用范围较广。为了更好地梳理相关工作，凸显实时防御工作的重要价值，本文以攻击者攻击利用内核漏洞的过程为主线进行建模，将成功的内核漏洞利用拆分为前、中、后三个阶段。每个阶段有特殊的攻击目标以及针对性的防御方案，如图 2-1 所示，此分类方式不仅详细梳理的各阶段防御方案的基本原理和应用范围，也为支持了 §1.2 中对现有防御机制的安全性不足的论证。我们在本章节中将分别深入介绍并探讨攻击利用前、中、后三个阶段的攻击目标和现有的防御方案。

其次，本文重点介绍了 eBPF 技术的基本原理及其在内核安全领域的应用潜力。我们将展示 eBPF 技术凭借其卓越的安全策略表达能力和实时部署能力，已经实现的对系统性能显著优化的研究工作。同时，本文还将展示 eBPF 技术如何为实现内核漏洞的实时防御提供强有力的支持。此外，我们也会在本章节统一阐述 eBPF 技术当前面临的挑战和局限性，说明本文研究工作对 eBPF 生态系统改造的必要性和 eBPF 自身的安全问题。

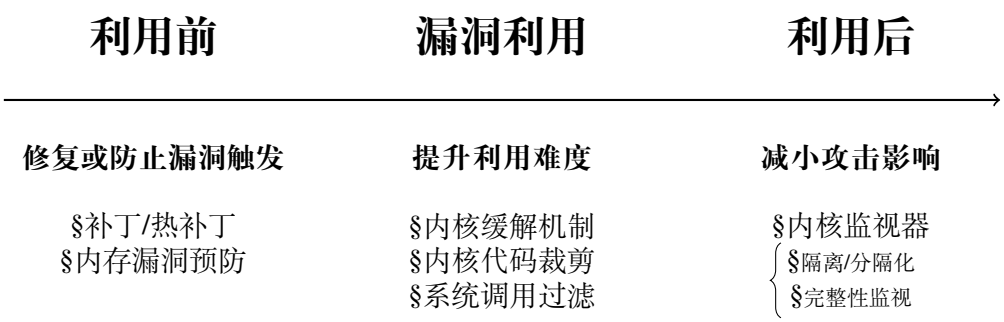


图 2-1　漏洞利用各阶段的防御方案

2.1 漏洞利用前阶段防御方案

在漏洞利用前阶段，攻击者通常集中力量于探测和验证漏洞的具体触发条件，进而显露出内核漏洞所带来的安全威胁，为后续攻击利用铺平道路。对此，防御方的主要策略是通过快速修补已知漏洞或采取预防措施阻止漏洞被触发，从根本上阻断攻击者的企图。这一阶段的防御措施关键在于提前介入，通过早期发现漏洞、快速响应和精准的修补策略，以有效避免内核漏洞成为攻击者可利用的入口。代表性的工作包括热补丁技术和一系列内存漏洞预防技术。

补丁/热补丁

内核漏洞的修复通常是开发者在分析和理解漏洞成因后，以补丁[16-18]的形式将修改的内核代码发布到社区邮件列表中。内核的维护者将修复补丁应用在存在漏洞的内核代码上，重新编译内核、载入并重启系统进行激活。热补丁 (hot-patch) 技术弥补了应用补丁需要重新编译并重启系统的缺陷，能够在存在漏洞的内核运行时应用漏洞补丁。其中 kpatch, kgraft, ksplice，livepatch[7,19-21]在内核地址空间中加入了应用修复补丁的内核代码，在执行到漏洞触发点时，将控制流转移到修复的内核代码进行执行。LUCOS[22]使用虚拟化技术，在 hypervisor 中完成对 Guest 模式下内核的热补丁加入、终止和回滚。KARMA[23]在内核中实现了一个 Lua 引擎，通过将内核补丁转换为对应的 Lua 脚本实现了运行时补丁。Vulmet[24]结合了最弱前置条件 (weakest precondition)，根据源代码修复补丁定位了在运行时内核中的漏洞修复位置和约束条件。

除此之外，还有 seamless[25]使用检查点记录了用户程序的状态，在内核达到静止状态后将其替换为修复完成的新内核，并恢复用户程序运行。Kshot[26]使用了 Intel SGX 飞地 (Software Guard eXtensions enclave) 和 SMM(系统管理模式) 技术，在可信执行环境中准备和应用内核漏洞补丁，防止不可信的内核破坏热补丁修复过程。

内存漏洞预防

内存漏洞预防方案[15]的目标是从漏洞根源出发，在运行时检查或破坏漏洞触发条件。相关工作的研究热点集中在用户空间程序中，代表性技术方案包括但不限于，针对堆溢出漏洞的数据对象边界检查[8]，和精度更高的指针边界检

查[27]，前者在运行时检查内存访问是否超出数据对象合法边界，后者将其进一步精确到数据对象内某个成员的合法访问区间。针对释放后使用漏洞，防止悬空指针解引用已释放数据对象的方案有：(1) 隔离 + 扫描[28-31]，通过扫描内存，保证程序中不存在悬空指针再释放被隔离的数据对象。(2) ID 匹配[9,32]，分配时为数据对象分配 ID，在运行时检查保证只有 ID 匹配的指针解引用，ID 可以也是类型、指针数量[32]等特征。(3) 一次性分配[33-34]，通过调整页表映射，在数据对象释放时仅回收物理内存，虚拟地址则不进行回收。上述工作由于性能和资源的开销过大导致很难直接应用在内核上，因此内核选择上述方案进行性能优化。Kruiser[35]采用了数据对象边界检查方案防止溢出漏洞，它在数据对象的前后设置 canary，并使用额外的内核线程并行地检查 canary 降低性能开销。Vik[36]则采用了 ID 匹配方案防止释放后使用漏洞，它使用更高效的 ARM MTE 硬件机制替换了代码实现的 ID 匹配检查。

除此之外，动态内核漏洞调试工具消毒器 (sanitizer) 也可以视为一种漏洞预防工作，但通常开销过大默认不作为运行时防御机制使用[10]，仅帮助开发者理解漏洞触发上下文[10,37-39]，常见的消毒器包括，探测内存损坏的 KASAN[40]，探测未初始化访问的 KMSAN[41]，探测数据竞争的 KCSAN[42]，以及探测包括整数溢出、数组越界等未定义行为 (undefined behavior) 的 UBSAN[43]等。上述消毒器都需要对内核的所有内存访问指令插桩并进行运行时检查，其中 KASAN 和 KMSAN 在内核地址空间中分配影子内存，记录地址对应内存的使用情况，用于运行时检查；KCSAN 随机选择插桩点增加时间间隔，检查间隔期间是否有其他代码访问了共享数据，以确定是否有数据竞争发生。而 UBSAN 则针对具体的未定义行为在指令执行之前进行相关的约束检查。

2.2 漏洞利用阶段防御方案

进入漏洞利用阶段，攻击者的目标是以特定方式触发漏洞，进而窃取部分内核权限，例如通过 ROP Gadget 缝合或堆风水获取信息泄露、代码指针等攻击原语。防御方采取的策略是显著增加漏洞利用的复杂度，从而使攻击者难以获得必要的攻击载体和资源。代表性的工作包括内核目前采用的一系列漏洞缓解机制，基于最小特权原则阻断或干扰攻击者的利用路径的内核代码裁剪和系统调用过

滤。

内核缓解机制

内核缓解/强化/自保护 (mitigation/hardening/self-protection) 机制[11]，与内存漏洞预防不同，是一系列被应用在内核中的降低漏洞触发风险的机制。例如，kernel canary 防止攻击者通过栈溢出攻击内核[44]。内核地址空间布局随机化 (KASLR) 随机化内核地址空间布局[45]，使攻击者无法准确定位内核代码、数据地址。内核栈随机化[46]，在每次系统调用进入内核时，为内核栈栈顶随机分配一个偏移量，使基于栈溢出的任意写原语无法准确修改内存。

针对内核堆的缓解机制则大量借鉴了用户层的安全分配器的设计思路[47-51]。例如针对内核小数据对象 slab 分配器的 slab nomerge[11]能防止数据对象类型不同但大小相同的 slab cache 互相污染；freelist 随机化[52]能避免数据对象在连续的地址上分配，提升攻击者通过堆风水放置攻击负载的难度；freelist 指针模糊化[53]通过简单加密算法提升攻击者破坏 slab 分配器元数据的难度；freelist 指针 naive check[54]防止连续两次释放相同地址，提升了双重释放攻击的难度；分配时置零和释放时置零，能够有效防止数据对象内容未清空导致的信息泄露。kfence[55]为数据对象分配额外的内存进行随机化，但出于性能考虑最大只支持 512MB 内存池，每 ms 选择一个数据对象进行随机化，经过实际测试仅有 0.005% - 0.35% 被保护。数据结构成员布局随机化[56]在编译时随机化了数据对象成员的布局，攻击者很难准确破坏数据对象成员实现利用。

此外，缓解机制还有 PAX_USERCOPY[57]，检查内核和用户交互数据对象的长度防止越界。内核页表隔离[58]，防止硬件漏洞侧信道等攻击[59]窃取内核机密数据。以及使用 CPU 硬件机制的 SMAP/PAN，SMEP/PXN[60]，防止内核发起 ret2usr 攻击[61]访问用户数据或执行用户代码，使用 Intel CET 机制[62]保证粗粒度控制流完整性。

内核代码裁剪

内核代码裁剪 (kernel debloating) 的原理是基于最小特权原则，裁剪未被内核执行的内核代码，减小攻击面提升攻击者发起代码重用攻击的难度。FACE-CHANGE[12]为每个应用程序裁剪了支持运行的最小内核镜像，在运行时使用基于虚拟化的系统进行切换。KASR[63]在 hypervisor 中通过配置 EPT 取消内核未

被使用代码的执行权限，再根据实际的执行情况分段重新激活将要执行的内核代码权限。SHARD[64]提出了基于进程和系统调用的更细粒度的内核裁剪。此外，Hacksaw[65]从系统的底层硬件驱动出发，通过依赖分析反向定位到需要保留的内核函数。

系统调用过滤

系统调用过滤技术同样基于最小特权原则，限定程序能够使用的系统调用范围，减小了内核的攻击面，提升了攻击者攻击难度。Lock-in-Pop[13]将系统调用的合法路径范围限制在存在较少内核漏洞的系统调用路径。Confine[66]使用静态分析技术提取容器运行访问的系统调用集合。Temporal specialization 过滤器[67]为大型程序的不同阶段设置不同的合法系统调用集合。C2C[68]基于程序的配置信息对系统调用合法集合进行进一步裁剪。Xing 的工作[69]结合静态和动态分析方案，对容器的合法系统调用集合进行限制。Saphire[70]针对使用解释器执行的不同 PHP 程序设置单独的系统调用合法集合。SysXCHG[71]进一步解决了子进程由父进程继承而来的系统调用合法集合过大的问题，为每个进程设置了单独的系统调用合法集合。sifter[72]通过检查系统调用执行的模式来保护内核 GPU 驱动等安全敏感模块。

2.3 漏洞利用后阶段防御方案

在漏洞利用后阶段，攻击者已经成功获取攻击原语并计划继续扩大危害范围，例如利用获取的攻击原语去执行攻击负载代码、修改安全敏感数据对象或载入 rootkits。此时防御者的策略转向封锁攻击者的活动范围和限制其影响力，目的是最小化攻击对系统的整体破坏。代表性的解决方案是构造内核监视器部署隔离或完整性监控策略，防止攻击者攻陷整个系统。

内核监视器

内核监视器 (reference monitor) 是 James P. Anderson 在上世纪提出的系统安全架构[73]，它要求具备：不可绕过 (complete mediation)，自保护 (tamperproof)，可验证 (verifiable) 三个特性。其中不可绕过要求内核监视器内部署的安全策略一定会执行；自保护特性要求内核监视器与被监视的系统存在权限差，使得内核

漏洞无法破坏监视器自身的完整性；可验证要求监视器必须能够被分析和测试，以确保不可绕过和自保护特性的完备。因此内核监视器可以作为可信基 (trusted computing base) 被应用在内核不可信的强威胁假设环境中，始终是系统安全领域的研究热点。

部分相关工作致力于探索内核监视器机制的构建，可行的技术路线包括：使用 Hypervisor[74-79], TEE[80], SMM[26]机制的更高特权层；使用内核页表[81-84]，MPK[85], SGX 等硬件机制[86]，使用软件错误隔离技术[87-89]，甚至在硬件扩展层面[90-92]，在相同特权层对内核进行降权等。部分工作通过实现 IDL(interface definition language) 简化了不可绕过的安全检查策略的维护[89,93-95]。以及 PCM[96]通过形式化技术验证监视器的安全性，Decentralized-KPD[97]构造多可信基防止监视器体积膨胀带来的额外风险。

而研究者在构造可靠的内核监视器机制后，通常对采用隔离/分隔化和完整性保护两种安全策略对防止内核漏洞进一步破坏内核。

隔离/分隔化

内核隔离/分隔化 (isolation/compartmentalization) 技术的基本原理是，在共享地址空间的宏内核系统中，强制执行不可信的部分代码对内核资源的访问控制策略，防止不可信部分的漏洞破坏整个系统。

最初的分隔化机制通常针对不可信的第三方内核模块/驱动，代表性工作包括：使用多套内核页表隔离不可信驱动的 NOOKS[83]，使用了 Hypervisor 的 EPT 页表隔离不可信模块的 HUKO[98]，隔离不可信子系统的 LXD[93]，后续通过 `vmfunc` 降低此类方案性能开销的 LVD[94]，研究交互接口复杂性的 KSplit[95]。将不可信驱动放在用户层执行的 SUD[81]和 SIDE[82]。使用编译器插桩检查不可信模块的 SFI[87]和 XFI[99]，后续的 BGI[88]使用了影子内存来管理访问控制策略，LXFI[89]引入了一种 IDL 维护分隔区和内核交互接口的完整性 (API Integrity) 等。

随着隔离技术的不断发展，分隔化技术开始逐步取代传统的不可信模块隔离，分隔化技术的隔离目标不局限于不可信内核模块，而是内核的任意组件，隔离的范围也可以根据实际需求从单个函数到完整的子系统之间进行调整。目前的相关工作包括，μScope[100]分析了基于最小特权原则的分隔区规模与性能开销的关系。HAKC[14]结合了 ARM 架构的 PAC 和 MTE 机制约束了分隔区对内

核资源的访问。SecureCells[90]采用了特殊的 CPU 机制来强制执行分隔化策略。ConfFuzz[101]使用了模糊测试技术分析了分隔区交互接口的安全漏洞等。

完整性保护

完整性保护也是内核监视器强制执行的安全策略。SecVisor[102]通过虚拟化技术保护内核的代码完整性。对于内核控制流完整性，KCoFI[103]借助 SVA[104]架构实现了对前向和后向间接跳转地址的运行时检查。PAL[105]使用了 ARM 平台的 PA(pointer authentication) 硬件特性强制执行内核的控制流完整性检查。Fine-CFI[106]结合了上下文敏感和数据对象敏感两种指针分析技术获取高精度控制流图。MLTA[107]使用了多层类型分析进一步提升内核数据流图的精度。对于数据流完整性保护，PT-Rand[108]通过随机化内核页表存储位置并加密指向页表指针防止内核页表被恶意攻击[109]，类似的还有 xMP[110]利用虚拟化的 EPT 访问权限设置保护内核的安全敏感数据，使用加密技术防止指向安全敏感数据的指针被修改；Chengyu 等[111]通过将分析识别出的敏感数据对象放置在隔离区域中，并借助 SFI 技术进行保护。除此之外，内核监视器也在内核不可信的情况下保护应用程序完整性[104,112-113]。

2.4 eBPF 技术

eBPF 技术目前是实现内核漏洞实时防御的理想内核基础设施之一，得益于其卓越的安全策略表达和实时部署能力，它能够为内核实时防御需求提供灵活的定制安全策略。因此我们将 eBPF 技术作为研究内核漏洞实时防御方案的基础。本文首先介绍 eBPF 技术的基本原理，详细阐述了支持 eBPF 关键功能的生态系统组件及 eBPF 程序的工作流程。在此基础上本文列举了近期 eBPF 技术在系统性能优化领域的代表性进展和安全领域的尝试，同时指出 eBPF 现有的局限性，说明了本文工作对 eBPF 生态系统改造的必要性和 eBPF 自身的安全问题。

eBPF(extended Berkeley Packet Filter) 技术提供了在内核特权环境中运行沙箱程序的能力，使得开发者可以安全而有效地扩展内核功能，而无需修改内核代码或加载内核模块。eBPF 技术起源于 BPF(Berekley Packet Filter) 网络包过滤器，其最初被设计为一套虚拟指令集和在内核地址空间运行的解释器。随着时间的推移，eBPF 的能力被不断挖掘和扩展，现已应用于网络管理、可观测性和安全

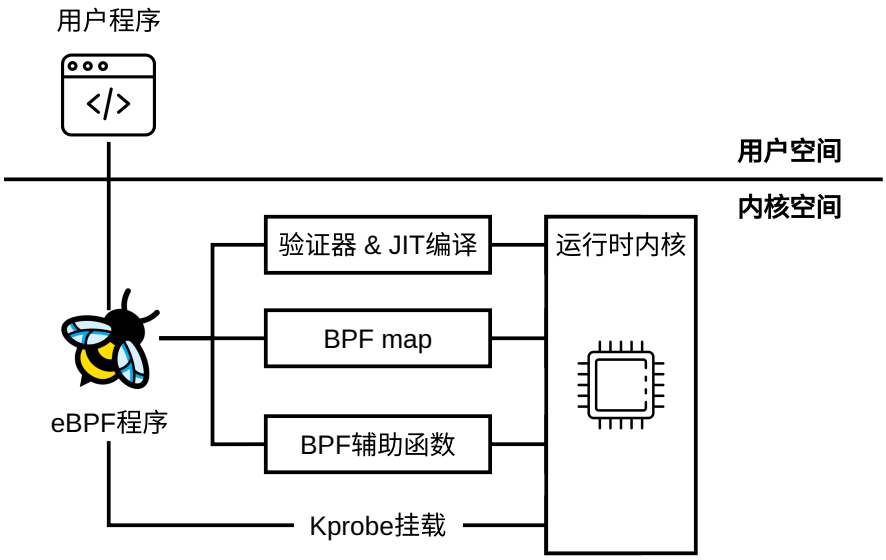


图 2-2　eBPF 架构示意图，上层为用户空间程序，下层为内核空间 eBPF 主体架构

审计等领域，是近些年内核领域的革命性技术之一[6]。

eBPF 的强大功能得益于其生态系统的多个关键组件，如图 2-2 所示。验证器用于在 eBPF 程序载入阶段对指令进行检查，它首先对指令进行静态分析，禁止循环、递归、调用内核函数、指针运算等不安全操作并移除不可达指令，其次模拟执行 BPF 指令观察寄存器和栈的状态，确保载入程序不会突破沙箱边界破坏内核；JIT(just-in-time) 编译器能够将 eBPF 指令转化为机器码，赋予 eBPF 程序接近原生的执行效率；BPF maps 提供多种数据结构，如数组、哈希表、环形缓冲区等，eBPF 可以高效存取数据并使用 BPF maps 与用户程序交互；BPF 辅助函数 (helper function) 协助 eBPF 在严格的监控下调用部分内核函数或读写指定内核数据对象, 为 eBPF 提供了强大的策略表达能力；钩子机制，eBPF 结合了 kprobe 内核追踪子系统，能够将 eBPF 程序挂载到内核的任意指令上执行。此外，eBPF 还具备 XDP 数据处理加速，与内核的 perf 性能、LSM 安全、error injection 调试等多个功能组件连接, 进一步扩展了 eBPF 的应用范围。

eBPF 的工作流程既简单又直接，具有较低的学习门槛。开发者首先编写 eBPF 代码并使用 LLVM 编译器将其编译为 eBPF 指令，随后通过用户程序将编译好的 eBPF 指令载入内核，交给验证器检查，若检查通过即可使用 JIT 编译优化指令，并将指令保存在可执行的内核代码区域。紧接着 eBPF 程序被分别挂载在对应内核指令的钩子上，当钩子被触发时，相关的 eBPF 程序会立即执行，配合 BPF maps 和辅助函数完成既定任务。

应用

近些年 eBPF 在系统性能和安全领域都涌现出了大量代表性工作，其中系统优化相关的 Syrup[114]利用 eBPF 实现了用户定义的内核调度策略，XRP[115] 和 BMC[116]利用 eBPF 增强了内存中 key-value 存储的性能，Syncord[117]支持根据使用场景调整内核的锁机制，Electrode[118]优化了分布式协议的性能，λ-IO[119]通过 eBPF 管理计算和存储资源优化了计算存储设备的性能开销，SPRIGHT[120]减少协议处理和序列化-反序列化开销来简化 serverless 计算，hXDP[121-122] 和 eHDL[123] 在智能 NIC 硬件中将 eBPF 用于策略实现。

在安全领域，LBM[124]使用 eBPF 过滤了恶意硬件对 USB 和蓝牙内核驱动的攻击，RapidPatch[125]使用 eBPF 为嵌入式系统构造了热补丁机制，Sifter[72]使用 eBPF 过滤恶意系统调用保护安全敏感内核模块。

局限性

eBPF 程序的功能存在诸多限制，由于其设计目标是在沙箱中提供可观测性和包过滤支持，因此并未被设计成图灵完备语言。首先，eBPF 程序为了防止程序过大影响内核运行，设置了 1,000,000 指令数上限，同时栈空间仅有 512 Bytes 且不支持堆，这要求开发者必须合理规划功能实现。其次，eBPF 禁止修改内核内存或直接调用内核函数，使得上述相关工作几乎都需要扩展 eBPF 生态系统的功能来实现既定任务，例如 Syncord[117]新增辅助函数实现了对内核锁机制的支持，XRP[115]新增了 eBPF 程序类型支持 key-value 存储。此外，eBPF 在内核环境运行，默认不对浮点数运算做任何支持。显然现有的 eBPF 生态系统无法直接应用于内核漏洞实时防御，本文也将在后续工作中根据实际需要使用合理的方案进行改进。

eBPF 技术虽然通过验证器大幅降低了安全风险，但其本身仍然存在被恶意利用的可能性。一方面，仍在活跃开发的 eBPF 不可避免地会引入漏洞[126]，但漏洞集中在 eBPF 验证器、辅助函数等基础设施。另一方面，攻击者也可以恶意地使用 eBPF 程序破坏系统安全，例如 EPF[127]展示了攻击者如何使用载入内核的 eBPF 程序攻陷整个系统，Cross Container Attacks[128]发现了在云环境中通过 eBPF 攻击其他容器的攻击面，ebpfkit[129]研究了如何使用 eBPF 构建 rootkits 等。然而，值得注意的是，只有拥有 root 权限的用户才能将 eBPF 程序载入内

核，没有 root 权限的攻击者既无法通过载入 eBPF 程序触发漏洞，也无法恶意利用 eBPF 程序攻击内核。因此在本文后续的工作中，我们将不会深入探讨 eBPF 子系统的漏洞和 eBPF 潜在的攻击风险。

第三章　ERA: 基于 eBPF 的内核堆漏洞缓解框架

3.1　引言

本研究率先将 eBPF 技术引入系统安全领域，进一步展现了 eBPF 在安全策略表达和实时部署方面的出色能力，特别是在实现内核漏洞的实时防御上。通过对 eBPF 生态系统的创新改造，本研究成功开发了动态安全堆分配器，实现了在漏洞利用阶段对内核堆漏洞的有效实时防御。以下将详细介绍本研究的核心内容及其成果。

内存损坏漏洞 (memory corruption) 是由不安全的 C/C++ 语言编写的程序的主要安全威胁之一，有报告显示近年来内存损坏漏洞的主要形式为堆漏洞[2,34]，常见的包括空间型 (spatial) 溢出漏洞[27]和时间型 (temporal) 释放后使用漏洞[130]。攻击者能够通过“堆风水”攻击手段对应用程序堆进行布局，将攻击目标数据对象放置在漏洞数据对象对应位置，进而触发漏洞破坏系统的机密性和完整性[131]。例如，利用溢出漏洞获取邻接数据对象的机密信息、修改敏感数据，或将攻击负载放置在释放后使用漏洞被释放的地址，并解引用悬空指针触发[132-133]。操作系统 Linux 内核的小数据对象分配器 slab 也受到相同的威胁，近三年公开的 57 个内核漏洞中有 40 个属于堆漏洞，用户层的攻击者通过一系列精心构造的系统调用，触发内核堆的内存损坏漏洞获取核心权限，进而控制整个系统[60,134]。

然而从发现漏洞到开发者给出补丁 (patch) 并应用修复往往需要较长时间，因为近些年来漏洞数量和复杂性剧增，但确认并修复漏洞要求开发者必须同时理解漏洞的根源和发现漏洞子系统的功能，否则对漏洞的修复可能无效或影响内核运行效率，这导致 Linux 中部分漏洞存在的时间超过 1300 天[4]，存在漏洞的内核难以保证系统安全性，因此，在这一时间窗口中内核需要缓解 (Mitigation)[135-136]机制降低内核漏洞的安全威胁性。

本研究对比了目前开源 Linux 内核采用的堆漏洞缓解机制，发现其中存在安全保障不足、无法灵活选择保护对象两类缺陷，导致漏洞缓解效果未达到预期，

无法应对真实环境中复杂多样的内核堆漏洞。(1) 安全保障不足是指现有缓解机制存在确定的绕过手段，例如 freelist 随机化[137]和 slab quarantine[138]可以被堆喷射[139]绕过、数据结构成员随机化[140]种子易于被获取、autoslab 难以抵抗同类型对象攻击等；(2) 无法灵活选择保护对象的原因有两方面，一方面是缓解机制依赖编译前静态配置，无法在系统运行时灵活增减，另一方面部分缓解机制在原理层面仅针对小范围安全风险，例如 freelist 指针模糊化[141]仅保护元数据、成员布局随机化和 autoslab 仅对固定成员数量和长度的数据结构生效，对于可伸缩和缓冲区对象均无能为力等。

有效的内核堆漏洞缓解机制应该达到以下目标：(1) 无法被确定性攻击，(2) 不依赖静态配置，支持运行时对任意存在威胁的数据对象实施，(3) 性能开销可以被操作系统接受。为满足以上目标，动态缓解机制需要合理设计更复杂的随机化方案，提供更强的安全保障，但不引入大量开销，同时应该具备部分类似热补丁的动态载入的特性，不影响系统正常运行。

我们提出了一种基于 eBPF[6]的内核堆漏洞动态缓解框架 ERA(**e**BPF-based **R**andomization **A**llocator)，用于在内核堆漏洞被修复前的时间窗口内，动态、高效缓解操作系统面临的安全风险，既无需等待安全专家发布修复补丁，又避免了内核探测追踪引入的巨大开销，而且可以根据漏洞报告中的安全威胁数据对象生成缓解程序，具备强大的易用性。ERA 动态缓解框架相比其他内核缓解机制，不存在确定性的绕过手段，为内核提供了更充分的安全保障；同时可以在运行时选择任意数据对象进行保护，不局限于运行前配置和数据结构类型。

ERA 采用了数据对象空间随机化方案提供了充足的安全保障。每次内存分配时在空间更大的 slab cache 中请求超过所需的内存，并在分配的内存中随机放置数据对象。这使得每一个空间随机化的数据对象均不存在有关联的地址，攻击者无从预测数据对象所在地址和偏移量，很难准确放置攻击负载，内核堆漏洞利用的难度急剧放大，其他内核缓解机制中提及的堆喷射、同类型对象攻击、cross-cache[142]攻击等绕过手段均无法突破 ERA 防护。

ERA 能够灵活地在运行时对任意数据对象施加保护，足以应对真实环境中复杂多变的堆漏洞。首先 ERA 充分利用 eBPF 动态、安全特性，可以在运行时将经过空间随机化的数据对象注入内核中，无需预先配置或重新编译内核。其次，ERA 采用的数据对象空间随机化不受数据对象成员的限制，包括可伸缩对

象、缓冲区对象在内的任意数据对象，均能够在内核地址空间中隐藏自身位置，相比其他内核堆漏洞缓解机制具备更广阔的适用范围。

本章节贡献如下：

1. 提出了一种内核堆漏洞动态缓解框架。采用动态的数据对象空间随机化模型，能够在内核堆漏洞修补前的攻击窗口降低安全威胁，同时利用 eBPF 技术动态、灵活、安全地将空间随机化应用于任意数据对象，具备易用性、灵活性、安全性和高效性。
2. 研究了支持内核堆漏洞动态缓解的关键技术。采用静态分析技术获取了精确的分配点上下文，控制数据对象随机化的开启和关闭; 设计了更加安全高效的数据对象空间随机化方案，解决了现有缓解机制的缺陷; 充分利用 eBPF 提供的动态、安全特性将被保护的数据对象重新注入内核，不引入额外风险。
3. 实现了原型系统 ERA，从收集到的漏洞报告出发，提取存在安全威胁的数据对象并生成 eBPF 程序载入内核，缓解内核堆漏洞安全风险。
4. 实验验证了 ERA 的有效性、高效性、创新性和易用性。首先验证 ERA 的有效性，我们评估了 40 个具有代表性的内核堆漏洞，并选取其中 12 个漏洞的 EXP 攻击程序进一步确认 ERA 具备快速缓解堆利用风险的能力；其次为验证 ERA 的高效性，我们选取了内核中大量分配的数据对象，并对比了原始内核 (编译时成员随机化)，仅增加了约 1% 的性能开销，同时内存消耗也因为回收机制微不足道；此外，与成员随机化的优化工作 SALAD 和 POLAR 的对比结果凸显了 ERA 机制的创新性；最后，我们确认了 ERA 的易用性，即使非安全专家的系统管理员，也无需等待漏洞修补补丁，提前缓解内核堆漏洞利用风险。

3.2 背景

3.2.1 威胁模型

本章节研究重点针对操作系统内核中的堆分配器 (slab) 内存损坏漏洞，主要包括溢出 (out-of-bound) 和释放后使用 (use-after-free) 两类。堆漏洞是目前内核的主要安全威胁之一，我们统计了近三年来公开的 57 个内核漏洞，其中有 40 个属于堆漏洞。其他类型的内核漏洞，例如 TOCTTOU、竞争和未初始化等作为攻击

的一环[142]，最后可能也会诱发或辅助堆漏洞利用。因此我们假设攻击者了解内核漏洞，但不具备核心 root 权限，系统开启了 freelist 随机化、指针模糊化、naïve check 和数据结构成员随机化等缓解机制提升堆漏洞攻击难度，开启了 SMAP、SMEP、NX 等硬件机制避免 ret2usr 和代码注入攻击，测试的 EXP(exploit 攻击利用程序) 通过堆风水、堆喷射一系列攻击绕过上述安全防护，试图在用户空间通过执行系统调用触发漏洞，读取或修改内核信息，进而获取 root 权限。

但是暂时不考虑的 ERA 方案依赖的 eBPF 机制本身存在的漏洞，并且假设 ERA 载入前漏洞没有被触发。内核中不存在 rootkits 形式的恶意代码[143]，不具备任意写原语任意破坏内核数据，系统的运行平台也不存在硬件漏洞或恶意硬件[124,144]。

3.2.2 内核堆内存漏洞利用

Linux 内核使用 slab/slub 分配器实现用户程序堆的功能，为内核动态分配小块内存 (小于 2 页，8192 字节)，原理是从 buddy 中分配连续的、整页的内存，划分成大小相同的数据对象进行分配，其中相同大小的一类数据对象由一个 slab cache 管理，每个 slab cache 中的数据对象以链表 (freelist) 的形式连接，分配时从 freelist 中取出数据对象，释放后再重新加入 freelist。

为了便于数据管理，内核根据分配数据的大小划分了 13 个通用 slab cache，分别是 8、16、32、64、96、128、192、256、512、1024、2048、4096 和 8192 字节，常见的内核堆内存损坏漏洞通常发生在这些 cache 中间。除此之外，内核还为 task_struct、cred 和 inode 等携带重要信息的安全敏感数据对象分配了专用的 slab cache 与通用 cache 隔离，由于只存在一类数据对象，因此漏洞利用风险相对较小。

溢出利用

空间型内存损坏漏洞的根源是指针 ptr 指向的地址范围 $[ptr+offset_{ptr}, ptr+offset_{ptr}+size_{access})$，超出了当前指针的合法范围 $[obj, obj+size_{obj})$，让攻击者获得了解引用超出指针合法范围的能力[15,27]。基本原理模型：

定义 1 对于一个区间的访问行为 $(ptr, offset_{ptr}, size_{access})$

被访问区间 $[obj, obj+size_{obj})$

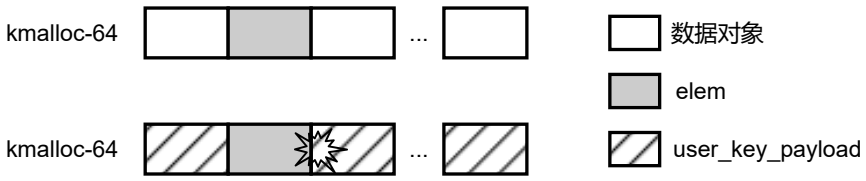


图 3-1 CVE-2022-34918 信息泄漏内核堆喷射攻击布局

如果 $(ptr, offset_{ptr}, size_{access}) > (obj + size_{obj})$，则存在一个溢出漏洞

即 $offset_{ptr} + size_{access} > size_{obj}$

在大多数空间型堆内存损坏漏洞利用的过程中，攻击者会想办法将受害者数据对象 (victim object) 放在发生溢出的指针指向的漏洞数据对象 (vulnerable object) 后面，控制发生溢出的指针读写受害者数据对象。受害者数据对象与漏洞数据对象大小一致，在同一 slab cache 中，而且会包含机密信息或安全敏感的代码指针和权限[145]。

```
1 void *nft_set_elem_init(...)
2     // 整数溢出，分配长度小于预期
3     elem = kzalloc(set->ops->elemsize + tmpl->len,  gfp);
4     ...
5     ext = nft_set_elemext(set,  elem);
6     ...
7     //此处elem地址+偏移量发生溢出
8     memcpy(nft_set_ext_data(ext),  data,  set->dlen);
```

代码 3-1 CVE-2022-34918 漏洞根源代码片段

如代码 3-1 所示，我们选取了 netfilter 包过滤器中 CVE-2022-34918 作为实际案例，在该漏洞中，第 3 行的 `elem` 缓冲区的分配大小发生了整数溢出，使得分配了小于预期的数据对象，而之后通过第 8 行的 `memcpy` 函数溢出写了超过范围的数据对象。攻击者使用堆喷射和堆风水等攻击技术，如图 3-1 所示，通过大量分配 `user_key_payload` 数据对象，绕过现有的内核缓解机制，将其放置在发生溢出的 `elem` 数据对象之后，攻击者控制溢出指针修改 `user_key_payload` 的 `data_len` 长度，恶意扩大 `user_key_payload` 的读取范围，实现任意读攻击。此外，slab 分配器元数据 freelist 由于保存在未分配数据对象中，也会作为溢出攻击的目标之一，但开发者使用了 freelist 模糊机制[141]，通过 ⊗ 加密的方式进行了缓解。

释放后使用利用

时间型内存损坏漏洞的根源在于 (悬空) 指针 p 指向了被释放的数据对象 O，释放前 O 的起始地址为 m，范围是 $[m, m+size)$。当程序解引用悬空指针 p 时能够泄露内核机密数据，或触发任意由攻击者构造的攻击负载[146]。基本原理模型：

定义 2 一个指针 p 是悬空指针，当且仅当，一个堆数据对象 O，
地址的范围是: $\forall m, size : [m, m+size)$，且这个对象被释放
指针变量 $p \in [m, m+size)$

在时间型堆内存损坏漏洞的攻击中，攻击者会在悬空指针指向的数据对象释放后，快速将精心设计的攻击负载放置在被释放的位置，在放置成功后解引用悬空指针触发攻击。与空间型漏洞类似，悬空指针指向的漏洞数据对象 (vulnerable object) 和攻击者选择的攻击负载喷射对象 (spray object) 也大小相同，在同一个 slab cache 中，喷射对象中也包含机密信息或安全敏感的代码指针和权限。

```
1  static int route4_change(...)
2      ...
3      f = kzalloc(sizeof(struct route4_filter), GFP_KERNEL);
4      ...
5      for (pfp = ...)
6          ...
7          if (pfp == f) {
8              *fp = f->next
9          } // f为新分配的route4_filter对象，被错误从bucket中删除
10     ...
11     //
         ↪ fold为本应删除的route4_filter对象，指针被错误保存，此处释放了fold内存
12     tcf_queue_work(&fold->rwork, route4_delete_filter_work);
```

代码 3-2　CVE-2021-3715 漏洞根源代码片段

如代码 3-2 所示，我们选择了网络包调度中的 CVE-2021-3715 作为实际案例，在该漏洞中开发者错将新分配的 `route4_filter` 指针在第 8 行删除，而本应删除的旧 `route4_filter` 指针保留在 `route4_bucket` 中，当旧 `route4_filter` 数据结构在第 12 行释放时，被错误保留的指针成为悬空指针。

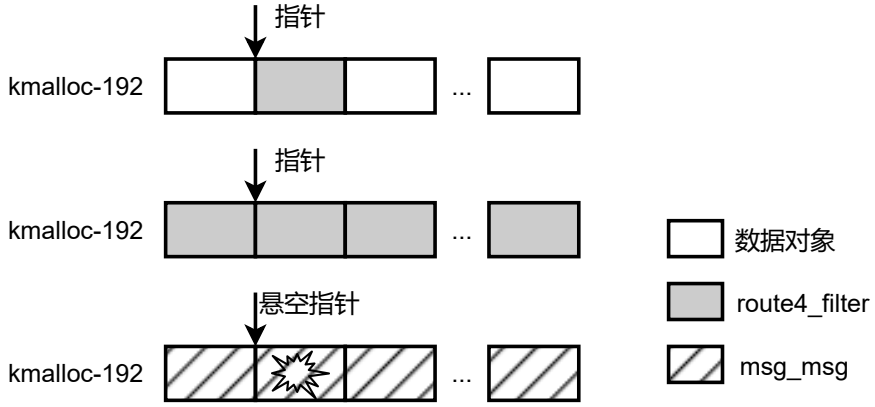


图 3-2 CVE-2021-3715 信息泄漏内核堆喷射攻击布局

攻击者选择了常见的可伸缩数据对象[147]的 `msg_msg` 实现信息泄露攻击。`msg_msg` 数据对象在悬空指针产生后快速填充了被释放 `route4_filter` 的位置。之后攻击者调用 `kfree` 函数，发起释放后使用攻击，释放了悬空指针指向的刚刚填充的 `msg_msg` 数据对象，因为 `msg_msg` 能够完整读取自身的 `security` 成员后面的全部内存，因此可以将任意携带安全敏感信息的数据对象再次填充到 `msg_msg` 被恶意释放后的位置，例如可以直接通过 `route4_filter.rwork.func` 成员泄露 `route4_delete_filter_work` 函数地址，绕过 KASLR 地址空间随机化泄露内核代码基地址。为了提升攻击的稳定性，攻击者同样使用堆喷射和堆风水等攻击技术，如图 3-2 所示，预先大量分配 `route4_filter` 数据对象，保证 `msg_msg` 数据对象能够成功覆盖到至少一个悬空指针指向的位置。

此外，双重释放 double free 也是 UAF 利用的一种类型，但是内核中给出了特定检查 (naive check)[54]，使得连续释放相同数据对象，操纵 slab freelist 的攻击行为被阻止。

3.2.3 内核堆漏洞缓解机制

内核采用堆漏洞缓解机制，在漏洞修复前的时间窗口中降低操作系统安全风险，但 §3.2.2 给出的攻击案例能够绕过现有的缓解机制，因此本文对缓解机制进行总结，从基本原理出发分析缓解机制被恶意绕过的原因。如表 3-1 所示。

Freelist 随机化[137]。在每次 slab cache 从内核中获取连续页内存拆分成固定长度小数据对象时，对拆分好的数据对象分配顺序进行洗牌 (shuffle)，使得攻击者难以预测连续分配对象是否相邻，因此攻击负载放置的难度被提升。但 freelist 随

缓解机制	缺陷	部署	绕过
Freelist 随机化	堆喷射攻击绕过	是	是
Freelist 指针模糊化	XOR 加密易于破解, 无完整性保护 静态配置, 仅对 slab 元数据	是	是
Autoslab	同类型对象攻击,cross-cache 攻击绕过 静态配置, 对可伸缩、缓冲区对象无效	是, 仅敏感对象	是
Slab quarantine	堆喷射攻击绕过	否, 未进入内核	是
Naive check	不连续释放绕过 静态配置, 仅对双重释放攻击生效	是	是
数据结构成员随机化	随机种子易获取 静态配置, 对可伸缩、缓冲区对象、 用户交互 UAPI 无效	是	是
ERA	-	-	否

表 3-1 内核堆漏洞缓解机制缺陷

机化无法应对堆喷射攻击，即攻击者通过大量分配的方式接触到攻击负载，存在安全保障不足的问题。

Freelist 指针模糊化[141]。为 slab 元数据 freelist 指针进行两次 ⊗ 加密，读取和修改必须采用 slab 分配器的专用函数，攻击者无法通过明文修改的方式操纵数据对象分配位置。但 ⊗ 加密易于破解，因为 freelist 指针由于系统架构固定以 0xffff888 开头，同一个 slab cache 的相邻数据对象指针仅有几个 bit 差别，攻击者可以尝试使用位翻转 (bit flipping) 攻击绕过[148-149]。而且 freelist 指针模糊化无法保护指针完整性，此外仅能保护元数据，对其他类型的堆攻击无效。

Autoslab[140]。由 grsecurity 内核安全公司开发，借鉴了内核中的专用 slab cache 思想，使内核中所有类型的数据对象均由专门的 slab cache 分配，不与相同长度的其他数据对象共享。这一机制很大程度上提升了内核堆风水难度，使得攻击负载难以放置。但此机制需要手动修改内核代码的所有 slab 分配位置，而且无法部署在无类型的缓冲区，对于 cross-cache 攻击、同类型对象攻击无法防御，因此目前仅作为付费软件，并未合并进入内核。

Slab quarantine[138]。数据对象释放后将其加入隔离列表，等待一段时间后再还给内核，试图以此让攻击者无法放置释放后使用攻击的攻击负载，提升攻击难度。此机制类似释放后使用漏洞探测采用的 quarantine & sweep 技术[28-29,150]，但仅隔离而缺乏扫描，没有从根源上消灭悬空指针，攻击者依旧可以通过堆喷射攻击绕过此机制。

Naive check[54]。检查连续分配的两个数据对象地址是否相同，有效避免了连续双重释放攻击的发生，攻击者仍然可以不连续释放相同的数据对象地址，进而

构造任意释放类型漏洞。

数据结构成员布局随机化[139]。在编译时对数据对象的成员的偏移量进行随机化洗牌。这样当内核堆内存损坏漏洞的漏洞数据对象和受害者/喷射数据对象的成员被随机化之后，攻击者无法获得准确的机密数据或安全敏感数据的偏移量，但存在随机性易被打破和随机化范围受限两种缺陷。(1) 由于内核需要支持可扩展模块，因此随机种子被保存在编译工程文件中易于获取。SALADS[151]和POLAR[152]分别采用了固定周期和每次分配再随机化提升随机化强度，但需要将相应取地址指令替换为查询偏移量函数，引入额外开销。(2) 成员随机化仅能编译时选择成员长度和类型确定的对象，无法保护用户交互 UAPI、可伸缩和缓冲区对象等。本文统计了 v5.15 内核版本成员随机化的使用情况，内核现有超过 50000 种数据结构，但仅随机化了其中 67 种，不足以真正提升漏洞利用难度。

3.3 内核堆漏洞的动态缓解

针对从内核漏洞发现到修复时间窗口缺乏可靠缓解机制的需求，本文提出了一种动态的内核堆漏洞缓解框架 ERA，如图 3-3 所示，系统管理员输入漏洞报告中存在威胁的数据对象，ERA 即可输出相应的缓解程序并载入内核降低内核安全风险。ERA 结合了数据对象空间随机化技术和 eBPF 技术，对任意数据对象均能够部署数据对象空间随机化保护，而且无需重新编译内核，在运行时即可将安全数据对象注入内核进行替换。

在动态替换过程中，ERA 首先定位漏洞数据对象的分配点上下文，即漏洞数据对象分配函数的调用点，当内核执行到分配点上下文时，ERA 开启数据对象空间随机化，当前分配点上下文执行过程中所有 slab 分配的数据对象地址均无法预测。ERA 将 cache 和 offset 随机化之后的地址返回给分配点上下文，同时记录该地址用于合法释放空间随机化数据对象。在当前分配点上下文执行完毕后，ERA 结束空间随机化，在此之后的数据对象恢复正常分配。当随机化数据对象生命周期结束后，ERA 在内存释放函数将随机化地址替换为记录的 cache 随机化分配的起始地址，正确释放随机化数据对象。

为了实现内核堆漏洞动态缓解框架，需要面对并解决以下挑战：

挑战 1：分配点上下文定位。分配点上下文决定了数据对象空间随机化的开启和

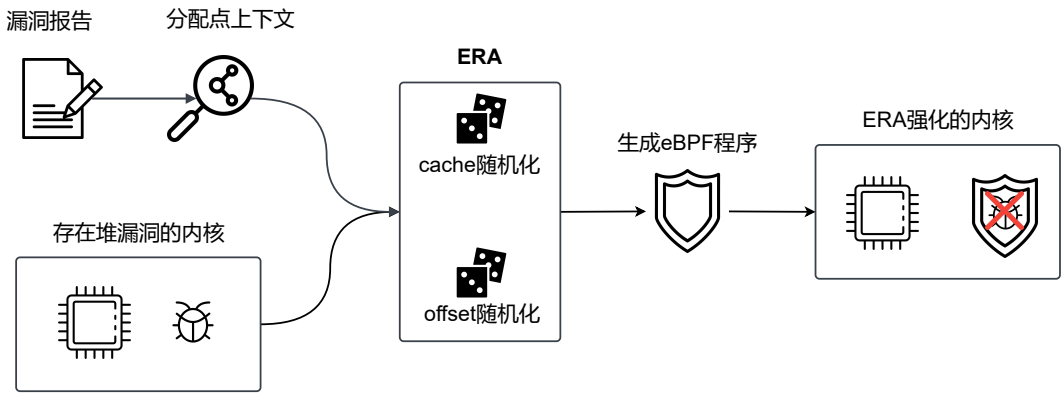


图 3-3　ERA 工作流程

关闭，但上下文不易获取。首先漏洞报告是对内核漏洞发生位置的描述，因此常见的漏洞报告形式包括 (1) 自然语言描述、(2) 内核的崩溃信息、(3) 复现 bug 的 Proof-of-concept 程序、(4) 等待审核的漏洞修复补丁、(5)SYZBOT fuzzing 工具的报告[153]。报告中通常能够确定存在漏洞的数据对象，但除 fuzzing 报告外，一般不包含该对象在何处分配。其次分配点上下文的地址随着内核地址空间布局随机化机制 (KASLR) 在每次启动时不断变化，而且发行版内核携带调试信息有限，调试信息也存在大量 bug，ERA 也很难根据调试信息准确定位分配点上下文。为保证随机化机制及时开启和关闭，分配点上下文的定位需要尽量避免地址变化的干扰。

挑战 2：数据对象空间随机化方案。堆漏洞动态缓解框架的核心在于数据对象的空间布局随机化，§3.2.3 中被绕过的内核缓解机制也采用了随机化方案，然而现有方案仅作用于类型明确的数据对象，而对于无类型的缓冲区和可伸缩数据对象则无能为力。因此新的随机化方案应该适应所有类型的数据对象，并提供更充足的随机性避免被绕过。

挑战 3：基于 eBPF 技术的随机化数据对象动态注入。 eBPF 技术不提供动态分配释放内存和修改内核数据的功能，而且还为此设计了一系列验证器避免用户进行相关操作，因为其设计初衷是在维护内核稳定运行的前提下提供强大的观测能力。但为了将数据对象空间随机化引入内核，ERA 不得不额外实现了两个 eBPF 辅助函数[6]为 eBPF 程序增加了内存管理功能，并充分利用了 eBPF 的验证器功能，用 eBPF 程序实现了空间随机化算法[48-50]来分配安全内存。同样的，ERA 使用了 eBPF 的 debug 子系统将空间随机化的内存注入内核，避免了对内核增加额外开销或引入漏洞。

下面详细研究和分析三个挑战的解决思路。

3.3.1 分配点上下文定位

ERA 需要使用 eBPF 探测分配点上下文，决定数据对象空间随机化的开启和关闭，但分配点上下文不易获取，为了保证分配点上下文的准确定位，ERA 采用静态分析定位分配函数的调用点 (caller)[154]，再结合 eBPF 的 BTF(bpf type format) 格式探测信息的 CO-RE(compile-once run-everywhere) 特性，将整个调用点所在的函数作为分配点上下文，规避 KASLR 带来的地址变化和不准确的调试信息[155]。

ERA 采用基于 LLVM-IR 的静态分析工具来定位分配点调用者函数。漏洞数据对象分配的标志是调用内存分配函数 (`kmalloc`, `sock_alloc`, `bio_kmalloc` 等)，因此 ERA 静态分析工具首先搜索 kmalloc 等分配函数的调用点作为备选结果[156]。

而分配的结果主要有如代码 3-3 所示两种情况，(1) 有类型 struct 分配点 (2 行), (2) 无类型缓冲区分配点，但结果保存在另一 struct 成员 (12 行)。而 LLVM-IR 携带了充足的类型信息，故我们通过分析分配结果的类型变化 (bitcast IR 指令) 和保存位置 (store IR 指令目标地址类型) 确定漏洞对象分配点的调用者 (caller)。

```
1  // 有类型struct分配点，直接分配
2  struct seq_operations *op = kmalloc(sizeof( *op),
     ↪ GFP_KERNEL_ACCOUNT);
3  %4 = call fastcc i8* @kmalloc(i64 32, i32 4197568) #15
4  %5 = bitcast i8* %4 to %struct.seq_operations*
5
6  // 无类型缓冲区分配点，且分配结果为另一struct成员
7  struct legacy_fs_context *ctx;
8  struct legacy_fs_context {
9     char *legacy_data;
10    ...
11 };
12 ctx->legacy_data = kmalloc(PAGE_SIZE, GFP_KERNEL);
13 %89 = getelementptr inbounds %struct.legacy_fs_context,
14                 %struct.legacy_fs_context* %5, i32 0, i32 0
15
16 %93 = call fastcc i8* @kmalloc(i64 4096, i32 3264) #12
```

```
17 store i8* %93, i8** %89, align 8
```

代码 3-3　两种代表性的数据对象分配点代码片段及 IR 表示

在有类型 struct 分配 IR 中 (3-4 行)，%4 临时变量保存了 `kmalloc` 分配 32 字节内存的返回地址，之后 `i8*` 类型的%4 临时变量被 `bitcast` 指令转换为 `struct seq_operations*` 类型指针，因此可以判断在当前函数 `seq_operations` 类型数据对象被分配；同理，在无类型的缓冲区分配 IR(12-14 行)，%89 临时变量保存了 `legacy_fs_context` 数据对象第 1 个成员 `legacy_data` 的地址，内存分配后地址被保存在%89 指向的内存中，由此可以判断在当前函数分配了 `legacy_fs_context` 的缓冲区成员的数据对象。

ERA 结合静态分析和 BTF 文件实现了数据对象空间随机化的及时开启和关闭，ERA 在分配点上下文，即分配点的调用者函数开始执行时，标记当前进程开启了数据对象空间随机化，在内存分配的核心函数 `kmalloc` 检测当前进程是否开启随机化，如开启则返回随机化结果，否则跳过随机化过程返回正常结果，在函数结束时取消对当前进程的标记，关闭随机化。如代码 3-4 所示，`struct seq_operations` 携带了 4 个代码指针，经常作为 kmalloc-32 cache 泄露内核代码地址的攻击负载，`seq_operation` 的分配点上下文为 `single_open` 函数，ERA 在 `single_open` 函数开始执行时加入探测点 `SEC('kprobe/single_open')` 开启随机化，结束时插入 `SEC('kretprobe/single_open')` 进行关闭。

```
1 // 标记 pid 进程，ERA开启随机化
2 int single_open(...)
3 {
4   struct seq_operations *op = kmalloc(sizeof( *op),
      ↪ GFP_KERNEL_ACCOUNT);
5       // 检查 pid 标记，ERA实施数据对象空间随机化
6       ...
7   return res;
8 // 取消 pid 进程标记，ERA结束随机化
9 }
```

代码 3-4　ERA 数据对象空间随机化开启关闭

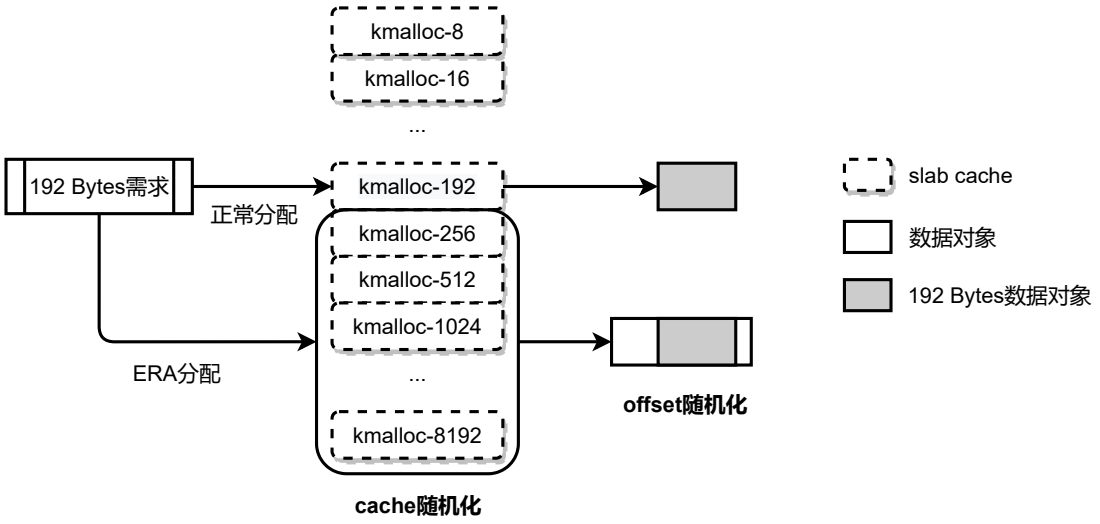


图 3-4 数据对象空间布局随机化模型

3.3.2 数据对象空间随机化

空间随机化的基本原理之一是额外分配超过需求的内存[48,157]，利用多分配的内存进行随机化，从而避免攻击者获取准确的数据对象在地址空间中的位置，防止恶意利用发生。类似现有的安全分配器机制，但是不对所有数据对象进行随机化，避免夸张的性能和内存开销。ERA 在现有的 slab 分配器的基础上设计数据对象的空间随机化，而非另起炉灶。根据空间随机化的基本原理，我们提出了两种随机化方案，如图 3-4 所示，

- slab cache 随机化：随机分配选择空间更大的 slab cache，让漏洞数据对象不在原本的 slab cache 中。
- offset 随机化：在分配的大块内存中再进行一次随机化，让漏洞数据对象的起始点不在这块内存开始的位置 (由于 CPU 的 ALIGNMENT CHECK，起始点需要 8-byte 对齐)。

ERA 的数据对象空间随机化方案提供了充足的安全保障。在未进行空间随机化的系统中，内核根据请求在对应大小的 slab cache 分配空闲数据对象，返回地址为该数据对象的起始位置。而 ERA 在每次分配时随机选择更大的 slab cache 中分配空闲数据对象，并在数据对象内的空闲空间中再进行一次随机化，使得地址无法预测。二者结合的随机化范围根据请求内存的大小存在 512 至 2070 种可能的分配为止。

值得注意的是，在开启 ERA 后，即使攻击者使用堆喷射攻击，也无从知晓

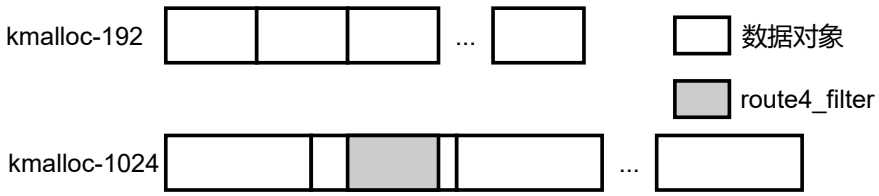


图 3-5　CVE-2021-3715 漏洞对象空间随机化

攻击程序控制的某个漏洞数据对象所在的具体 cache，因此虽然可能出现可选 offset 随机化数量较少的情况，例如 kmalloc-8 在随机化后选择 kmalloc-16 只有 1 个 offset 可选项，但是攻击者无法在大量喷射的数据对象中准确锁定 kmalloc-16 上的漏洞数据对象，因此 ERA 的随机化安全性体现为可选 cache 及其对应可选 offset 数量的叠加，例如 kmalloc-8 的安全性体现为 12 个可选 cache 中的 2070 个可选 offset。

对比相关工作中的缓解机制，在安全性上，数据对象空间随机化不存在被确定性绕过的攻击途径，即使攻击者使用堆喷射、堆风水大量分配数据对象，甚至最新型的 cross-cache 替换整个 slab cache 页，也无法准确放置攻击负载达成攻击目标；在功能上，数据对象空间随机化机制能够同时应用于长度确定的 struct 对象和长度不确定的可伸缩对象、缓冲区对象，能够覆盖溢出和释放后使用多种情况。由此可见随机化既能够用于被漏洞报告确定的漏洞数据对象，还支持部署在常见的攻击负载对象预防攻击的发生。

本章节继续以 §3.2.2 中 CVE-2021-3715 释放后使用漏洞为例，论证 ERA 数据对象空间随机化提供了充足的安全保证。攻击者在 kmalloc-192 cache 中大量分配 `route4_filter` 数据对象，并在释放后将原地址替换为相同大小的 `msg_msg` 数据对象。而在部署数据对象空间随机化机制后，原本一定在 kmalloc-192 cache 中分配的数据对象可能出现在 kmalloc-256 到 kmalloc-8192 的任意 cache 中。如图 3-5 所示，`route4_filter` 被分配在 kmalloc-1024 中，且不在该 cache 管理的数据对象起始地址，不易被攻击者预测，而且由于悬空指针指向的地址实际为随机化后的地址，该数据对象的起始地址在第一次释放后即从 ERA 机制的哈希表中移除，如果再次使用 `kfree` 函数触发释放后使用攻击，极大概率会导致内核直接崩溃，因为释放地址无法恢复到非随机化地址。

假设攻击者为了突破 ERA 使用堆喷射技术请求了大量 `route4_filter` 对象，随机化后的数据对象布满 kmalloc-256 到 kmalloc-8192 的 6 个 cache 中，同时攻

击者利用 msg_msg 数据对象可变长的特点 (64-4096 字节) 占据 route4_filter 释放之后的空间。但 offset 随机化保证了 route4_filter 的悬空指针指向的地址很难与 msg_msg 构造的攻击负载完全重叠，只有当 route4_filter 的悬空指针正好位于数据对象的起始地址，即 offset 随机化为 0 时才能够成功实现攻击，否则通过 kfree 解引用悬空指针极有可能造成内核崩溃。

我们估算了在 ERA 机制仅随机化 route4_filter 一类数据对象时，利用堆喷射、堆风水等技术攻击的绕过 ERA 所需的分配数据对象的数量和成功触发几率。尤其要注意的是，kfree 函数释放非 slab 对象的起始地址时会极大概率导致内核直接崩溃，因此在此漏洞中释放后使用漏洞仅能够触发一次。假设在 8 核 16GB 硬件的计算机上，锁定攻击程序只运行在 1 个 cpu 上，只使用 slab 的一组 per-cpu cache。攻击程序需要首先采用堆风水攻击填满 per-cpu 局部的半满和全满 cache 页，才能让 cache 从 buddy 分配器继续获得新的内存页，使用堆喷射攻击填满该内存页绕过 freelist 随机化，乐观估计 ERA 未部署时单个 cache 达到攻击条件需要至少 126 个 route4_filter 对象 (6 页内存)，如果考虑到 ERA 的 cache 随机化，route4_filter 对象会出现在 6 个 cache 中，那么攻击程序至少需要连续分配 126 * 6 = 756 次，才能勉强保证每个 cache 上都有 route4_filter。如果再考虑到 ERA 的 offset 随机化，route4_filter 在 kmalloc-256 到 kmalloc-8192 可能出现的位置有 1872 个，想要保证 route4_filter 刚好出现在数据对象的起始位置，乐观估计可能也需要将分配次数扩大 3 个数量级达到 75 万次分配。然而攻击者只能从 75 万悬空指针中选择一个触发释放后使用漏洞，可见被利用的风险已经被极大降低，如果 msg_msg 数据对象也部署 ERA 机制，那么攻击风险将会被进一步缓解。

值得思考的是对于 CVE-2021-3715，如果 ERA 的 offset 随机化直接避免了 route4_filter 对象出现在 offset 为 0 的位置，那么使用 kfree 触发悬空指针类型的释放后使用漏洞就能被避免。然而此种思路降低了 offset 随机化的随机熵，而且 ERA 还支持 cache 随机化，已经具备较强的随机性，因此我们选择保留 offset 随机化为 0 的可能性。此外，在其他同类型漏洞中，如果通过溢出或修改悬空指针指向内存可能尝试次数稍多，但指针被破坏仍然容易导致内核崩溃；但如果仅仅读悬空指针指向内存威胁性相对较小。

3.3.3 基于 eBPF 的动态注入

eBPF 技术的设计初衷是在不影响内核稳定运行的情况下提供强大的观测能力，因此 eBPF 并不具备内存分配释放和内核数据修改功能，同时还设计了一系列验证器防止用户编写相关 eBPF 程序。但是 ERA 需要在尽可能不破坏这一设计规范的前提下，将空间随机化方案和随机化后的数据对象注入内核，缓解安全威胁。

空间随机化的 eBPF 实现

由于 eBPF 机制不具备动态分配和释放内存的功能，因此 ERA 在内核中增加了额外的辅助函数 (helper function)，将内核的分配函数和释放函数导出给 eBPF，使其依旧能够支持分配参数 (`GFP_FLAG`)。为了避免探测嵌套，导出的函数被设置禁止探测。同时动态缓解机制充分利用了 eBPF 的内核态验证器功能，主要使用 eBPF 程序实现了空间随机化模型，对内核代码的修改量非常小约 100loc，几乎不会引入额外的安全漏洞。

eBPF 动态缓解程序在执行到分配点上下文时开启空间随机化，将 cache 和 offset 随机化之后的地址返回给分配点上下文，并将该地址和 cache 随机化的数据对象起始地址记录在 eBPF 的哈希表中；当释放时，动态缓解程序首先检测释放地址是否属于随机地址，如果属于，就从 eBPF 的哈希表中找到 cache 随机化后的数据对象起始地址并合法释放，再将记录移出哈希表。eBPF 的哈希表是线程安全的，构成其基础的 bucket 使用自旋锁保护，而 bucket 里面的成员使用 RCU 机制读写。

随机化数据对象注入

ERA 借助了内核调试使用的错误注入机制 (`error_injection`) 来对内核函数返回值进行修改。通过在编译时标记 `ALLOW_ERROR_INJECTION` 宏将函数加入错误注入白名单，用 eBPF 探测该函数起始点，不执行函数直接返回指定的结果。错误注入已经作为 eBPF 的子系统之一加入内核，只需在编译时额外开启 `CONFIG_FUNCTION_INJECTION` 等几个编译选项即可安全使用 eBPF 提供的 `bpf_override_return` 辅助函数实现相关功能。

3.4 实验评估

本章节设计了 4 组实验分别对 ERA 的有效性、性能开销、相关工作对比和易用性进行验证。首先为证明 ERA 的有效性，我们定性分析了 ERA 方案的安全性，并从公开渠道收集并评估了 40 个内核堆漏洞，同时进一步测试了其中 12 个绕过现有缓解机制的攻击程序 (EXP)；其次为了体现 ERA 的具体开销，我们通过采样内核运行过程，选取了其中分配量较大的 4 类数据对象，分别测试了基准测试程序的性能和内存开销，以验证即使在严苛条件下，ERA 仍能够表现出较好的性能和空间消耗；此外，我们对比了同样采用空间随机化技术的数据对象成员随机化机制，体现了 ERA 在随机性、适用范围和性能上的优势；最后，我们通过分析一位系统管理员需要付出多大努力才能使用 ERA 缓解内核堆漏洞，展示 ERA 工具易于部署的特性。测试过程中 ERA 的 cache 和 offset 随机化全部开启。

3.4.1 有效性

实验设置

本文从 NVD、syzkaller、github 等公开渠道[1]收集了 2010 年-2022 年的 40 个内核堆漏洞，漏洞类型覆盖了溢出 (OOB) 和释放后使用 (UAF) 两类代表性的内核堆内存损坏，并涵盖了 slab、buddy 和 vmalloc 三类内核堆分配器。在此基础上收集了其中 12 个内核堆漏洞的公开 EXP 程序，这些程序为确保攻击的稳定性，均采用了堆喷射、堆风水的攻击技术绕过 §3.2.2 中内核堆漏洞缓解机制。

首先，我们对 ERA 缓解溢出、释放后使用、双重释放和非法释放四种最常见的堆漏洞的能力和 ERA 框架自身的安全性进行定性分析，并根据数据对象空间随机化的原理归纳出 ERA 缓解方案适用的两个条件：(1) 由 slab 分配器分配，(2) 漏洞的攻击方式是读写攻击者控制的其他数据对象，并以这两个条件作为标准评估收集的 40 个内核堆漏洞，如果满足标准则说明该漏洞能够被 ERA 机制缓解。

其次，我们将选取的 12 个内核堆漏洞全部导入到 v5.15 版本内核，使用 qemu-kvm 虚拟机构建测试环境，测试环境符合 §3.2.1 威胁模型假设，分别执行漏洞对应的 EXP 攻击程序，测试是否能够在满足威胁模型的前提下，同时绕过

现有缓解机制和 ERA 缓解机制，成功实现攻击负载准确放置，进而泄露内核代码地址、提升程序执行权限。我们认为由于运行时的数据对象随机化程度不同，在攻击发生后如果系统维持运行、提示错误或崩溃，而攻击目标未能达成，那么 ERA 缓解程序对此漏洞是有效的。

实验结果

CVE ID	类型	分配器	有效性
CVE-2010-2959	溢出	slab	●
CVE-2017-7184	溢出	slab	●
CVE-2022-0185	溢出	slab	●
CVE-2022-34918	溢出	slab	●
CVE-2016-6187	溢出	slab	●
CVE-2021-22555	溢出	slab	●
CVE-2021-43276	溢出	slab	●
CVE-2017-1000112	溢出	slab	○
CVE-2021-27365	溢出	slab	●
CVE-2017-7308	溢出	page	◐
CVE-2022-27666	溢出	page	◐
CVE-2020-14386	溢出	vmalloc	◐
CVE-2017-8824	释放后使用	slab	●
CVE-2020-16119	释放后使用	slab	●
CVE-2021-23134	释放后使用	slab	●
CVE-2022-2586	释放后使用	slab	●
CVE-2021-3715	释放后使用	slab	●
CVE-2021-4154	释放后使用	slab	●
CVE-2019-18683	释放后使用	slab	●
CVE-2022-2588	释放后使用	slab	●

CVE ID	类型	分配器	有效性
CVE-2021-33909	释放后使用	slab	●
CVE-2017-7533	释放后使用	slab	●
CVE-2016-8655	释放后使用	slab	●
CVE-2021-26708	释放后使用	slab	●
CVE-2017-15649	释放后使用	slab	●
CVE-2021-20226	释放后使用	slab	●
CVE-2021-27365	释放后使用	slab	●
CVE-2021-22600	释放后使用	slab	●
CVE-2022-1786	释放后使用	slab	●
CVE-2022-2602	释放后使用	slab	●
CVE-2017-11176	释放后使用	slab	●
CVE-2022-1116	释放后使用	slab	●
CVE-2022-29581	释放后使用	slab	●
CVE-2022-25220	释放后使用	slab	●
CVE-2020-14356	释放后使用	slab	●
CVE-2022-29582	释放后使用	slab	●
CVE-2017-10661	释放后使用	slab	●
CVE-2016-10150	释放后使用	slab	●
SYZBOT-1e2ff6d	释放后使用	slab	●
SYZBOT-ea6a322	释放后使用	slab	●

表 3-2　内核堆漏洞评估结果，● ERA 有效；◐ 表示 ERA 原理有效，等待未来实现；○ 表示 ERA 无效

对于溢出 (out-of-bound) 漏洞：ERA 方案通过数据对象的空间随机化，让攻击者无法确定漏洞数据对象和受害者数据对象到底在哪个 slab cache 中，§3.2.2 溢出漏洞模型中的 ptr 和指向的 obj 均存在随机性，因此无法将被攻击对象正好放在漏洞数据对象后面，如果按照 $offset_{ptr} + size_{access}$ 进行溢出，则无法预测溢出结果，因此无法进行攻击。

对于释放后使用 (use-after-free) 漏洞：ERA 方案能够成功地随机化分配漏洞数据对象在内核地址空间的位置，§3.2.2 释放后使用模型中的悬空指针 p 的位置存在随机性，同理 p 指向的 O 的位置 $\forall m, size : [m, m + size)$ 很难被攻击者准确定位，因此构造的攻击负载无从放置，即使触发悬空指针 p 指向的地址，也很难引起负载中引入的下一步攻击。

对于双重释放 (double-free) 漏洞：内核的 slab 机制本身提供了 naive check，使得这类漏洞无法通过连续释放相同地址控制 slab 的 freelist，即使攻击者可能将其转化 kfree 函数释放后使用漏洞，也无法绕过 ERA 实现攻击利用。

对于非法释放 (invalid free) 漏洞：内核 slab 机制在释放时检查地址是否由 slab 分配，因此这类漏洞在内核中无法利用。

此外，攻击者对 ERA 框架的攻击尝试也无法生效，首先 ERA 载入内核的代码通过了验证器检查，并设置为只读，因此不存在代码完整性问题；其次 ERA 使用的数据和空间随机化分配的数据天然具备了一定随机性，攻击者也很难通过漏洞利用找到 eBPF 使用的数据位置；最后，eBPF 程序的运行和载入需要 root 权限，攻击者无法注入恶意的 eBPF 程序破坏系统安全。

内核堆漏洞的评估结果如表 3-2 所示，在公开收集的 40 个漏洞中，36 个漏洞能够被 ERA 缓解机制成功缓解，这些堆漏洞均由 slab 分配器分配，且攻击方式为读写攻击者控制的其他数据对象，ERA 采用空间随机化技术使得这些漏洞数据对象的分配点无从预测，攻击者无法准确放置或解引用负载，因此内核堆漏洞被利用的风险显著降低。

CVE 编号	类型	威胁对象	分配上下文	有效性
CVE-2010-2959	溢出	缓冲区	bcm_sendmsg	●
		shmid_kernel	newseg	●
CVE-2017-7533	溢出	缓冲区	inotify_handle_event	●
		iovec	iovec_from_user	●
CVE-2021-22555	溢出	xt_table_info	xt_alloc_table_info	●
		pipe_buffer	alloc_pipe_info	●
CVE-2022-34918	溢出	缓冲区	nft_set_elem_init	●
		simple_xattr	simple_xattr_alloc	●
CVE-2017-7184	溢出	xfrm_replay_state_esn	xfrm_alloc_replay_state_esn	●
		cred	cred_alloc_blank	●
CVE-2016-8655	释放后使用	packet_sock	packet_create	●
		user_key_payload	user_prepare	●
CVE-2021-26708	释放后使用	vsock_transport	vmci_transport_socket_init	●
		msg_msg	load_msg	●
CVE-2020-16119	释放后使用	sock	sk_alloc	●
		msg_msg	load_msg	●
CVE-2017-10661	释放后使用	timerfd_ctx	__x64_sys_timerfd_create	●
		msg_msg	load_msg	●
CVE-2016-10150	释放后使用	kvm_device	kvm_vm_ioctl	●
		key	key_alloc	●
CVE-2017-11176	释放后使用	netlink_sock	__netlink_create	●
		缓冲区	____sys_sendmsg	●
CVE-2017-15649	释放后使用	packet_sock	packet_create	●
		msg_msg	load_msg	●

表 3-3 面对真实 CVE 漏洞的动态缓解技术有效性 (EXP 均能绕过现有缓解机制)，● 表示 ERA 有效

无法应用的情况包括 CVE-2017-7308、CVE-2022-27666 和 CVE-2020-14386，此处三个漏洞的漏洞数据对象由 buddy 或 vmalloc 分配器分配，而非 slab 分配器，但是数据对象空间随机化的思想对于这两类分配器依旧有效，我们将在未来的工作中将 ERA 机制部署在这两类分配器上。CVE-2017-1000112，则属于特殊情况，溢出发生在数据对象内部。该漏洞中的漏洞数据对象 `sk_buff` 虽然也

由 slab 分配器分配，但分配长度远大于 `sk_buff` 成员所需长度，在额外分配的长度中，`struct sk_shared_info` 数据结构位于数据对象的结尾，`sk_buff` 成员的结尾到 `sk_shared_buff` 数据结构开始的区域为保存 `sk_buff` 信息的缓冲区，该缓冲区可能发生溢出，破坏位于数据对象结尾处的 `sk_shared_info` 成员，造成数据对象内溢出。攻击时没有破坏其他攻击者控制的数据对象，因此 ERA 无法缓解此类漏洞。虽然 ERA 和现有缓解机制均无法直接应对上述两种情况，但对于第一类 buddy 或 vmalloc 分配器分配漏洞数据对象情况，ERA 可以间接随机化攻击过程中用到的受害者/喷射数据对象，由于这类对象可选范围通常较固定且符合 ERA 适用范围，故 ERA 依旧能够缓解安全风险。12 个漏洞的 EXP 攻击测试结果如表 3-3 所示，本文分析了 EXP 攻击中漏洞数据对象的类型和攻击负载的数据对象类型，并利用 ERA 的分配点上下文定位技术确定了 ERA 部署位置。ERA 机制启动了漏洞数据对象和攻击者使用的攻击负载数据对象的空间随机化，测试结果显示均能够成功缓解，攻击目标无法达成，即使采用堆喷射、堆风水等能够稳定绕过现有缓解机制的 EXP 程序仍无法绕过 ERA 的数据对象空间随机化，进一步说明了 ERA 的有效性。

3.4.2 性能和内存开销

实验设置

性能和内存开销实验在 Intel core i5 12600K，内存 16GB DDR4，1TB NVMe 固态硬盘，内核版本 v5.15 物理机上进行。由于漏洞的触发路径并非实际内核的常用执行路径[13]，常规的内存密集型测试实际上几乎不触发 ERA 的空间随机化机制，无法测试出 ERA 机制的额外开销，因此我们对内核的运行过程中的数据对象分配数量采样，随机选取一定时间内分配较为频繁的 4 类数据对象部署 ERA 空间随机化机制，体现严苛条件下 ERA 的性能损耗。采样工具选取 eBPF 的命令行工具 bpftrace，执行 `bpftrace -e 'tracepoint:kmem:kmalloc @[kstack()]=count(); '`，通过统计内存分配函数的调用栈数量，选取测试的数据对象。

性能开销重复多次取平均值，我们首先选取了 lmbench 微基准测试程序 (micro benchmark)[158]进行测试，测试范围涵盖系统调用上下文切换、文件系统、本地通信延迟和带宽四类基本功能，测试了随机选取的四类数据对象分别部署的

开销和同时部署的开销，判断单个数据对象和多个数据对象累加对系统性能造成的影响。同时我们使用 phoronix-test-suites 宏基准测试程序 (macro benchmark)[159]集合，测试了 ERA 在具体应用程序下的表现，测试包含 PostMark、OSBench、IPC_benchmark、HackBench、OpenSSL、BenchmarkMutex、PyBench 和 Apache HTTP Server，该测试主要针对全部四类数据对象累加的性能开销。

由于内存开销随系统启动时间和正在执行的程序不断发生波动，而且由于系统中断、网络通信等功能的偶然性，很难像性能开销测试一样横向对比，因此我们聚焦于内存占用的增量而非具体内存开销，即采样的每组内存占用量减去采样过程中的最小内存占用量。表明了内核在执行测试任务的内存分配和释放情况。我们将内存增量的差值作为对比标准，用于比较 ERA 开启后的额外内存开销，同时注重分析差值的最大值，即 ERA 开启后的最大额外内存开销。我们编写了系统内存采样程序，每秒采集一次当前系统的内存使用情况，分别测试了执行 lmbench 和 chrome 浏览器在线播放相同视频 5 分钟两项指标，其中后者反映了 ERA 应对复杂任务时的表现，同时更好控制采样时间变量，主要测试了全部四类对象累加的内存开销。

最后，在工作日开启 ERA 空间随机化 24 小时，期间使用者正常进行日常研究工作，用于测试 ERA 长期工作给系统和使用者带来的影响。

实验结果

根据采样结果，本文选取了两类长度确定的数据对象 `kernfs_open_file`(缩写为 kof) 和 `seq_operations`(缩写为 so)，还选择了两类长度不确定的缓冲区数据对象，分别由 `load_elf_phdrs`(缩写为 lep) 函数和 `inotify_handle_inode_event`(缩写为 ihie) 函数分配。以上对象在采样过程中均分配 2000 次以上。

性能测试的结果均以未开启 ERA 内核的测试结果进行标准化，大于 0 则表示 ERA 性能开销大于未开启 ERA 内核，小于则开销好于未开启 ERA 内核。lmbench 基准测试程序的性能损耗如表 3-4 所示，单项测试的最坏结果约造成 3% 的额外开销，同一数据对象性能损耗的平均性能开销仅为约 1%。而且单独的数据对象和全部数据对象累加的性能开销几乎没有显著变化。宏基准测试程序的各类性能也仅仅导致了微不足道的约 1% 的性能开销，如表 3-5 所示。

导致 ERA 性能表现优秀的原因主要有两点，一是漏洞数据对象的分配和释

	kof	so	lep	ihie	All	POLAR
Simple syscall	2.23%	2.37%	2.30%	1.06%	2.08%	-2.15%
Simple read	-0.48%	-0.83%	3.04%	-0.36%	-0.36%	-0.48%
Simple write	0.56%	0.23%	0.75%	0.19%	0.47%	2.20%
Select on 100 fd's	-0.15%	0.07%	-0.17%	-0.07%	-0.12%	3.01%
Signal install	1.72%	-0.44%	1.68%	0.60%	0.56%	-0.56%
Signal overhead	0.10%	0.31%	0.92%	0.38%	0.74%	3.75%
UDP latency	0.54%	1.40%	-0.54%	-1.47%	0.15%	-1.24%
AF_UNIX bandwidth	0.92%	1.07%	0.94%	2.32%	0.32%	-0.50%
Pipe bandwidth	2.05%	2.41%	1.22%	1.73%	1.22%	0.19%
Average	0.83%	0.73%	1.13%	0.49%	0.56%	0.47%

表 3-4 LMBench 性能开销，结果与原生内核结果进行标准化

	All	kof+so	POLAR
PostMark	0.00%	0.00%	-2.60%
OSBench	-3.59%	-1.55%	-2.38%
IPC_benchmark	-1.73%	1.74%	1.69%
Hackbench	0.07%	6.89%	7.08%
OpenSSL	0.22%	-0.02%	-3.70%
BenchmarkMutex	0.39%	0.31%	-0.29%
PyBench	0.00%	0.18%	0.36%
Apache Server	0.58%	1.01%	-1.00%
Average	-0.51%	1.07%	-0.11%

表 3-5 Phoronix 性能开销，结果与原生内核结果进行标准化

放，即使在大量分配的情况下仍然在内核的整条执行路径上占比极低，因此原本的开销较低。其二是，ERA 基于 eBPF 插入的探测点仅有分配点上下文开启、结束、slab 分配函数和释放函数四处，其中前两处探测点执行数量较少，而且负责开启关闭当前进程的随机化，指令较少；而后两者在所有调用分配和释放函数时都会触发，执行次数较多，但首先 eBPF 插入探测点使用高效的 ftrace 函数跳转机制而非 int3 中断跳转；其次在跳转后第一步采用的是 eBPF 的哈希表查找，查找 key 为 32 位长度的进程 pid 或 64 位的随机化地址，时间复杂度较低；最后随机化设计比较巧妙，随机化仅发生在查找哈希表存在匹配项的条件下，cache 和 offset 随机化机制仅需要获取一个随机数，而且原有的内存分配逻辑被跳过，无需额外分配，因此开销同样非常低。同理，增加保护对象的数量并没有显著放大 ERA 在内核执行路径上面的占比，故性能损耗几乎没有增加。

在部分测试指标中，系统在开启缓解机制后获得了更好的性能表现，例如 ihie 的 UDP latency 性能好于未开启 ERA 内核。类似情况在内核性能测量工作中也有出现[160]。究其原因可能是由于 CPU 采用了大小核调度、变频架构或 cache 命中率提升，测试时 CPU 的工作状态和系统后台程序运行的噪声干扰了测试结果[161]。

内存开销增量结果如图 3-6 所示，图中横坐标为执行相同任务的内存采样点，纵坐标表示内存占用的增量。其中 lmbench 执行和 chrome 浏览器在线播放视频的内存占用增量和其走势基本吻合，相同采样点 ERA 开启和未开启时内存增量的差值不大，说明内存使用情况基本一致，因为测试时执行的任务相同。而在两组测试中 slab 内存占用情况走势基本吻合，但测试过程中相同采样点 ERA 开启和未开启时内存增量的差值较为明显，说明了 ERA 开启时 slab 内存消耗略高于未开启时，经过计算差值为 5-150MB，最高的额外内存损耗占比 0.9%，该现象符合 ERA 的数据对象空间随机化原理。需要额外强调的是测试所选四类对象在内核中大量分配 (远高于真实漏洞情况[13])，同时 slab 分配器占用的内存并未全部分配，而是作为缓存由 slab cache 管理，数据对象生命周期较短，因此 ERA 在严苛的测试环境下开销能够被接受，且 slab 内存的循环使用不会对系统带来过大压力。

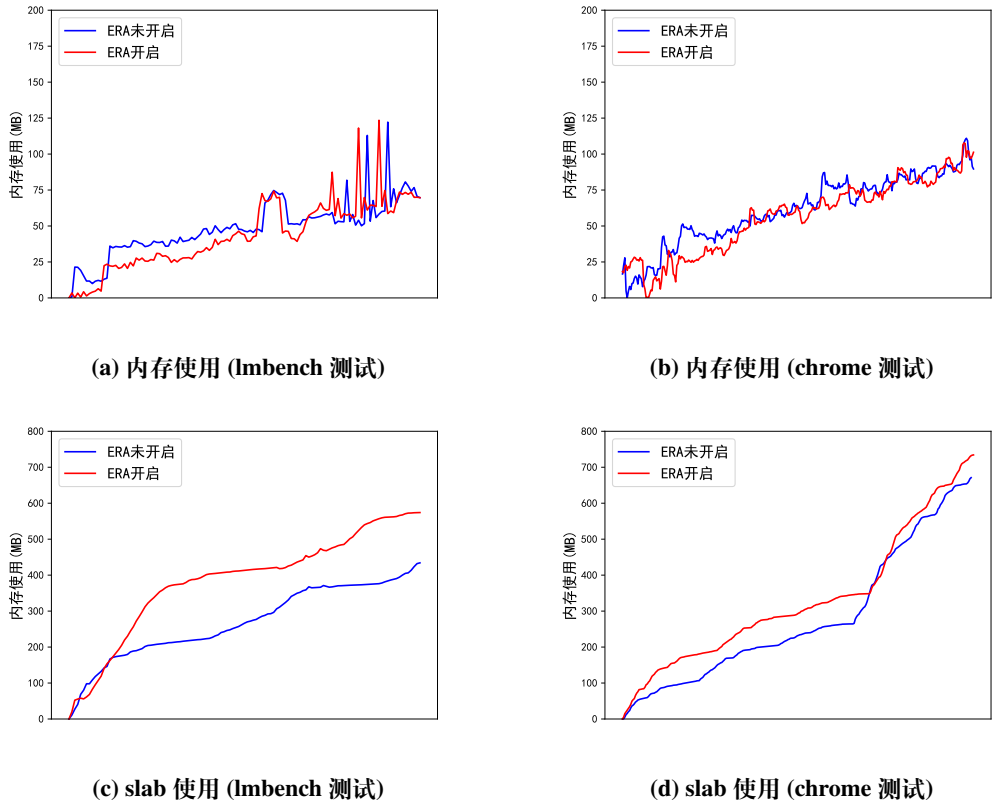
在工作日 24 小时开启 ERA 的测试中，系统始终保持正常运行，而系统使用者很难感知 ERA 带来的影响，因此 ERA 能够被应用在研究和生产环境下有效缓解内核堆漏洞风险。

3.4.3 相关工作对比

实验设置

我们选择了同样采用空间随机化技术的数据对象成员布局随机化 (randstruct) 及其后续改进 SALAD 和 POLAR，对比 ERA 设计的 cache 和 offset 两类数据对象空间随机化。成员随机化的基本原理在于打乱数据对象内成员的布局，使得攻击者无法准确放置攻击负载，不能准确溢出或解引用数据对象的安全敏感成员，从而提升攻击利用难度。然而目前内核中的成员随机化 (randstruct) 仅在编译时通过编译器扩展随机化一次，而且为了支持可扩展模块载入，随机种子公开可见，攻击者可以轻易得到安全敏感成员的准确偏移量。为了提升成员随机化的机密性，相关工作 SALAD 和 POLAR 在随机化频率上进行优化，前者在固定周期更换数据对象的随机种子，后者则在每次分配时重新随机化数据对象布局，不过都需要将取数据对象成员取地址指令替换为专用的取地址函数，增加了额外开销。

由于成员随机化不具备动态增加保护对象的能力，故无法对类型多变的漏



(a) 内存使用 (lmbench 测试)　(b) 内存使用 (chrome 测试)

(c) slab 使用 (lmbench 测试)　(d) slab 使用 (chrome 测试)

图 3-6　lmbench 和 chrome 播放在线视频的系统内存使用、slab 占用增量

洞数据对象及时提供保护，与 ERA 工作的安全背景和目标不同，安全性远弱于 ERA 方案，因此本章节仅在原理层面上对随机性、随机频率、功能、性能和内存消耗上对比 ERA 和成员随机化，首先设置了一个度量指标表示随机化强度，即一个 8 字节成员能够出现在分配的数据对象中的位置数量，代表攻击者想要读写指定安全敏感指针的难度，数量越多则代表随机性越强。其次，为了验证性能开销，我们采用 LLVM IR Transform Pass[162]和 Fisher & Yates 洗牌算法[163]模拟了成员随机化工作，对比了 `kernfs_open_file`(缩写为 kof) 和 `seq_operations`(缩写为 so) 两类长度和成员确定的数据对象，在 ERA 和成员随机化情况下的基准测试程序性能开销，并测量 chrome 浏览器播放在线视频 5 分钟的内存使用增量，其中成员随机化仅选择随机化频率最高的 POLAR 方案。

实验结果

随机性度量，功能和内存消耗的对比如表 3-6 所示。表中的 size 表示分配的数据对象的大小，n 表示数据对象成员的数量，entropy 表示随机化时额外增加的空间。在随机性方面，randstruct 方案的 8 字节成员仅能出现在长度为 size/8 个位

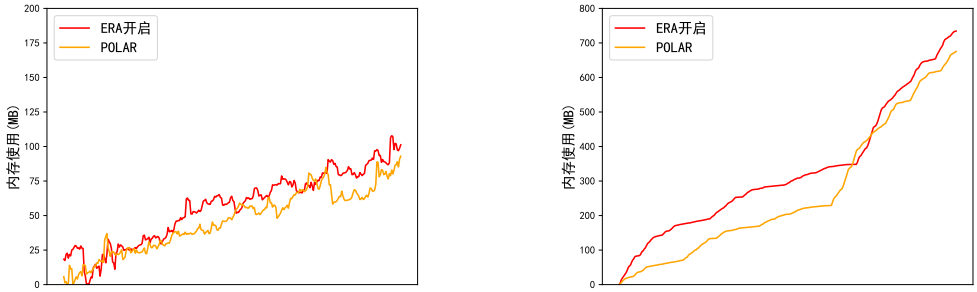
置。而 SALAD 和 POLAR 可以人为设置一个额外的 entropy 区间，一般 entropy 最大等于 size，那么这两类成员随机化方案的强度最大为 randstruct 的两倍。而 ERA 采用的空间随机化因为将分配内存放在更大空间的 cache 中，在通用的 slab cache 里，多数情况下后一个 cache 数据对象大小是前一个的两倍，例如 kmalloc-128/256，而特殊情况下为 1.33 至 1.5 倍 (kmalloc-64/96/128，kmalloc-128/192/256)，因此 ERA 空间随机化的强度多数情况下能够达到 2 倍以上优于成员随机化，最大为常数 1024 个可行位置，远超过成员随机化。而对比随机化频率，randstruct 的每次编译时随机化频率最低，其次是 SALAD 固定周期随机化，如果 SALAD 的周期调整为每次分配，则认为与 POLAR 和 ERA 随机频率一致，显然分配次数越多攻击难度越大。

在功能上，成员随机化必须依赖重新编译内核增加被保护数据对象，且仅支持确定长度、有类型的数据结构，而 ERA 则能够动态增加数据对象，且不存在数据对象类型限制。显然 ERA 在适用范围上有毋庸置疑的优势

	随机性	度量	随机频率	动态长度	运行时开销	内存开销
Rand struct	成员随机化	$(size)/8$	每次编译	否	无	无
SALAD	成员随机化	$(size+entropy)/8$	固定周期	否	周期 shuffle+ 取地址	$entropy+n*4+8$
POLAR	成员随机化	$(size+entropy)/8$	每次分配	否	每次分配 shuffle+ 取地址	$entropy+n*4+8$
ERA	空间随机化	$[(size*1.33)/8, 1024)$	每次分配	是	每次分配随机化	$[0.33*size, 8184)$

表 3-6 数据对象空间随机化与成员随机化对比

在运行时性能消耗方面，randstruct 因为仅在编译时随机化，因此性能几乎没有损耗，而 SALAD，POLAR 及 ERA 则普遍存在每次分配时随机化的开销，但不同点在于 SALAD 和 POLAR 采用 shuffle 算法需要获取多个随机数，而 ERA 虽然有 cache 和 offset 两种随机化，但仅需要获取一个随机数，此外 SALAD 和 POLAR 需要将数据对象成员的取地址指令替换为专用的取地址函数，根据成员的需要返回准确的地址，ERA 并不存在此类开销。对于额外的内存消耗，randstruct 同样没有运行时消耗，而 SALAD 和 POLAR 则需要专门的元数据保存随机化前后的映射关系，因此对于一个数据对象所需要的额外内存开销则包括 (1) 8 字节指定数据对象地址，(2) n 个成员对应的偏移量 (n*4)，(3) 额外的 entropy 区间，最大为 size。ERA 的原理是通过增加空间的分配降低地址的可预测性，因此对于内存的额外消耗量大于 SALAD 和 POLAR，至少为分配数据对象的 0.33 倍，最多为 8184 字节 (8 字节内存被随机化到 kmalloc-8192)。但根据上文结论，数据对象的生命周期较短，因此此类工作的内存能够快速循环使用，造成的内存开销



(a) 内存使用 (chrome 测试)　　**(b) slab 使用 (chrome 测试)**

图 3-7　chrome 播放在线视频的系统内存使用、slab 占用情况

可以接受。

POLAR 和 ERA 对比基准测试程序的结果如表 3-4 和表 3-5 所示，在 lmbench 测试中，各项测试指标各有高低，但平均测试结果二者几乎持平。phoronix 测试组件的测试结果也体现二者开销基本持平，但原理上 ERA 的性能开销应该明确好于 POLAR，造成二者性能基本持平的主要原因是，选取的 `kernfs_open_file` 和 `seq_operations` 数据对象成员数量较少，而且在内核中仅替换了 176 个取地址指令，且替换后取地址函数仅负责根据成员编号获取偏移量，相比复杂的内核程序，取地址函数的指令和执行次数较少，因此造成的开销几与原理上性能更好的 ERA 一致。

POLAR 和 ERA 对比 chrome 浏览器在线播放视频 5 分钟的结果如图 3-7 所示，因为测试任务相同，ERA 和 POLAR 的使用内存增量基本一致，在 slab 占用指标上 ERA 稍高于 POLAR 符合上文原理分析，但因为数据对象生命周期较短，且 slab 占用内存并未被全部分配，因此对比相关工作 ERA 内存开销劣势并不显著。

综上所述，ERA 相比同样采取空间随机化的数据对象成员随机化，在安全性上提供了更强的随机性；功能上适应范围更广；而性能损耗方面 ERA 在原理上也具备优势，同时空间随机化设计原理上存在的内存损耗在实际测试中额外开销有限，因此 ERA 采用的数据对象空间随机化具备较为显著的创新性。

3.4.4 易用性

为了证明 ERA 系统的易用性，我们具体分析使用者得到漏洞报告后使用 ERA 的流程。

首先，根据内核堆漏洞发生的位置，找到漏洞数据对象的类型仍需要 ERA 使用者手动分析，但是从源代码中找出漏洞数据对象的类型并不困难。常见的 IDE 集成开发环境都能够提示选定数据结构的类型和声明位置，即使是对于没有数据结构类型的缓冲区，通常也被其他数据结构的指针成员指向，因此使用者可以利用静态分析工具找到分配点上下文。其次，ERA 已经提供了 eBPF 程序生成的功能，使用者只需要将第一步提取的漏洞数据对象分配点上下文输入，ERA 即可输出相关的安全缓解程序。最后，使用者必须使用 root 权限启动 eBPF 程序将漏洞缓解策略载入内核中，从此刻开始攻击者将很难利用漏洞破坏内核完整性。

鉴于近三年公开的 57 个内核漏洞中有 40 个属于内核堆漏洞，ERA 具备广阔的应用前景。以 CVE-2017-7533[164]为例，漏洞报告中已经给出溢出的根源在于 `event=kmalloc(alloc_len, GFP_KENEL);` 处分配的内存不足以保存用户传递的文件名数组，那么无需进行额外分析，只需将分配点上下文，即 `inotify_handle_event` 函数交给 ERA 原型系统，生成对应 eBPF 程序并载入内核，即可实现堆漏洞的动态缓解。

3.5 讨论

ERA 采用了分配点上下文控制数据对象空间随机化机制开启和关闭，然而实际测试中我们发现这一方案还存在一定提升空间。例如 CVE-2010-2959[165]中存在两个问题，首先内核编译优化将漏洞数据对象分配点 `bcm_rx_setup` 被内联在 `bcm_sendmsg` 函数中，并且可能被优化成 `bcm_sendmsg.cold` 缩减指令数量，因此在选择探测点时需要选择上层的 `bcm_sendmsg` 函数。但是 memorizor[166]已经给出了精准的运行时指令到数据结构的关系，我们将在未来的工作中将其整合进分配点上下文定位模块中。

其次 `bcm_rx_setup` 函数中不止调用了一次内核分配函数，更何况该函数被内联到了上层函数 `bcm_sendmsg` 中，但 ERA 采用的方案会随机化 `bcm_sendmsg`

函数中所有分配的数据对象，可能会造成短时间内较多的无效额外开销，但是这也使得 ERA 拥有了远大于相关工作的随机熵，抵御未知的安全威胁，而且测试结果表明，随机化数量的增加对系统运行时性能和内存的开销均未造成太大影响，因此这些开销是可以接受的。

此外，ERA 数据对象空间随机化中的 slab cache 随机化方案最大支持 4096 字节，slab cache 中最大的 kmalloc-8192 cache 暂不支持 ERA，ERA 可以通过创建更大的 slab cache 实现对 kmalloc-8192 的支持，但根据我们在内存密集测试下的统计结果，kmalloc-8192 仅占全部分配量的 0.5%，而且此 cache 中较少存在包含安全敏感信息的对象，因此我们将在未来的工作中实现该功能。

内核堆除了文中重点提到的 slab 分配器，还有 buddy 和 vmalloc 两种分配器，例如 CVE-2017-7308[167]页缓冲区溢出，我们强调 ERA 的随机化设计思想对这两类较为少见的漏洞也有效，并在未来的工作中会将 ERA 拓展到这两类分配器中。CVE-2017-1000112 数据对象内溢出漏洞超出了 ERA 的适用范围，为实现对此类漏洞的缓解，我们将探索 eBPF 对漏洞触发更直接的监控和阻止。

3.6 小结

本章节提出了一种针对内核堆漏洞的动态缓解技术，在漏洞修复前的较长时间窗口内降低内核面对的安全威胁。动态缓解采用了数据对象空间随机化技术，通过在更大空间的 slab cache 中分配大于所需的内存，并在内存中随机放置数据对象，使得原本存在安全威胁的数据对象地址难以被攻击者预测，并在此基础上利用 eBPF 技术的特性将空间随机化的数据对象安全、高效地注入内核。

我们强调基于 eBPF 的动态缓解技术的安全性、高效性和易用性。ERA 能够动态对任意存在安全威胁的数据对象部署空间随机化，使得攻击者无法准确预测数据对象的位置或偏移量，很难放置攻击负载破坏整个系统。动态缓解机制相比原始内核仅引入约 1% 性能开销，内存开销由于快速分配释放也几乎与原始内核一致。系统管理员无需等待安全专家发布的补丁或重新编译内核即可部署缓解程序，因此我们认为本文的动态缓解技术有广阔的使用前景。

第四章　PET: 基于 eBPF 的防内核漏洞触发框架

4.1　引言

本研究对比第三章 ERA 框架，将实时防御的内核漏洞类型扩展到更大范围，在漏洞利用前阶段阻止漏洞被攻击者触发，实现了对内核漏洞的实时防御。以下将详细介绍本研究的核心原理、贡献、技术细节及实验评估结果。

Linux 操作系统已经成为了人类信息化社会运行的基石，其被广泛应用包括云服务器、智能手机、交通系统甚至核电站控制系统等各类计算设备中[168-169]。但是 Linux 操作系统的核心组件——内核则非常复杂且容易出错，除了最被攻击者利用的内核堆漏洞，仍有大量其他类型的内核漏洞[1]被不断公开[60,132,134,142,145,147,156,170-175]，例如整数溢出、未初始化、数据竞争、野/空指针等。尽管现有的 KASAN[40]、KMSAN[41]、KCSAN[42]、UBSAN[43]等消毒器 (sanitizer) 在很大程度上能够帮助开发者理解内核漏洞触发上下文[10,37-39]，但完善的漏洞修复并发布修复补丁 (patch) 仍需要开发者花费大量精力进行漏洞根源分析，无法在超过 66 天的漏洞修复时间窗口中保护内核[3]。

本文提出了基于 eBPF 的防内核漏洞触发框架 PET(**P**revent **E**rrors from being **T**riggered)。PET 规避了复杂且耗时的漏洞根源分析[176-180]，而是将防止内核漏洞触发作为工作的主要目标，漏洞触发的上下文和触发条件均能够通过消毒器的漏洞报告直接获取，因此 PET 能够在漏洞修复补丁发布之前的时间窗口中保护内核。PET 将内核消毒器的漏洞报告作为输入，构造了 eBPF 程序监视漏洞触发上下文的代码执行，检测漏洞触发条件是否满足。如果漏洞触发条件未被满足，则内核继续正常工作。如果漏洞触发条件被满足，则立刻跳过漏洞触发上下文代码，并终止用户层的恶意攻击程序。相较于第三章的 ERA 仅针对内核堆漏洞，PET 主要针对内核漏洞的触发条件而非攻击利用条件，因此具备更强的可扩展性，可以应对多种类型的漏洞。

1　此处为 Software Security Error，直译为软件安全错误，本文后续使用内核漏洞指代

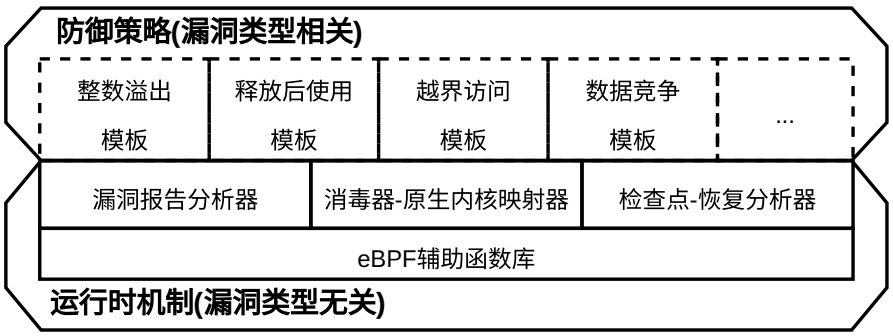


图 4-1　PET 框架的两层架构，上层为漏洞类型相关的防御策略，下层为漏洞类型无关的支持 PET 框架功能的基础设施

PET 框架被设计为如图 4-1 所示的两层架构，分别为漏洞类型相关的上层架构 (§4.4) 和漏洞类型无关的下层架构 (§4.5)。漏洞类型相关上层架构，负责根据漏洞类型给出对应检查漏洞触发条件的防御策略，PET 框架为此提供了一系列 eBPF 程序模板，安全研究者只需将对应的漏洞上下文和触发条件填入模板中即可生成 eBPF 漏洞防御程序。对于漏洞类型无关的基础设施，PET 提供了一系列支持防御策略生成的工具和支持防御策略强制执行的运行时机制，包括 ❶ 漏洞报告分析器，根据消毒器给出的漏洞报告提炼漏洞触发的上下文和漏洞触发条件；❷ 消毒器插桩内核到运行时原生内核的映射器，用于将漏洞报告分析器提炼出的触发上下文和条件映射到目标内核；❸ 检查点-恢复分析器，用于在跳过漏洞触发上下文后将内核恢复到正常工作状态；❹ eBPF 辅助函数库 (helper function)，提供了一系列 eBPF 辅助函数，帮助检查策略运行监控漏洞触发条件是否满足。

为证实 PET 框架的可扩展性，本文目前配备了针对最先进内核消毒器可报告的五种最常见漏洞的防御策略。收集到的消毒器漏洞报告包括整数下溢/溢出，占所有 68 个 UBSAN 报告；越界访问和释放后使用，占 KASAN 报告的 94%(995/1059)；未初始化访问，占所有 295 个 KMSAN 报告；以及数据竞争，涵盖所有 123 个 KCSAN 报告。它们被广泛认为是可利用的漏洞，根据我们的广泛收集，过去十年针对 Linux 内核的 75.4%(184/244) 公开的攻击利用程序是针对这些漏洞的。此外，微软报告称，每年大约 70% 的软件漏洞与这些内存安全问题有关[181]。我们对 v5.15 版本内核的评估显示了 PET 在防止这些漏洞类型方面的有效性。未来，随着新消毒器的设计和发布，PET 可以进一步扩展以支持更多类型。

PET 框架的防御程序的性能开销非常轻量级，在部署 32 个漏洞防御程序时平均性能开销小于 3%，最大额外内存开销仅为 5.56%。对比性能和内存开销均和相关工作 undo workaround[3]在同一水平，但 PET 无需漏洞补丁或重新编译、重启内核，可以在内核漏洞被发现的同时生成 eBPF 防御程序，并运行时部署防御策略。此外，PET 框架在跳过漏洞上下文恢复系统状态的功能存在一定局限性，本文首先通过连续运行 eBPF 防御策略超过 66 天 (漏洞修复时间窗口)，从经验角度验证 PET 框架的可行性，其次本文在后续 §4.8 对局限性进行讨论，并建议将 PET 作为漏洞修复补丁发布前的临时防御措施使用。

本章节贡献如下：

1. 设计并实现了 PET 框架，能够在漏洞修复补丁发布前，防止内核漏洞被攻击者恶意触发。
2. 用目前最先进的消毒器的 5 类漏洞报告，生成了对应的检查漏洞触发条件防御策略，证明了 PET 框架的可扩展性。
3. 实现了 PET 的原型系统并使用真实内核漏洞测试了其安全性和性能开销。

4.2 背景

本章节首先阐明了 Linux 内核的内存区域及对应的生命周期、合法边界、漏洞的类型和根源，帮助读者进一步理解 PET 框架如何防止漏洞触发，其次介绍了 Linux 内核消毒器 (sanitizer)，最后从攻击者和防御者角度介绍了威胁模型。

4.2.1 Linux 内核内存区域及漏洞

linux 内核的所使用的数据对象存储在不同的内核内存区域中，有着不同的生命周期和可访问的合法边界，下面逐一进行介绍。

全局和静态数据区

存储在全局和静态数据区 (例如 `.data`, `.rodata` 区) 的数据对象的生命周期是从内核启动到内核关闭。他们在编译阶段就初始化为 0 或特定的值，而且数据对象的大小也在载入物理内存前被确定，该大小即为内核代码允许访问的合法边界。

内核栈

内核为每个用户空间进程分配了一个栈，用于调用系统调用时使用，并为每个 CPU 分配了一个中断栈来处理外部中断。两种栈中对象的生命周期和边界遵循相同的原则。就生命周期而言，当调用内核函数时，其栈帧被创建，栈数据对象在序言 (prologue) 后开始存在。在尾声 (epilogue) 后，栈帧被销毁，使栈数据对象不再存在。在这一生命周期中，栈数据对象必须在读取前被初始化。就边界而言，虽然 C99 标准引入了变长数组 (VLA) 特性，允许在运行时确定栈数组的长度，但 Linux 内核从 v4.20 开始为了更好的安全性和降低开销而放弃了这一特性。因此，所有栈数据对象的大小在编译时预定义，且对它们的访问绝不应超出边界[182]。

内核堆

内核堆主要由三类堆分配器组成，分别为 buddy、slab 和 vmalloc。正如 §3.2.1 所述，buddy 分配器是所有分配器的基础，管理了所有物理内存，用于分配连续的大于一页的内存；slab 分配器从 buddy 分配器中获取连续的多页内存并使用他们来存储小数据对象，这些分配的页通常被成为 slab 缓存 (slab cache)，每个页都被划分为大小相同的槽 (slot) 用于存储小数据对象；vmalloc 分配器也从 buddy 分配器中获取内存页，但不要求物理地址连续，而是通过构造页表实现虚拟地址连续。

这些内核堆数据对象的生命周期的在调用数据对象分配接口时开始，调用释放接口时结束。三类堆分配器的常见分配/释放接口包括，`alloc_pages/get_free_pages` 系列，`kmalloc` 系统和 `vmalloc` 系列。已经分配的内存在释放后会被回收并重新使用。与栈上的数据对象类似，对数据对象也必须在访问之前初始化，但数据对象的大小可以在编译时预定义，也可以在运行时动态分配，例如 elastic object[147]。一旦数据对象的大小被确定了，对该数据对象的访问必须在合法边界之内。

内核内存漏洞

当内核代码在执行时破坏了内核对象的生命周期或合法访问边界，就会发生内存漏洞，因此可以被归类为时间型 (temporal) 或空间型 (spatial) 漏洞。

释放后使用 (use-after-free) 及其特殊情况双重释放 (double-free)，指的是在栈或堆对象的生命周期结束后，通过悬空指针 (dangling pointer) 访问该对象的情

况。在对象被初始化之前对栈或堆对象进行读取时，会发生未初始化访问漏洞(uninitialization)。全局和静态区域中的内核对象免受时间型漏洞的影响，因为它们从内核启动时就存在，并且会被编译器自动初始化。越界访问 (out-of-bound access) 可能由整数溢出 (integer overflow)、类型混淆 (type confusion)[183-184]、数据竞争 (data race) 以及许多其他根本原因引起。由于此类漏洞并不需要缓冲区，我们使用“越界访问”而非“缓冲区溢出”来描述所有内核对象的读取或写入超出其合法边界的情况。

本研究广泛收集了过去十年针对 Linux 内核的共 244 个公开的漏洞攻击代码，来源包括 Github、BlackHat、BlueHat、Pwn2Own 和个人博客 (例如,[185])。统计数据显示，其中 75.4%(184/244) 利用了上述描述的内存漏洞。具体来说，这些包括 5.7% 的整数下溢/溢出、26.3% 的越界访问、37.3% 的释放后使用、4.9% 的未初始化访问和 1.2% 的数据竞争。这些漏洞可以被消毒器检测并且由 PET 覆盖。

4.2.2 Linux 内核消毒器

Linux 内核消毒器 (sanitizer) 主要用于实现对内核内存漏洞和漏洞根源的探测，其基本原理为使用编译器插桩和影子内存技术监控内核代码执行是否合法。当内核代码执行非法时，消毒器会获取内核发生漏洞的代码执行上下文信息，生成详细的漏洞报告并发送给开发者。本研究正是通过详细分析漏洞报告，防止该漏洞被触发或被攻击者利用。

消毒器可以分为两类：一类是直接检测内存漏洞的，如内核地址消毒器 (KASAN) 用于检测释放后使用和越界访问，内核内存消毒器 (KMSAN) 用于检测未初始化访问。另一类是检测内存漏洞成因的，如未定义行为消毒器 (UBSAN)，它可以在整数被用作缓冲区索引并导致越界访问之前检测整数溢出；内核线程消毒器 (KTSAN) 和内核并发消毒器 (KCSAN)，两者均用于数据竞争，由于 KCSAN 的原理较为简单，使用更为广泛[186]。值得注意的是，消毒器的大规模代码插桩通常会导致内核性能严重下降，因此消毒器一般不用于内核运行时保护[10]。

4.2.3 威胁模型

攻击者角度。攻击者可以获得任何内核子系统 (包括 eBPF 子系统) 的漏洞，同时拥有对应 PoC 程序或漏洞利用程序。这些漏洞可能是尚未公开披露但已被蓝队检测到的 0-day 漏洞，或者是已经披露但尚未修补的 n-day 漏洞。通过执行 PoC 或利用程序，攻击者可以触发漏洞，导致内核崩溃或执行利用程序以提升权限并窃取敏感数据。攻击者知道受害者机器上有 PET 的存在。他们可以通过使用系统调用或发送网络包来间接影响 PET 的执行。然而，eBPF 程序必须由特权用户安装或卸载，攻击者作为非特权用户，不能直接通过卸载 eBPF 程序来禁用 PET。

防御者角度。防御者 (即 PET 的用户) 是拥有特权的系统管理员或蓝队成员，能够根据实际需要，访问存在漏洞内核的调试信息和配置文件，将 eBPF 程序安装到内核空间来部署 PET。我们假设在 PET 保护指定漏洞前，内核仍未被攻陷，攻击者依旧不具备 root 权限。同时运行整个系统的硬件不存在漏洞，来自恶意硬件的攻击也暂时不予考虑。我们假设内核中 eBPF 验证器检查过的 eBPF 程序是没有漏洞的，但 eBPF 生态系统的漏洞暂时不予考虑，理由见 §2.4 局限性。防御者能够发现漏洞并有一个由消毒器生成的报告。对于突发的 0-day 漏洞攻击利用，防御者可以通过在一个消毒器插桩的内核上复现它们来获得漏洞报告。对于 n-day 漏洞，大部分 Linux 内核漏洞由 Syzkaller 报告[1]，它在发现内核漏洞时生成并发布消毒器报告。对于通过静态分析检测到的漏洞，防御者可以手动使用消毒器插桩的内核来验证并生成报告。注意，报告信息可能不总是准确的 (例如，KCSAN 和 KTSAN 报告的良性数据竞争)。为了解决这个问题，本文鼓励防御者使用之前研究工作[187]中讨论的方法提供更正。鉴于单一报告只展现了漏洞的一种触发行为，防御者可以使用[188]的技术来多样化漏洞触发行为并生成相应的报告。

4.3 概述

PET 框架的核心目标是根据漏洞报告生成 eBPF 程序，监控漏洞触发点的漏洞触发条件是否满足。如果漏洞触发条件满足，eBPF 程序能够跳过漏洞触发点，将内核恢复到工作状态避免崩溃，并终止攻击者利用进程。如果漏洞触发条件不满足，内核继续不受影响正常工作。本文以 CVE-2016-6187 越界访问为例说

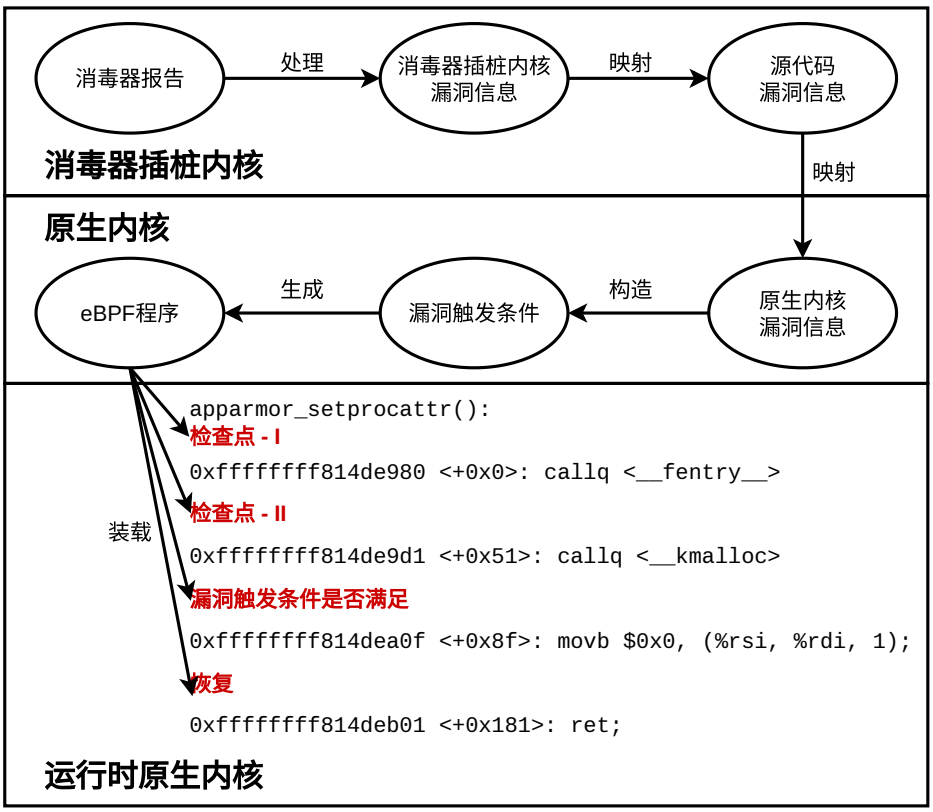


图 4-2 PET 框架三阶段的工作流程

明 PET 框架如何避免内核漏洞触发的工作流程，如图 4-2 所示。

在第一阶段，PET 分析消毒器报告以提取有关消毒器插桩内核中漏洞的关键信息。代码 4-1 显示了当 CVE-2016-6187 在 v5.15 版本中迁移并复现时的部分 KASAN 报告。从报告的第 1 和 2 行，PET 确定漏洞是一个影响由 slab/slub 分配器管理的内存的越界写入。从第 1 和 4 行，PET 精确定位了触发越界写入的漏洞触发点是 `apparmor_setprocattr+0x116` ——从函数 `apparmor_setprocattr` 开始的偏移量 `0x116`。对消毒器插桩的内核镜像的进一步分析揭示了漏洞触发点位于指令 `movb $0x0, (%r14)`。当漏洞被触发时，寄存器`%r14` 引用了 slab cache 中某个内核数据对象的合法边界之外的区域。

在第二阶段，PET 将消毒器插桩内核中的漏洞触发点映射到原生内核，并相应地构建触发条件。由于消毒器在编译期间插桩内核，消毒器插桩的内核和原生内核中的漏洞触发点及触发条件在二进制级别上是不同的。在我们的示例中，原生内核中的漏洞触发指令是 `apparmor_setprocattr+0x8f` 处的 `movb $0x0, (%rsi, %rdi, 1)`，这与消毒器插桩的内核中的对应指令 `movb $0x0, (%r14)` 不同。PET 利用消毒器插桩的内核和原生内核的 DWARF 调试信息，将消毒指令映射

到源代码 `args[size] = '0'`，然后映射到原生内核指令。通过进一步分析映射的 `mov` 指令，PET 了解到寄存器`%rsi` 指向溢出的缓冲区 `args`，而`%rdi` 存储造成越界访问的索引大小的值。报告的越界访问漏洞的触发条件被编码为`%rsi+%rdi*1` 超出 `args` 缓冲区的边界。这个触发条件在 eBPF 程序中借助我们添加的 eBPF 辅助函数进行表达。

第三阶段利用 eBPF 生态系统的优势，在运行时将 eBPF 程序安装到原生内核中。这个防止漏洞触发的 eBPF 程序在原生内核中的 `mov` 指令执行之前立即执行。它使用 BPF 辅助函数从内核执行环境中 `mov` 指令的运行时信息，并检查是否满足触发条件。如果没有满足，内核将继续其正常执行。否则，这表明内核内存将被 `mov` 指令破坏，eBPF 程序将立即采取行动以防止这种情况发生。

```
KASAN: slab-out-of-bounds in apparmor_setprocattr+0x116/0x590
   Write of size 1 at addr ffff888007449c80
   Call Trace:
   apparmor_setprocattr+0x116/0x590
   proc_pid_attr_write+0x15f/0x1e0
```

代码 4-1　CVE-2016-6187 在 v5.15 版本中移植并复现时的部分 KASAN 报告

```
static int apparmor_setprocattr(const char *name, void *value,
      size_t size)
{
   ...
   if (args[size - 1] != '\0') {
      ...
      if (size == PAGE_SIZE)
         return -EINVAL;
      args[size] = '\0'; // off-by-one-byte
   }
   ...
   return error;
}
```

代码 4-2　CVE-2016-6187 访问越界代码片段

4.4 漏洞类型相关的防御策略

本文在此节中详细介绍五种最常见内核漏洞的防御策略，这些漏洞由目前最先进的消毒器产生对应的漏洞报告。这些漏洞是最经常被攻击者利用的，也几乎是目前消毒器能报告的全部漏洞。

4.4.1 整数溢出策略

整数溢出 (underflow/overflow) 类型漏洞可以被 UBSAN 消毒器探测，通常包括三类特定场景，(1) 算数溢出 (arithematic overflow)，例如 `add %al, 0x2`，当`%al` 为 `0xffff` 时超出了 16 位寄存器的范围导致溢出。(2) 移位溢出 (shift overflow)，例如 `shl %al, 10`，当`%al` 为 `0x0012` 时超出了 16 位寄存器的最高位导致溢出。(3) 隐式转换 (implicit conversion)，例如 `mov %rbx, %eax`，将 64 位寄存器中的内容保存在 32 位寄存器中，造成高 32 位内容丢失导致溢出[1]。

```
1 UBSAN: shift-out-of-bounds in
    ↪ drivers/usb/gadget/udc/dummy_hcd.c:2293:33
2 shift exponent 257 is too large for 32-bit type 'int'
3 Call Trace:
4    dummy_hub_control.cold+0x1a/0xbc
       ↪ drivers/usb/gadget/udc/dummy_hcd.c:2293
```

代码 4-3 UBSAN 探测到的整数溢出报告片段[189]

对于上述三类整数溢出漏洞的防御策略是相同的，即将 eBPF 程序插入到导致漏洞触发的指令之前，提取指令的操作数，并使用 2 倍大小的寄存器模拟漏洞触发指令的计算。例如，将原始的 32 位操作数替换到 64 位的寄存器中进行运算，原本的 32 位值`%eax+%ebx` 就变成了 64 位的`%rax+%rbx`, 通过对比两次计算的结果即可判断是否发生了整数溢出。eBPF 的程序模板可以通过 `(%eax+%ebx)==(%rax+%rbx)?false:true` 形式的代码进行判断。对于移位溢出还需要额外的检查判断移位的长度是否小于原始操作数的长度，例如代码 4-3 中，移动的长度在 32 位的 int 类型中必须小于 32。

如果满足漏洞触发条件，即上述表达式返回值为 `true`，那么 PET 会阻止整数溢出漏洞被触发。为了实现该目标 PET 提供了底层的防御机制和预先定义

1 该指令仅用于说明，x86 并不存在该指令

的 eBPF 程序目标作为防御策略，系统管理员只需要根据漏洞详细信息填写代码 4-4 中 eBPF 模板的第 1 和 4 行，即可生成对应的漏洞防御程序。

```
1  SEC("kprobe/?") // 漏洞触发点
2  int BPF_KPROBE(...) {
3      // 模拟漏洞触发指令的计算
4      if (?) { // 满足触发条件
5          // 记录错误事件
6          // 发送 SIGKILL 终止恶意进程(中断处理中除外)
7          // 跳过漏洞触发指令
8          // 控制流指向函数结尾
9          return -1;
10 }
```

代码 4-4　整数溢出防御策略的 eBPF 程序模板，”?” 表示需要被填入的漏洞相关细节

4.4.2　越界访问策略

越界访问能够被 KASAN 探测并报告，具体的漏洞报告如代码 4-1 所示，不仅包括漏洞的触发点，还包括了漏洞的触发区域，即栈、堆、全局/静态区。eBPF 程序类似整数溢出防御策略，挂载在漏洞触发点点之前，监视访问内存时是否超出了数据对象的合法边界。位于不同区域的数据对象的合法边界有不同的定义。

对于位于栈和全局/静态区域的数据对象，合法的访问边界可以直接从 DWARF 调试信息中获取。而位于堆的数据对象长度通常是运行时代码动态决定的，例如 elastic object[147]，本文没有使用从消毒器漏洞报告或者通过静态分析获取的数据对象大小，而是加入了两个 BPF 辅助函数 (helper function)，`bpf_get_start` 和 `bpf_get_len` 来读取堆管理相关的内核元数据进而确认数据对象的合法边界。

更具体地说，`bpf_get_start` 将一个地址作为参数并确定该地址所指对象的起始地址。它调用内核函数 `find_vm_area` 和 `virt_to_page` 来检查这个地址属于哪个堆区域。如果地址位于 vmalloc 区域，它读取 `struct vm_struct->addr` 来获取起始地址。如果地址是由 buddy 管理的内存，buddy 分配器负责处理跨越多个页面的大对象，溢出对象的起始地址是由 `struct page` 对象表示的第一页的地址。如果内存地址由 slab/slub 分配器管理，该分配器负责管理页面粒度内的小对象，它根据数据对象所在的 slab 槽的大小来计算起始地址。

同理 `bpf_get_len` 获取堆数据对象的大小，vmalloc 数据对象的大小存储在数据结构 `page->slab_cache->objec_size` 中。对于 buddy 分配器的数据对象，如

果 page->flags 位为 0，那么该数据对象的大小为一页 (4KB)，否则数据对象的大小通过 page->compound_nr*PAGE_SIZE 计算。

对于所有的越界访问漏洞,触发条件可以由表达式 (start<=addr)&&(addr<end)?false:true 表示。仍以 CVE-2016-6187 为例，eBPF 程序模板 4-5 需要填写的内容为，内核代码访问地址 addr 对应 %rsi+%rdx*1,数据对象起始地址 start 对应 bpf_get_addr(%rsi),数据对象结束地址 end 对应 bpf_get_addr(%rsi)+bpf_get_len(%rsi).

```
1  SEC("kprobe/?") // 漏洞触发点
2  int BPF_KPROBE(...) {
3      u64 addr = ?;
4      u64 start = bpf_get_start(addr);
5      u64 end = start + bpf_get_len(addr);
6      if (addr<start||addr>=end)
7          // 满足触发条件
8          // 记录错误事件
9          // 发送 SIGKILL 终止恶意进程(中断处理中除外)
10         // 跳过漏洞触发指令
11         // 控制流指向函数结尾
12         return -1;
13 }
```

代码 4-5 越界访问防御策略的 eBPF 程序模板，”?” 表示需要被填入的漏洞相关细节

4.4.3 释放后使用策略

释放后使用类型漏洞也由 KASAN 探测并报告，由于其时间型的特性一般包括两个漏洞触发点。第一个触发点为导致悬空指针产生的数据对象释放点，例如代码 4-6 第 9 行展示的 route4_delete_filter_work+0x17。第二个触发点为导致悬空指针解引用的指令，例如代码 4-6 第 4 行的 route4_get+0x7d。为了阻止释放后使用类型的漏洞，eBPF 程序需要装载在这两个漏洞触发点前。

```
1  KASAN: use-after-free in route4_get+0x7d/0xc0
2  Read of size 4 at addr ffff888006358640
3  Call Trace:
4      route4_get+0x7d/0xc0
5  Allocated by task 1137:
6      route4_change+0x18f/0xde0
```

```
Freed by task 69:
   kfree+0x90/0x220
   route4_delete_filter_work+0x17/0x20
```

代码 4-6 KASAN 探测到的释放后使用漏洞报告片段[190]

在数据对象释放位置之前的 eBPF 程序会将数据对象地址记录到一个 BPF map 中，并通过跳过释放操作来隔离数据对象。这意味着该数据对象不会被释放，也不会回收到分配器中以供将来重用。这种隔离机制能防止攻击者的攻击利用，因为攻击者无法重新占用同一块内存并将释放的数据对象与另一个受控的数据对象重叠，而这是利用释放后使用漏洞的必要条件。本文将释放位置的操作记为 `map=map+addr`。在使用位置之前的 eBPF 程序会查询 BPF map，以查看被解引用的指针是否指向一个被隔离的对象。如果是，eBPF 程序将采取行动防止漏洞触发，与其他策略类似。这个触发条件被记为 `ptr` ∈ `map?true:false`。通过将漏洞触发点和触发条件填充到代码 4-7 模板中，可以合成对应的 eBPF 程序。注意，双重释放是释放后使用的特殊情况。二者检测策略没有区别，除了双重释放的使用位置也是一个释放位置。

然而实现释放后使用防御策略的真正挑战不在于如何构造漏洞触发条件，而是如何避免内存耗尽。因为使用保持隔离数据对象而不进行循环使用，攻击者可以通过重复调用数据对象释放点来占用大量内存影响系统正常运行。因此 PET 的释放后使用策略必须实现对隔离数据对象的安全释放。

PET 的解决办法参考了用户空间释放后使用探测器常用的隔离-扫描策略 (quarantine & sweep)[28-31]来防止内存耗尽。首先 PET 使用了两个 BPF 辅助函数，`bpf_timer_init` 和 `bpf_timer_start` 在数据对象的释放点启动了一个定时器。定时器会定时唤醒一个内核线程来执行回调函数，该回调函数可以通过 `bpf_timer_set_callback` 辅助函数进行设置。在该防御策略中，回调函数被设置为一个扫描整个物理内存的扫描器，检查内存中是否存在悬空指针指向的被隔离的数据对象。如果发现了指向被隔离数据对象的悬空指针，那么该数据对象将继续被隔离。若没有发现指向被隔离的数据对象的指针，则该数据对象则可以被安全释放并重新使用。这个扫描器被集成在 eBPF 程序的模板中，作为防御机制可以直接被调用。

由于物理内存空间通常非常大，导致对物理内存的扫描通常开销较大，影响

系统性能。对此 PET 根据内核特性给出优化，因为悬空指针通常存储在指定的 slab/slub 数据对象中，那么只需要在分配点提取分配数据对象的类型，然后检查其他类型的数据对象中是否存在被提取数据对象类型的成员。如果存在，该成员可能成为一个悬空指针，被优化的扫描器只需扫描存储这些数据对象的 slab cache 中对应指针成员的偏移量，而非全部物理内存。

```
1  SEC("kprobe/?") // 漏洞触发点
2  int BPF_KPROBE(...) {
3     u64 addr = ?;
4     u64 *v = bpf_map_lookup_elem(map,addr);
5     if (v) // 满足触发条件
6        // 记录错误事件
7        // 发送 SIGKILL 终止恶意进程(中断处理中除外)
8        // 跳过漏洞触发指令
9        // 控制流指向函数结尾
10       return -1;
11 }
12
13 SEC("kprobe/?") // 数据对象释放/隔离点
14 int BPF_KPROBE(...) {
15    u64 addr = ?;
16    // quarantine
17    bpf_update_map_elem(map,addr);
18    if (sweep) // 启动扫描器
19       bpf_sweep(&map,...)
20 }
```

代码 4-7　释放后使用防御策略的 eBPF 程序模板，”?” 表示需要被填入的漏洞相关细节

4.4.4　未初始化策略

未初始化访问类型漏洞由 KMSAN 探测并报告。作为另一种时间型内存漏洞，它也有两漏洞触发点。一个是创建点，即在栈或堆上创建数据对象的地方，如代码 4-8 中第 4 行指示的 `__sys_recvfrom+0x81`。另一个是访问地点，即数据对象被读取但未完全初始化的地方，如代码 4-8 中第 2 行指示的 `tcp_recvmsg+0x6cf`。

```
1  KMSAN: uninit-value in tcp_recvmsg+0x6cf/0xb60
2     tcp_recvmsg+0x6cf/0xb60
3  Local variable msg created at:
```

```
4    __sys_recvfrom+0x81/0x900
```

代码 4-8　KMSAN 探测到的未初始化访问漏洞报告片段[191]

因此，PET 需要两个 eBPF 程序来防止未初始化访问。第一个 eBPF 程序安装在创建点之后。它使用对象的地址作为两个 map 的 key，在 BPF map X 中存储创建的数据对象的大小，在 BPF map Y 中存储未初始化的内容。第二个程序使用数据对象的地址查询 map X 并检索其大小，然后使用数据对象大小从 map Y 检索对象的完整内容。通过比较访问前和创建后对象的内容，该安全策略可以确定该数据对象是否已经被正确初始化。激进的策略认为，如果内容至少有一个字节相同，则触发条件得到满足。保守的策略要求内容完全相同。这两种策略各有问题。激进策略可能导致假阳性，因为一些字节在初始化后保持不变；而保守策略可能错过部分初始化，导致假阴性。实验结果表明，保守政策更有效，因此默认在 eBPF 模板 4-9 中使用。如果允许手动调整，理想的策略应该是指定对象内未初始化的范围。

```
1  SEC("kprobe/?") // 未初始化对象访问点
2  int BPF_KPROBE(...) {
3     u64 addr = ?;
4     u64 len = bpf_map_lookup_elem(lenmap,addr);
5     u64 *mem = bpf_map_lookup_\elem(memmap,addr);
6     u64 *cur = bpf_core_read(addr, len);
7     if (compare(mem,cur,len)!=0)
8        // 满足触发条件
9        return -1;
10 }
11
12 SEC("kprobe/?") // 未初始化对象创建点
13 int BPF_KPROBE(...) {
14    u64 addr = ?;
15    u64 len = ?;
16    u64 *mem = bpf_core_read(addr,len);
17    bpf_map_update_elem(lenmap,addr,len);
18    bpf_map_update_elem(memmap,addr,mem);
19 }
```

代码 4-9　未初始化防御策略的 eBPF 程序模板，”?” 表示需要被填入的漏洞相关细节

4.4.5 数据竞争策略

数据竞争发生在两个不同 CPU 上执行的指令同时访问同一块内存，而没有适当同步时。例如，代码 4-10 展示了一个数据竞争的情况，其中 CPU 1 在 `tcp_send_challenge_ack+0x116` (第 3 行) 处从 0xffffffff8713bbb0 读取数据，而 CPU 0 则在相同的地址 `tcp_send_challenge_ack+0x15c` 处写入数据 (第 5 行)。

```
1 BUG: KCSAN: data-race in tcp_send_challenge_ack /
   ↪ tcp_send_challenge_ack
2 read to 0xffffffff8713bbb0 of 4 bytes on cpu 1:
3    tcp_send_challenge_ack+0x116/0x200
4 write to 0xffffffff8713bbb0 of 4 bytes on cpu 0:
5    tcp_send_challenge_ack+0x15c/0x200
```

代码 4-10 KCSAN 探测到的数据竞争报告片段[192]。

在 PET 中，我们利用四个 eBPF 程序来实时识别数据竞争，如代码 4-11 所示。这四个 eBPF 程序成对组织，并共享一个 BPF map。每对负责监视潜在数据竞争中的一个访问。每对程序中的第一个程序在访问之前安装，并执行 P 操作：调用带有 `BPF_NOEXIST` 参数的 `bpf_map_update_elem` 辅助函数，以自旋锁定共享映射并原子性地进行查找更新。如果共享 BPF map 中存在相同内存的记录，则返回 `EEXIST`，表明另一个来自不同 CPU 的指令当前正在访问同一内存，从而标志着潜在的数据竞争。如果没有数据竞争，辅助函数返回 0，表明更新成功且没有发生数据竞争。每对程序中的第二个程序在访问后安装，并执行 V 操作：调用 `bpf_map_delete_elem` 以原子性地删除已访问内存的记录。由于数据竞争可能是良性的也可能是有害的[171,193-194]，默认情况下，数据竞争的 eBPF 程序不会像其他漏洞那样，在 P 操作失败时发送 `SIGKILL` 信号终止当前进程。当然，如果有专家确认报告的数据竞争造成了损害，eBPF 模板也支持终止进程。

```
1 SEC("kprobe/?") // P 操作点
2 int BPF_KPROBE(...) {
3    u64 addr = ?;
4    int err = bpf_map_update_elem(map,addr,BPF_NONEXIST);
5    if (err!=0) // 数据竞争
6 }
7
8 SEC("kprobe/?") // V 操作点
9 int BPF_KPROBE(...) {
```

```
10    u64 addr = ?;
11    bpf_map_delete_elem(map,addr);
12 }
```

代码 4-11 数据竞争防御策略的 eBPF 程序模板，“?”表示需要被填入的漏洞相关细节

4.5 漏洞类型无关的防御机制

PET 框架中漏洞无关的防御机制用于协助漏洞防御策略的生成和运行，主要包括漏洞报告处理器、消毒器-原始内核映射器、检查点-恢复分析器和 eBPF 辅助函数库四类组件。

4.5.1 漏洞报告处理器

漏洞报告处理器负责从消毒器报告中提取关键信息。正如我们在前几节中展示的，不同漏洞类型的消毒器报告格式各不相同：越界访问的漏洞触发点可以在报告标题中找到 (代码 4-1 中的第 1 行)，而整数下溢的漏洞触发点在函数调用栈中 (代码 4-3 中的第 4 行)。除了漏洞触发点，一些预防策略还需要特殊的信息。例如，释放后使用漏洞需要分配地点 (代码 4-6 中的第 6 行) 来优化悬空指针扫描器。为了消除信息需求上的差异，PET 中的报告处理器提供了一个统一的接口。这个接口是一组正则表达式——从提前分隔好的报告行中过滤出所需的关键词。

4.5.2 消毒器-原始内核映射器

从报告中提取的漏洞信息需要从消毒器插桩的内核翻译到原生内核，以构建 eBPF 程序检查内核漏洞的触发条件。为了展示这一映射过程，本文继续使用 slab 越界访问漏洞 CVE-2016-6187 作为示例。

在这个消毒器-原生映射器中，有两个翻译流程。第一个流程识别可以装载防止漏洞触发的 eBPF 程序的漏洞触发点。它将在消毒器插桩的内核中触发漏洞的指令翻译为源代码中的代码行，最终翻译为原生内核中的对应指令。如图 4-3 所示，以报告处理器输出的 `apparmor_setprocattr+0x116` 为线索，PET 使用它来精确定位消毒器插桩内核镜像中的漏洞触发指令 `movb $0x0, (%r14)`。利用消毒器插桩的内核的 DWARF 调试信息，PET 将这一漏洞触发指令映射到源代

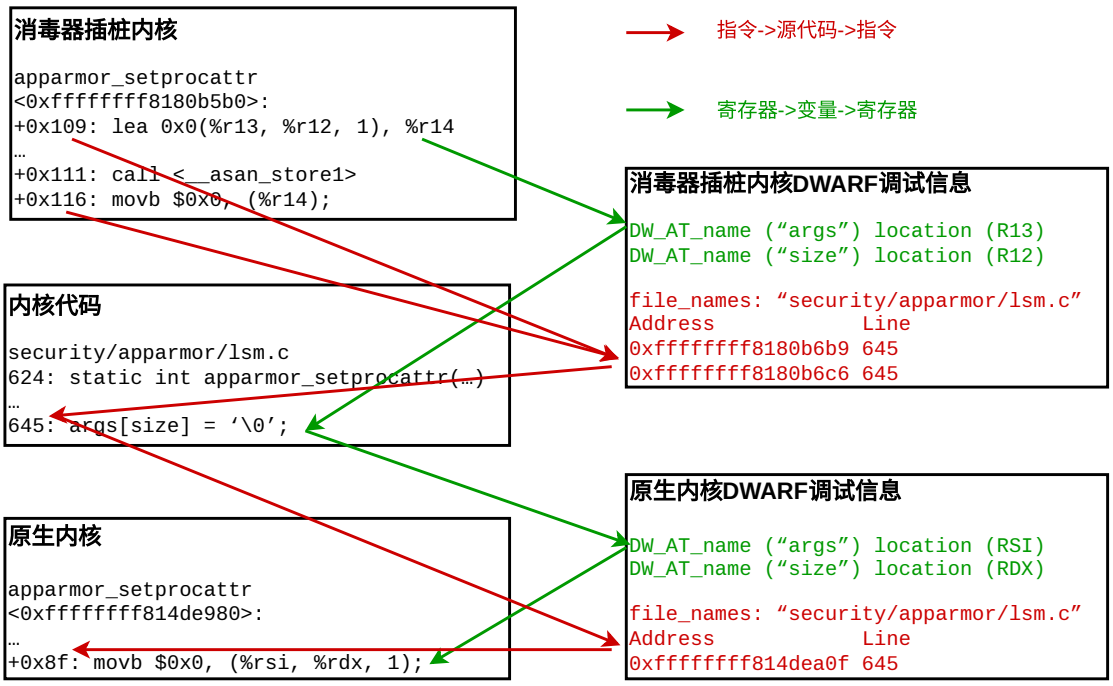


图 4-3 在消毒器-原生映射器中的两个翻译流程，它们在二进制级别构建触发条件，以便被表达在检测原生内核中漏洞的 eBPF 程序中

码中 `securit/apparmor/lsm.c` 文件的 645 行。无需进一步分析，PET 检索原生内核的 DWARF 调试信息，将这一行源代码映射到原生内核镜像中的对应指令——`apparmor_setprocattr+0x8f` 处的 `mov` 指令，并插入 eBPF 程序防止内核漏洞触发。

第二个翻译流程构建了 eBPF 程序中实现的触发条件。在图 4-3 中，mov 指令的目标操作数`%r14` 记录了消毒器插桩的内核中写入的地址。这个地址通过在 `apparmor_setprocattr+0x109` 处的 `lea 0x0, (%r13, %r12, 1)` 指令计算得出。通过分析这条 `lea` 指令的 DWARF 信息，PET 了解到寄存器`%r13` 指向一个名为 `args` 的缓冲区，`args` 位于一个 slab cache 中。内核使用名为 `size` 的索引访问这个缓冲区，该索引存储在寄存器`%r12` 中。因此，PET 得出结论，溢出的缓冲区是 `args`，溢出的索引是 `size`。为了继续第二个翻译流程，PET 使用原生内核的 DWARF 信息，并在漏洞触发点将 `args` 变量与`%rsi` 寄存器、`size` 变量与`%rdx` 寄存器相连接。

如 §4.4.2 所述，越界访问的触发条件是 `(start<=addr)&&(addr<end)?true:false`。eBPF 程序使用这个条件来检查访问的地址`%rsi+%rdx*1` 是否在合法边界内，以 `bpf_get_addr(%rsi)` 作为起点，`bpf_get_addr(%rsi)+bpf_get_len(%rsi)` 作为终点。这个条件被填充进 eBPF 程序，该程序安装在通过第一个翻译流程识别的漏

洞触发点，以防止越界访问漏洞。

4.5.3 检查点-恢复分析器

当消毒器-原始内核映射器构造的触发条件在原生内核中被满足，PET 框架采取了包括检查点-恢复 (checkpoint-restore) 在内的一系列运行时机制保证系统的平稳运行。当内核漏洞触发条件满足时，PET 构造的 eBPF 程序能够跳过内核漏洞的触发点，不执行漏洞触发指令，并保证内核持续响应。PET 通过额外的 eBPF 程序和 BPF map 来实现这一需求。

PET 框架通过两个 eBPF 程序来跳过内核漏洞的触发点。第一个程序在存在漏洞触发点函数的起始点，在函数序言执行前保存寄存器上下文 (内核使用全部寄存器的值)。第二个程序在函数的退出点，在执行函数尾声之前恢复第一个程序保存的寄存器上下文，并重写`%rax` 寄存器，将错误码 (ERROR CODE) 返回给此函数的调用者处理。通过返回错误码，PET 能够利用 Linux 内核中的错误处理机制，快速地沿着调用链将错误信息传递回系统调用入口，尽可能快地终止恶意进程。为了返回合适的错误码，检查点-恢复分析器将遍历函数中的错误处理路径并选择最常用的一个。如果最常用的错误码是 `EAGAIN`，这表示需要重新执行被调用的函数，PET 将其替换为 `EACCESS` 以防止无休止的漏洞触发。在返回类型为指针或 `void` 的情况下，PET 使用 `PTR_ERR` 将返回值转换为负数或重写`%rax` 寄存器为 0。

在多数情况下，上文的检查点-恢复机制足够防止漏洞被触发并保持系统平稳运行。然而在漏洞触发点所在的函数可能存在一些成对的操作，在函数起始点开启功能，函数结束点关闭功能，例如数据对象分配 ((de)allocation)、加解锁 ((un)lock)、设备注册/注销 (device (un)registration)、引用计数器增/减 (reference count (in|de)crease)。PET 增加了额外的 eBPF 程序来解决此类问题，通过一个共享的 BPF map 记录数据对象分配、锁、注册的设备数据结构、引用寄存器的地址，然后在函数结束时根据 BPF map 记录的地址去执行对应的配对操作，即释放数据对象、解锁、注销设备、恢复引用计数器。为了准确定位上述检查点，PET 使用静态工具分析了所有从函数起始点到漏洞触发点的代码执行路径，识别出这些成对的操作，并使用消毒器-原生内核映射器在原生内核镜像中定位。

局限性。尽管本文的实验评估表明，在 PET 防止漏洞触发后内核运行稳定——

这可能是由于从检查点到恢复的路径较短 (平均 44.17 条指令) 所致，但本文强调 PET 作为恢复一致状态的临时解决方案的存在一定局限性。一方面，PET 依赖于人类专家来标记成对操作。这可能导致遗漏，尤其是内核功能始终在进行开发，这要求 PET 相应地进行更新。另一方面，与之前采用日志记录和恢复技术的工作 (例如,[195-199]) 相比，PET 也可能会忽视影响内核功能的内存单元[200]。因此，建议在官方补丁发布时将 PET 下线并应用官方补丁。

4.5.4 eBPF 辅助函数库

PET 采用三种类型的 eBPF 程序：(1) 在函数入口处安装的检查点和针对漏洞的成对操作，(2) 放置在漏洞触发点之前的漏洞触发预防，(3) 在函数出口处安装的恢复程序。PET 通过增加更多的 eBPF 辅助函数来增强 Linux 内核中现有的 eBPF 生态系统，以支持这三种 eBPF 程序类型。

对于检查点程序，PET 使用 `bpf_get_regs` 来检索寄存器上下文，找到分配的数据对象、锁、设备等数据的地址，并将它们存储到与恢复程序共享的 BPF map 中。考虑到漏洞所在函数可能会被重新进入，这些 BPF map 的 key 是函数名称和使用 `bpf_get_current_pid_tgid` 辅助函数获取的当前进程 ID 的组合。

对于漏洞触发预防程序，PET 使用 `bpf_printk` 将恶意事件记录到 `/sys/kernel/debug/tracing/trace_pipe`，并使用 `bpf_send_signal` 发送 `SIGKILL` 信号——这两个辅助函数已经存在于 eBPF 生态系统中。然后，PET 添加 `bpf_set_regs` 将寄存器 `%rax` 设置为魔数 `0xdeadbeef`。这个魔数通知恢复程序触发条件已经满足，必须进行恢复。接着，PET 再次调用 `bpf_set_regs` 设置寄存器 `%rip`，指导内核跳转到函数退出。

最后，对于函数出口处的恢复程序，PET 使用 `bpf_get_regs` 检查寄存器 `%rax`。如果 `%rax` 不等于魔数 `0xdeadbeef`，BPF 程序会清除掉共享的 BPF map 记录的寄存器上下文并恢复到上层函数。如果相等，BPF 程序会使用 `bpf_set_regs` 恢复内核程序的寄存器上下文，使用新增加的 `bpf_kfree` 释放被分配的数据对象，使用 `bpf_unlock` 解锁，使用 `bpf_register` 注册或注销设备，使用 `bpf_(in|de)refcnt` 来增加或减少引用计数器，最后通过 `bpf_set_regs` 设置 `%rax` 返回的由检查点-恢复分析器分析出来的错误码。

4.6 实现

PET 共使用了 500 行 Python 脚本来整合所有漏洞防御机制和防御政策的 eBPF 程序模板。为了实现 BPF 辅助函数，新增了 639 行 C 代码，另外还为所有 eBPF 程序模板新增了 440 行 C 代码。DWARF 分析和对 LLVM IR 的静态分析大约包括 3000 行 C/C++ 代码。

eBPF 程序

内核地址空间布局随机化 (KASLR) 每次系统启动时随机更改内核基址。这不影响安装 eBPF 程序的位置选择,因为该位置可以使用相对偏移量如"func+offset"来指定。对于在 eBPF 程序中无法静态指定的其他地址，PET 使用最新的 BPF 工具链 Skeleton[201]。Skeleton 将这些地址定义为全局变量，并在加载 eBPF 程序时在`/proc/kallsyms` 中查找它们。通过向这些地址添加相对偏移量，Skeleton 重写全局变量的值以绕过 KASLR。在针对释放后使用漏洞的防御策略中，为了性能考虑，扫描器每次唤醒时只扫描固定的内存范围，避免过度的 CPU 占用。在实验中，本文经验性地测试了不同的范围大小和时间间隔，发现每 8 秒一个 256 MB 的范围是最佳选择，如图 4-4 所示。

DWARF 分析

消毒器-原生映射器使用 DWARF 调试信息来识别原生内核中的漏洞触发指令。然而，其准确性可能因编译器优化而受到影响，因为消毒器插桩的内核中的单一指令对应多个源代码语句，进一步对应于原生内核中的多条指令。映射器可能识别出原生内核中的一组指令，但其中只有少数是漏洞触发指令。PET 通过在映射器中交叉检查两个翻译流程来解决这个问题。当且仅当原生内核中的指令的操作符和其操作数变量的名称与消毒器插桩的内核中的漏洞触发指令匹配，该指令才被视为内核漏洞触发点。这种交叉检查将本研究评估测试集的误报数从每个案例的 2 个误报减少到 0。

静态分析

PET 采用静态分析方法来定位检查点位置并优化释放后使用预防策略中的扫描器。本文的实现基于 LLVM 基础设施，并使用自定义的 Clang[202]生成未优化的 IR 文件。

为了识别需要检查点的成对操作，本文维护了一系列分配接口 (例如，`kmalloc`)、锁接口 (例如，`raw_spin_lock`)、设备注册接口 (例如，`register_filesystem`)、引用计数接口 (例如，`refcount_inc`)。静态分析从构建包含漏洞触发指令的函数的控制流图开始，从函数的入口点到漏洞触发指令，遍历所有路径，收集所有 Call 指令，将它们标记为检查点位置。

PET 选择可能包含悬空指针的 slab cache 以优化扫描器的扫描过程。静态分析采用工作列表程序 (worklist procedure)，并同时分析 `GetElementPtr` 和 `Cast` 指令。这一工作列表程序是可靠的，因为即使在最坏的情况下，即所有 slab cache 都被选中时，扫描器也能降级为全面扫描模式正常工作。

4.7 实验评估

在此章节中，本文收集了真实世界的内核漏洞，生成对应的 eBPF 漏洞防御程序，并构造了对应的实验评估 PET 的有效性，性能和内存开销以及可扩展性。

4.7.1 测试用例

我们从两个来源收集了现实世界中的 34 个漏洞作为 PET 的测试用例。一部分是 Syzkaller 在 2018 到 2023 年间报告的。另一部分是过去 10 年中公布的内核攻击利用工作中使用的漏洞。测试用例集中的所有漏洞必须满足以下标准：1. 它们的 PoC 程序或利用程序可公开获取，用于评估 PET 防护成功性；2. 它们可以迁移到 v5.15 版本 (最近的长期维护版本 LTS) 内核，并能成功复现，以便在多个内核漏洞同时存在时测量 PET 的可扩展性和开销。

根据这些标准我们构建了一个覆盖 5 种不同漏洞类型的测试用例集，包括 UBSAN 报告的 2 个整数溢出，KASAN 报告的 15 个越界访问和 10 个释放后使用，以及 KMSAN 报告的 2 个未初始化访问。这个测试用例集具有代表性，覆盖了现有消毒器报告的各种常见漏洞类型。特别地，我们的测试用例集涵盖了所有可能被破坏的内存区域类型，包括栈、全局和静态区域、buddy、slab/slub 分配器和 vmalloc 区域。

由于 KCSAN 报告没有附带任何概念验证 (PoC) 程序，无法找到满足选择标准的数据竞争。因此，我们随机选择了两个满足上述标准的竞态条件 (race

condition) 加入测试用例集：CVE-2017-2636[203]和 CVE-2021-4083[204]，并随机选择了 v5.15 中报告的 3 个数据竞争用于性能测量。另外我们没有找到任何来自 KTSAN 的报告。

4.7.2 有效性

本研究构造了三组实验并对比相关工作，通过回答以下问题来评估 PET 的有效性，❶ PET 能够阻止内核漏洞的触发和攻击利用吗? ❷ 当漏洞触发条件没有被满足，PET 会影响内核的正常工作吗？❸ 在内核漏洞触发被 PET 阻止后，内核仍然能够保持稳定运行不发生内核恐慌 (panic) 吗?

实验设置

为了回答第一个问题，我们将测试用例集中的所有内核漏洞迁移到了 v5.15 内核。首先我们构建了一个启用了消毒器的内核镜像，以验证收集的 PoC 和漏洞利用程序是否可以触发这些内核漏洞并生成 PET 将要分析的报告。之后我们编译了不启用消毒器的原生内核镜像，并将合成的 eBPF 程序部署到这个内核上。最后我们在 PET 加固的内核上运行 PoC 和漏洞利用程序，观察这些漏洞是否仍然可以被触发和利用。相关工作 undo workaround[3]的实验设置与本章节类似，不同之处在于相关工作的第二个内核镜像被植入了撤销 (undo) 操作。

为了回答第二个问题，我们修改了收集到的 PoC 和漏洞利用程序，确保漏洞触发点仍然被执行，但触发条件不再满足。我们在消毒器插桩的内核上确认了这一点，然后在加入内核漏洞的原生内核上运行修改后的 PoC 和漏洞利用程序。我们观察 eBPF 程序是否仍会记录漏洞，如果漏洞被记录则表明发生误报。这次修改大约花费了 150 人小时 (man-hour)。

为了回答最后一个问题，我们通过重复运行所有 PoC 和漏洞利用程序 100 次，对迁移了漏洞的内核进行了广泛测试，以确保内核被彻底“攻击”。在 eBPF 程序阻止了所有漏洞触发后，我们继续使用这台机器进行日常活动，如在 Google、YouTube、Twitter、Overleaf、Email 和其他互联网服务上浏览网页，参加 Zoom 会议，通过 Slack 发送消息，在 Spotify 上播放音乐，运行 Docker 容器进行 CTF 挑战，插拔外部显示器等。这个实验持续了 7×24 小时。

实验结果

给定一个内核漏洞，如果要认为 PET 和 undo workaround[3]能有效阻止该漏洞被攻击者触发，需要满足一下三个标准：❶ 当漏洞触发条件满足时能够成功防止漏洞触发，❷ 当漏洞触发条件不满足时确保系统正常执行，❸ 在阻止漏洞触发后内核保持稳定，不发生恐慌 (panic)。

测试用例的有效性结果如表 4-1 所示。根据实验结果分析，PET 对测试用例集中的所有漏洞都是有效的，除了由于缺乏 PoC 程序而无法评估的三个数据竞争。PET 准确地定位了所有类型的漏洞触发点，并在 5 分钟内有效地合成了触发条件和 BPF 程序来防止它们被攻击者触发。因此在内核开发人员修补这些漏洞的 17-344 天时间窗口期间，PET 可以为内核提供临时保护。

对于在栈、全局/静态区域的越界访问漏洞，PET 使用消毒器-原生内核映射器来准确获取合法边界，例如，栈数据对象的合法边界 `[$rsp+0x50, $rsp+0xa0)` 和全局数据对象的合法边界 `[0xffffffff822479c0, 0xffffffff822479c0+0x18)`。对于在堆的越界访问漏洞，PET 使用 BPF 辅助函数库里面的 `bpf_get_start` 和 `bpf_get_len` 来动态获取堆数据对象的合法边界。漏洞触发点的访问地址被保存在寄存器中，我们可以使用漏洞报告处理器和消毒器-原生内核映射器两个组件进行识别。

对于释放后使用类型漏洞，PET 在数据对象的释放点启用了扫描器，以固定周期扫描内核中是否存在悬空指针。扫描器用两种工作模式：完整扫描模式和选择性扫描模式，前者扫描了全部物理内存 (例如, be93025d[209])，后者只扫描可能包含悬空指针的 slab cache(例如，CVE-2022-2586 漏洞的悬空指针保存在 `kmalloc-256` cache 中数据对象的 `0x20` 偏移处)。在收集到的测试用例集共有 10 个释放后使用，其中 3 个使用了完整扫描模式，剩余 7 个使用了选择性扫描模式。关于两种扫描模式的性能开销对比在 §4.7.3。

对于未初始化访问漏洞，我们的评估显示保守政策比激进政策更有效，因为在执行被修改为不触发漏洞的 PoC 时，激进政策会报告误报。PET 默认使用保守方法，除非可以保证激进方法不会产生误报。

对于数据竞争，PET 识别了至少包含一次写操作且竞争相同内存的内核代码，并使用 P/V 操作有效检测到竞争。

除了漏洞触发点，PET 还为所有测试用例定位了成对操作，例如，在表 4-1 中的 b5b251b[205], CVE-2020-14386, 797c55d[207], be93025d[209], 和 CVE-2021-

CVE/SYZ ID	eBPF 程序装载位置	漏洞触发条件 & PET 防御策略	有效性 (PET/[31])	修复时间 (天)
整数溢出				
70c77ab[189]	`__qdisc_calculate_pkt_len`	`%eax>>%cl == %rax>>%cl & %cl<32`	●/◐	415
b5b251b[205]	`dummy_hub_control+0x3f` (spinlock) `dummy_hub_control+0x225`	`lock_map[pid] = %rdi` `%eax<<%edx==%rax<<%edx & %edx<32`	●/◐	79
栈上的越界访问				
2022-1015	`nft_do_chain+0x243`	`%rdi ∈ [%rsp+0x50, %rsp+0xa0)`	●/◐	147
2c09122[206]	`ethnl_parse_bitset+0x45f`	`%rdi+%rdx ∈ [%rdi, %rdi+0x40*8)`	●/◐	104
全局和静态区域的越界访问				
2017-18344	`show_timer+0x81`	`%rdx ∈ [0xffffffff822479c0, 0xffffffff822479d8)`	●/◐	90
buddy 堆上的越界访问				
2017-7308	`tpacket_rcv+0x2ff`	`%rdi+%rsi*%rcx ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	1015
2022-27666	`null_skcipher_crypt+0x4b`	`%rdi+%rdx ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	17
vmalloc 堆上的越界访问				
2020-14386	`tpacket_rcv+0x21a` (spinlock) `tpacket_rcv+0x6f6`	`lock_map[pid] = %rdi` `%rax ∈ [start(%r10), start(%r10)+len(%r10))`	●/◐	39
在 slab/slub 堆上的越界访问				
2010-2959	`bcm_sendmsg.cold+0x568`	`%rdi ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	13
2021-22555	`xt_compat_target_from_user.cold+0x23`	`%rdi+%rdx ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	86
2021-43276	`tipc_crypto_msg_rcv.cold+0x6d`	`%rdi+%rdx ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	12
2022-34918	`nft_set_elem_init+0x3e`	`%rdi+%rcx ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	38
2016-6187	`apparmor_setprocattr+0x8f`	`%rdi ∈ [start(%rdi), start(%rdi)+len(%rdi))`	●/◐	111
2017-7184	`xfrm_replay_advance+0x250`	`%rbx+0x18 ∈ [start(%rbx), start(%rbx)+len(%rbx))`	●/◐	108
2022-0185	`legacy_parse_param+0x27e`	`%rbp ∈ [start(%r12), start(%r12)+len(%r12))`	●/◐	0
797c55d[207]	`watch_queue_set_filter+0x81` (alloc) `watch_queue_set_filter+0x78d`	`alloc_map[pid]=%rdi` `%r15+0x8 ∈ [start(%r15), start(%r15)+len(%r15))`	●/◐	344
e4be308[208]	`sha512_final+0x34a/0x3e0`	`%r12+%rax ∈ [start(%r15), start(%r15)+len(%r15))`	●/◐	30
释放后使用				
2019-18683	`__vb2_queue_free+0x13e` (free) `vid_cap_buf_queue+0x49` (use)	`map ∪ %rdi; full_sweep(0, 16GB)` `%rbp+0x3a8 ∈ map ? true: false`	●/◐	29
2021-23134	`nfc_llcp_local_put` (free) `nfc_llcp_sock_unlink` (use)	`map ∪ %rdi; full_sweep(0, 16GB)` `%rdi ∈ map, ? true: false`	●/◐	201
2021-4154	`put_fs_context+0xec` (free) `filp_close` (use)	`map ∪ %rdi; full_sweep(0, 16GB)` `%rdi ∈ map ? true: false`	●/◐	233
2022-2586	`nft_obj_destroy+0x3f` (free) `nf_tables_fill_setelem.isra.0+0x140` (use)	`map ∪ %rdi; selective_sweep(kmalloc-256, 0x20)` `%rbx+%rax ∈ map ? true: false`	●/◐	97
2017-8824/ 2020-16119	`ccid_hc_rx_delete+0x2e` (free) `ccid_hc_rx_delete+0x2e` (use)	`map ∪ %rsi; selective_sweep(DCCPv6, 0x628)` `%rdi ∈ map ? true: false`	●/◐	128
2021-3715/ 5d5bb09c[190]	`__route4_delete_filter+0x3c` (free) `route4_get+0x58` (use)	`map ∪ %rdi, selective_sweep(kmalloc-192, 0x28)` `%rax+0x40 ∈ map ? true: false`	●/◐	59

be93025d[209]	`__vb2_queue_free+0x13e` (free)
	`vb2_mmap+0x52` (mutex)
	`vb2_mmap+0xa29` (use)
		`map ∪ %rdi; full_sweep(0, 16GB)`
		`mutex_lock[pid]=%rdi`
		`r8 ∈ map ? true: false`
			●/◐	73

2022-2588	`__route4_delete_filter+0x3c` (free) `__route4_delete_filter+0x3c` (use)	`map ∪ %rdi; selective_sweep(kmalloc-192, 0x28)` `%rdi ∈ map ? true : false`	●/◐	38
未初始化访问				
2039c557[191]	`__sys_recvfrom` (create) `tcp_recvmsg+0xb8` (use)	`map[%rsp+8-200] = mem(%rsp-0xc0, 0x60)` `map[%r13] == mem(%r13, 0x60)`	●(保守)/ ◐(激进)	248
e476b01d[210]	`__alloc_slab+0x237` (create) `simple_copy_to_iter+0x11` (use)	`map[%rdi] = mem(%rdi, 0x80)` `map[%rdi] == mem(%rdi, 0x80)`	●(保守)/ ◐(激进)	111
数据竞争				
2017-2636[203]	`n_hdlc_send_frames+0x118` (write) `n_hdlc_tty_ioctl+6b` (write)	`P(%rbp+0x310); V(%rbp+0x310)` `P(%rsp); V(%rsp)`	●/◐	40

2021-4083[204]	`unix_stream_read_generic+0xeb` (spinlock)
	`unix_stream_read_generic+0x120` (mutex)
	`unix_stream_read_generic+0x138` (read)
	`unix_gc+0x33` (spinlock)
	`unix_gc+0x28e` (write)
		`lock_map[pid]=%rdi`
		`mutex_lock[pid]=%rdi`
		`P(%r13); V(%r13)`
		`lock_map[pid]=%rdi`
		`P(%rsp+0x38); V(%rsp+0x38)`
			●/◐	224

f6e95af7[192]	`tcp_send_challenge_ack.constprop.0+0x5d` (read) `tcp_send_challenge_ack.constprop.0+0x7b` (write)	`P(%rip+0x2108765); V(%rip+0x2108765)` `P(%rip+0x2108765); V(%rip+0x2108765)`	N/A / ◐	178
cb2264a[211]	`tcp_send_challenge_ack.constprop.0+0x65` (read) `tcp_send_challenge_ack.constprop.0+0x7b` (write)	`P(%rip+0x2108765); V(%rip+0x2108765)` `P(%rip+0x2108765); V(%rip+0x2108765)`	N/A / ◐	340

a834b99[212]	`netlink_getname+0x44` (read)
	`netlink_insert+0x3c` (lock_sock)
	`netlink_insert+0x87` (write)
		`P(%rbp+0x310); V(%rbp+0x310]);`
		`lock_map[pid]=%rdi`
		`P(%rbp+0x310); V(%rbp+0x310]);`
			N/A / ◐	35

表 4-1 PET 有效性评估表，● 表示满足三个标准；◐ 表示标准❷未被满足 (即没有报错提醒)；N/A 表示 POC 程序无法获取

4083[204]。PET 采用检查点-恢复机制，在阻止触发后将内核恢复到正常状态。尽管这种方法有一定的局限性，正如我们在 §4.5.3 中讨论的，但在我们的评估中，在运行所有测试用例的 PoC 和利用程序 100 次后，内核稳定运行了 7×24 小时。我们继续使用它超过 3 个月，到目前为止，内核仍然正常运行。

相比之下，虽然相关工作 undo workaround[3]阻止了漏洞触发，但它不可避免地引发误报并影响正常执行，因为撤销操作总是无论条件是否满足都会执行。因此，根据我们定义的标准，undo workaround[3]被认为是部分有效的。

4.7.3 开销和可扩展性

本研究真实运行 PET 防御程序，并对比相关工作，回答以下问题以评估 PET 的性能、内存开销及可扩展：❶ PET 在多大程度上减慢了内核运行的速度？❷ eBPF 程序对内核执行引入了多少延迟？❸ 在释放后使用防御策略中，扫描器的最佳配置是什么？❹ PET 是否具备可扩展性能支持更多内核漏洞？❺ 支持 PET 运行需要多少额外内存?

实验设置

我们使用一系列基准测试 (benchmark)，包括 OSBench 用于测量基础操作系统操作的性能，如进程和线程创建，perfbench 用于测试调度器和 IPC 机制，以及一系列真实世界应用，如 MP3 编码用于计算密集型任务，SQLite 用于 IO 密集型任务，以及 Redis, perl-benchmark, Apache 和 Nginx 用于服务器负载。装载到内核中的 eBPF 程序出于性能考虑而启用了 JIT 编译。为了确定内核执行期间关键 eBPF 操作的延迟，我们在 BPF 防御策略的相关操作前后插入了两个 `rdtsc` 指令并收集了它们之间的时间差。

为确定释放后使用防御策略中扫描器最高效的完整扫描模式。我们进行一系列实验，变更了扫描间隔 (1 秒，2 秒，4 秒，8 秒) 和范围 (128 MB，256 MB，512 MB) 来识别部署的最佳配置。此外，我们比较了完整扫描模式和选择性扫描模式的性能，以展示我们的优化工作的有效性。我们随机选择了 5d5bb09c[190]作为选择性扫描的代表，CVE-2021-4154 作为全面扫描的代表，进行性能开销测试。

在存在多个内核漏洞的情况下，不同的 eBPF 程序需要同时安装。为了评估

PET 的可扩展性，我们的实验随机排序测试用例集中的漏洞对应的 eBPF 缓解程序，并在测试同时运行 2 个、4 个、8 个、16 个以及所有内核漏洞防御程序时的性能开销。

上述实验均自动运行多次，我们将实验结果标准差控制在 0.04 以内取平均值进行展示。实验运行的平台为 Ubuntu 22.04 LTS 系统，内核版本 v5.15，机器配置为 Intel(R) Core(TM) Intel i7-6700HQ @ 3.50GHz (4 Cores / 8Threads) CPU, 16GB RAM，1000GB SSD。

实验结果

表 4-2 展示了不同防御策略的性能开销作为问题❶的答案。PET 的平均性能开销在所有情况下都低于 3%，与相关工作 undo workaround[3]相当，除了 CVE-2020-14386，它是 vmalloc 区域的越界访问。这一例外是由网络相关基准测试的开销引起的，包括 WireGuard、Git、Apache 和 Nginx，它们的开销都超过了 10%。这是因为漏洞触发点位于内核函数 `tpacket_rcv` 中，该函数在网络栈中被大量用于从 NAPI NIC 接收数据包。即便如此，平均开销为 4.27%，远低于近年来提出的其他 Linux 内核保护措施 (例如，xMP[110])。Redis 的开销没有受到重大影响，因为在本文评估中，它以“standalone”模式运行，使用 loopback 而不是 NIC。

	整数溢出	**Slab 越界**	**Page 越界**	**栈越界**	**全局变量越界**	**Vmalloc 越界**	**释放后使用**		**未初始化访问**	**数据竞争**	
	b5b251b[205]	2021-34693	2022-27666	2c09122[206]	2017-18344	2020-14386	5d5bb09c[190]	2022-4154	2039c5[191]	2017-2636	2021-4083
操作系统核心操作											
OSBench	0.75%	0.01%	0.71%	0.38%	0.09%	4.21%	2.12%	3.05%	1.87%	-0.42%	-0.90%
perf-bench	0.61%	0.35%	0.03%	-0.18%	0.12%	6.06%	3.42%	5.86%	1.39%	-0.04%	1.71%
计算密集型											
OpenSSL	1.90%	0.03%	-0.07%	0.19%	0.19%	0.22%	1.24%	0.44%	2.07%	0.26%	-0.02%
GIMP	-0.46%	-1.13%	1.34%	-1.12%	-2.96%	0.88%	-0.17%	1.09%	0.67%	0.17%	1.36%
MP3 Encode	0.79%	0.19%	0.19%	0.95%	0.59%	1.42%	0.71%	1.59%	1.85%	-0.05%	0.02%
I/O 密集型											
SQLite	-1.86%	-0.71%	-0.20%	-0.39%	-1.50%	-1.22%	-0.01%	1.88%	1.62%	-1.11%	-0.12%
WireGuard	-0.85%	0.14%	0.05%	-0.19%	-0.47%	11.48%	1.06%	1.57%	-0.07%	0.77%	-2.90%
XZ Compress	-0.72%	0.66%	0.91%	0.03%	0.45%	1.66%	1.62%	2.29%	1.20%	0.09%	-0.64%
常规服务器任务											
Git	0.86%	0.07%	0.24%	0.39%	0.16%	14.51%	0.58%	0.47%	1.71%	0.26%	-0.43%
Kernel Compile	1.94%	-0.12%	0.10%	0.03%	0.25%	1.41%	2.15%	3.23%	2.80%	1.56%	0.95%
Apache	-1.86%	0.38%	0.38%	-1.14%	-0.39%	10.62%	4.11%	3.64%	0.56%	1.17%	1.58%
Nginx	1.14%	0.80%	-0.18%	0.23%	0.55%	10.27%	6.00%	5.32%	1.22%	0.20%	-1.10%
perl-benchmark	-0.42%	-0.39%	-0.02%	-1.21%	-0.48%	-1.10%	0.12%	2.97%	-0.39%	0.45%	0.55%
redis	1.72%	1.66%	-0.83%	0.49%	-1.69%	-0.60%	0.79%	2.97%	1.66%	0.51%	-1.23%
Average	0.25%	0.14%	0.19%	-0.11%	-0.36%	4.27%	1.70%	2.60%	1.30%	0.27%	-0.08%
undo workaround[3]											
Average	-0.12%	-0.70%	-0.42%	0.91%	0.91%	-0.03%	-0.36%	-0.28%	0.49%	0.22%	0.31%

表 4-2 PET 防御程序在应对不同类型漏洞时的性能开销

表 4-3 展示了执行各种防御策略中关键操作的 eBPF 程序引入的延迟，如问题❷所述。检查点和恢复、整数溢出的模拟、堆对象的动态边界确定、未初始化访问漏洞的存储和比较、数据竞争中的 P/V 操作以及释放后使用中的数据对象

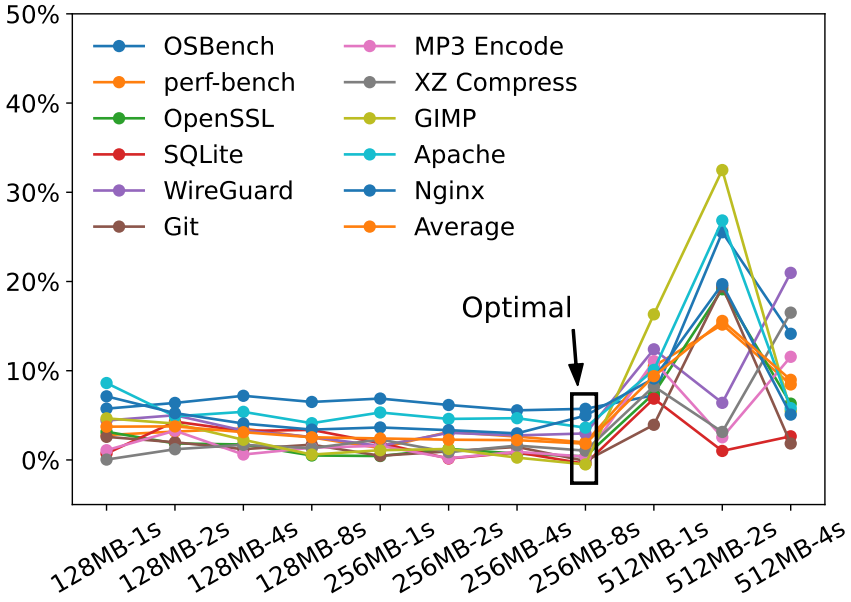


图 4-4 以 CVE-2021-4154 防御程序为测试平台，展示在释放后使用策略中全面扫描模式的配置影响，最佳配置是每 8 秒扫描 256MB

地址存储的延迟都低于 3500ns，远远超出人类感知范围。对于选择性扫描和完整扫描，它们的延迟都小于 300ms，并且在单个 CPU 上异步在后台执行。因此，如表 4-2 所示，无论扫描模式如何，释放后使用漏洞的总体开销都在 2% 左右。此外，小于 300ms 的延迟完全在可接受范围内，例如相关工作 Shuffler[213]。

	功能	时间
检查点-恢复	检查点	2702.72 ns
	恢复-触发条件未满足	2617.72 ns
	恢复-触发条件满足	2898.06 ns
整数溢出	模拟执行	1796.61 ns
越界访问	`bpf_get_start`	1935.16 ns
	`bpf_get_len`	1829.12 ns
未初始化访问	创建点保存未初始化对象内容	2820.52 ns
	访问点比较数据对象内容	3386.4 ns
数据竞争	P/V 操作	2608.23 ns
释放后使用	释放点保存数据对象内容	2806.69 ns
	选择性扫描模式 (异步)	13.96 ms
	完整扫描模式 (异步, 256MB/8s)	277.46 ms

表 4-3 eBPF 内核漏洞防御程序关键策略的延迟

此外，表 4-3 显示选择性扫描模式比完整扫描模式的最佳配置快 21 倍：13.96ms vs. 277.46ms，正如问题❸所关注的。完整扫描模式的最佳配置，即每 8 秒扫描 256MB 如图 4-4 所示。

图 4-5 展示了在存在多个内核漏洞的情况下 PET 的可扩展性，用于回答问题❹。如图所示，随着需要防御的内核漏洞增多，性能开销线性增加，这与相关工

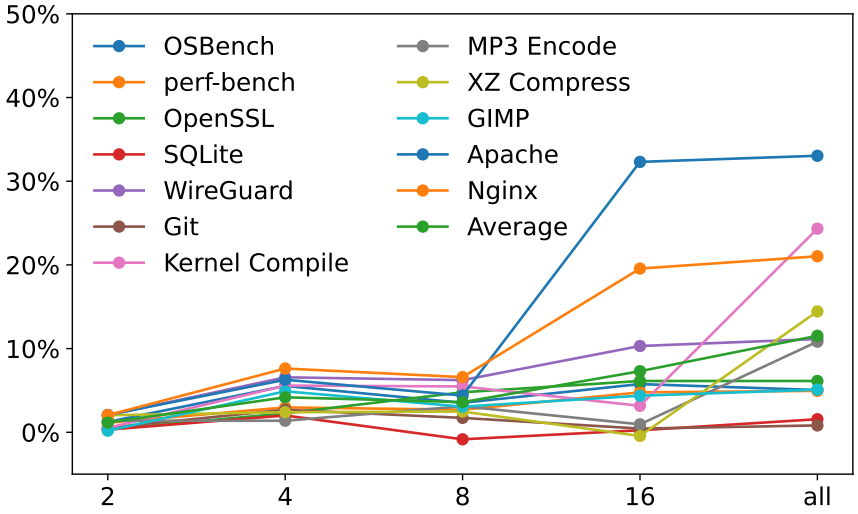


图 4-5　PET 的可扩展性，支持同时运行多个内核漏洞防御程序，Apache 和 Nginx 中的峰值是由 CVE-2020-14386 - vmalloc OOB 引起的

作 undo workaround[3]的结果相当。大多数基准测试的所有 34 个漏洞的总累计开销大约为 10%。Apache 和 Nginx 测试的开销显著增加，是由先前讨论的 vmalloc OOB 导致的。

对于问题❺，PET 使用的额外内存的主要来源是 BPF map。所有防御策略中最大的 BPF map 是用于隔离释放后使用类型漏洞的数据对象。它包括 2000 万条目，相当于 915MB。当存在多个释放后使用漏洞时，这个 map 由所有扫描器共享，从而降低对额外内存的需求。对于现代操作系统内核来说，随着 SDRAM 技术的进步这样的内存开销完全能够接受。

4.8　讨论

支持更多类型的漏洞。 PET 目前支持由消毒器报告的五个最常见的漏洞，但其具备强大的可扩展性可以防御更多类型的内核漏洞。例如，空指针解引用 (null-pointer dereference), PET 可以通过检查指针是否等于 NULL 来检测；野访问 (wild access) 和用户内存访问 (user-memory access)，PET 可以通过检查地址是否在内核空间中的合法地址来检测；以及内存泄露 (memory leak)，可以通过重用释放后使用扫描器来检查是否还有指针引用的某个数据对象来检测。我们计划在未来扩展 PET 以支持这些漏洞类型。

局限性。 PET 无法处理起始地址已超出范围的越界访问场景，因为 PET 依

赖于起始地址来确定合法对象大小。为了解决这一问题，PET 可能需要开发者手动调整漏洞触发条件。例如，一种可能的情况是内存访问指令的目标地址是从 addr+offset 计算得来的，并且 offset 非常大，PET 现有的 BPF 函数库 `bpf_get_start` 无法获得准确的数据对象起始地址。针对此种情况，开发者可以手动编写 eBPF 程序在运行时检查 addr+offset 是否正确。

4.9 小结

本章节提出了 PET，它能够在补丁可用之前防止已发现的内核漏洞被攻击者触发。本文验证了其对消毒器可以报告的五种最常见内核漏洞的有效性。在存在多个漏洞的情况下，PET 具有较强可扩展性，且性能、内存开销均可接受。

第五章　O2C: 基于 eBPF 和机器学习审计的实时内核分隔探索

5.1　引言

成功的内核漏洞攻击利用是一个多步骤的过程。从最初的内核漏洞开始，攻击者需要利用一系列的攻击利用技术来获得可利用的原语，并提升攻击者的权限，直到实现最终目标，如未授权访问敏感信息。不同于第三章 ERA 框架防止内核堆漏洞被利用，和第四章 PET 框架防止内核漏洞被触发，本研究聚焦于分隔化 (Compartmentalization) 技术，将内核的不可信部分隔离，防止最初的内核漏洞在漏洞利用后阶段转变为成功的攻击，破坏内核的其他部分。它将危害限制在有漏洞的组件内部，使得内核的其他部分继续正常运作。

安全研究者们已经为操作系统内核提出了多种隔离解决方案。值得注意的工作包括利用硬件特性的方案，如 NOOKS[83]，HAKC[14]，以及使用虚拟化技术构的工作，包括 HUKO[98]，LXD[93]，LVD[94]，和 KSplit[95]。此外，还有基于软件的方法，如 SFI[87]，XFI[99]，BGI[88] 和 LXFI[89]。

虽然这些解决方案可以实现分隔化目标，但启用它们需要系统重启，阻碍了其广泛部署。更进一步，安全事件通常是意外发生的[214]，很难预测哪些内核组件可能存在漏洞。因此，分隔化主要是在安全威胁公开后执行的。面对亟待解决的威胁，防御者必须关闭系统，配置 hypervisor/硬件或重新编译内核进行插桩，然后重启机器。这个过程不可避免地中断了机器上运行的关键服务，与此同时，由于 Syzkaller 每天平均报告 7.62 次不同的内核恐慌 (panic)[1]，需要频繁重启，加剧了内核服务中断。因此，一种不中断服务的、实时的分隔化技术有利于大规模部署。

鉴于 hypervisor 和硬件功能也可以在运行时配置，因此实现实时分隔化有可能被误认为仅仅是一个工程挑战。然而，本文强调即使分隔化的主要挑战不是在硬件或 hypervisor 功能选择之间的权衡，而是管理**状态切换风险 (Transition Hazard)**——属于不可信组件的内核对象在强制执行 (enforce) 分隔化前就存在，

且未被追踪。通过真实世界的攻击和对内核的采样分析，我们展示了这些数据对象对象可以被攻击者进行利用，忽视状态切换风险可能导致整个分隔化工作失效。

为了降低状态切换风险,本工作通过设计并实现 O2C(**O**n-the-fly **OS** **C**ompartmentalization) 探索了一种**审计 (auditing)** 解决方案。O2C 有两个关键技术创新。首先，我们展示了将机器学习模型嵌入到内核中并根据数据对象内容对数据对象类型进行分类的可行性，进而有效地支持了状态切换风险的审计。但内核缺乏对浮点计算的支持，并且对响应性和可靠性有很高的要求，因此该任务具有很强的挑战性。其次，我们介绍了动态插桩作为一种新方法来实现分隔化的强制执行，区别于依赖 hypervisor、硬件或静态插桩的传统技术。O2C 强化了现有 eBPF 生态系统的能力，来应对动态插桩实现分隔化带来的技术挑战，如创建和管理私有栈和堆，以及在主体切换中维持控制流完整性。

本文使用真实世界案例全面评估了 O2C。安全分析表明，O2C 可以有效防止不可信组件中的漏洞危及整个系统。我们比较了多个机器学习模型的性能，并验证了决策树是最适合 O2C 的模型，这得益于其在处理表格数据方面的优势、决策树的可解释性，以及其符合 eBPF 生态系统的约束。性能测试表明，O2C 仅对系能性能增加了 <4% 额外开销，并显示出优秀的可扩展性。对于分隔区的影响，O2C 能够与目前最先进的硬件辅助离线分隔化技术 HAKC 相匹配，并在不同条件下显示出优点和缺点。

根据我们对相关工作的了解和总结，O2C 是第一个研究实时分隔化的工作。本章节旨在通过以下贡献倡导更多这方面的工作：

1. 揭示**状态切换风险**作为实时内核隔离的主要挑战，并对此问题进行详细调查。
2. 开发 O2C 来审计状态切换风险，提出了两个关键技术创新：将机器学习模型嵌入到内核空间和强化现有 eBPF 生态实现基于动态插桩的分隔化。
3. 使用真实世界案例和多个基准测试全面评估 O2C，验证设计的有效性和高效性。

5.2 背景

5.2.1 内核分隔化技术

宏内核架构的操作系统内核，例如 linux/FreeBSD，所有的内核组件，包括内核函数、内核模块和第三方内核驱动都共享了同一个内核地址空间。一旦攻击者成功攻击利用了一个漏洞就可以导致整个内核被破坏。因此已经有研究者实践了最小特权原则 (principle of least privilege)[14,100]将不可信的内核组件与内核其他部分进行分隔，分隔的粒度可以从整个设备驱动程序到单个文件和函数不等。分隔化 (compartmentalization) 的核心目标是限制不可信内核组件的资源访问，将可能由不可信内核组件漏洞带来的破坏限制在分隔区内，不影响内核其余部分的正常运行。一个典型的分隔化设计包括访问控制策略和强制执行机制。

策略

分隔化的策略规定了分隔区主体对分隔区和内核的代码区、数据区、堆、栈等客体的访问控制权限。通常，内核具有最高权限，允许读写任何内存区域和访问任何资源。相比之下，分隔区的权限被降低，仅限于使用维持其功能所必需的最小一组内存和资源。为防止分隔区危及内核的任何其他部分，需要确保三个属性：❶ 控制流完整性，分隔区内的控制流不能被劫持到未授权的目标代码。为了保证这一属性，分隔化策略需要验证间接调用和跳转的目标地址以及栈的返回地址是否合法。❷ 数据完整性，分隔区只允许访问其自身拥有或与内核共有的数据。为了维护这一属性，分隔化策略必须审查隔离区内的每次内存访问，特别是写操作，确保被访问的对象位于正确的区域并且类型正确。❸ 合法的参数和返回值，分隔区被禁止传递恶意参数或向内核的其他组件返回恶意值，因为这可能导致混淆代理 (confused deputy) 攻击[101]和 Iago 攻击[215]。

机制

分隔化需要依赖运行时机制来执行分隔化的访问控制策略，运行时机制可以是基于硬件[81-82]的或基于 hypervisor 的[77-78]。正如 §2.3 所介绍的，代表性工作包括使用多套内核页表的 NOOKS[83]和 SKEE[84]，使用 PIPE 硬件架构的 SCALPEL[91]，使用 ARM 架构 PAC 和 MTE 硬件机制的 HAKC[14]，以及设计用于执行分隔化策略的特殊 CPU 机制的 SecureCells[90]。除此之外，HUKO[98]使用

了 Hypervisor 的 EPT 页表来分隔不可信内核模块, LXD[93], LVD[94], 和 KSplit[95] 更进一步使用 `vmfunc` 降低性能开销。

运行时机制同样可以基于软件实现，即通过在编译时插桩所有内存访问指令, 检查指令的目标地址是否在分隔区可访问的合法区域。之前的相关工作包括 SFI[87]和 XFI[99], 后续的 BGI[88]使用了影子内存来管理访问控制策略, LXFI[89]引入了一种 IDL 维护分隔区和内核交互接口的完整性 (API Integrity)。由于内存区域需要预先划分，指令插桩需要在编译时进行，不可信内核组件的分隔化只能在内核启动时开始，关机时结束。

5.2.2 现有内核分隔化技术局限性

尽管操作系统内核分隔化技术多种多样，但启用内核分隔化技术需要重启系统，这一局限性阻止它们产生最佳的保护效果。安全事件通常突然发生，威胁不断变化，系统管理员几乎无法预测哪个内核组件将来可能存在漏洞，并提前对其进行分隔化。本文对最近 100 个 Linux 内核漏洞的研究显示，这些漏洞分布在 23 个单独的子系统中，范围从网络中的 Netfiler、文件系统的 JFS、异步 I/O 的 io_uring，再到 usb 驱动等等。因此，分隔化通常在安全威胁暴露后才被执行，系统管理员需要关闭系统，配置 hypervisor/硬件或为静态插桩重新编译内核，最后重启机器。这一过程不可避免地会中断机器上运行的关键服务。

使情况变得更糟的是，由内核漏洞带来的服务中断可能非常频繁。从 2021 年到 2023 年，Linux 内核平均每 3.98 天就会被分配一个 CVE ID[2]。如果我们考虑到那些被报告和修补但没有被分配 CVE ID 的漏洞，漏洞披露的频率甚至更高，例如 Syzkaller 每天报告 7.62 次不同的内核恐慌 (panic)[1]。如果采用现有的分隔化技术被广泛部署，机器在一天内可能需要多次重启。这种频繁的服务中断是不可接受的，特别是考虑到许多现代应用程序，如机器学习训练，通常需要几小时才能完成，在软件测试中，模糊测试的连续性对于状态积累至关重要。即使在云环境中，任务可以在机器之间迁移，增加的带宽消耗和调度的复杂性也呈现出重大挑战。

因此，能够不中断服务和重启机器的实时内核分隔化技术更有利于实现分隔化技术的部署。

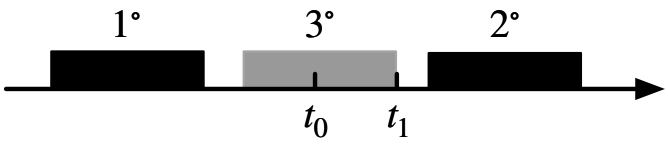


图 5-1 当在 t_0 时刻执行分隔化时，数据对象生命周期的三种情况

5.3 实时内核分隔化的状态切换风险

有观点认为实现实时分隔化仅需要工程努力。以 HUKO[98]为例，它通过 EPT 将特定子系统与其他部分隔离开来，这些 EPT 可以在运行时设置和修改[216-217]。然而，实现实时分隔化的挑战并不在于选择硬件或 hypervisor 特性构造运行时分隔化机制，相反，主要的挑战在于管理**状态切换风险**。本文首先详细介绍了**状态切换风险**的基本原理，并结合真实世界漏洞和更深入的静态分析结果揭示其内在风险。

5.3.1 状态切换风险

图 5-1 展示了在 t_0 时刻分隔化开始执行时，属于不可信内核组件的数据对象的三种情况。情况 1°，数据对象生命周期在 t_0 时刻前终止，分隔化的执行无需处理这些数据对象。情况 2°，数据对象生命周期在 t_0 时刻后开始，分隔化可以将它们隔离在一个独立的内存区域，或通过指令插桩很好地追踪和管理。

棘手的部分是如何处理情况 3° 状态切换阶段，即数据对象的生命周期跨越了 t_0 时刻，并最终在 t_1 时刻终止。这些数据对象既不存储在分隔的内存区域中，也不被跟踪。如果分隔化将所有不可信组件对它们的访问视为违反数据完整性，内核无法继续执行，因为某些数据对象合法地属于该组件。相反，如果所有访问都被允许，攻击者可以利用不可信组件内的漏洞突破到分隔区之外，使分隔化无效。

图 5-2 展示了在没有额外工作负载的情况下，内核版本 v6.1 中数据对象生命周期在 20 分钟内的分布情况。该图体现了一个 “长尾效应”：尽管大多数数据对象的生命周期在一秒内终止，但相当数量的数据对象 (总共 10,862 个) 展示出生命周期超过 10 秒。这个分析结果说明，情况 3° 中的对象数量不可忽视，从 t_0 到 t_1(所有对象终止时) 的过渡阶段相当长。

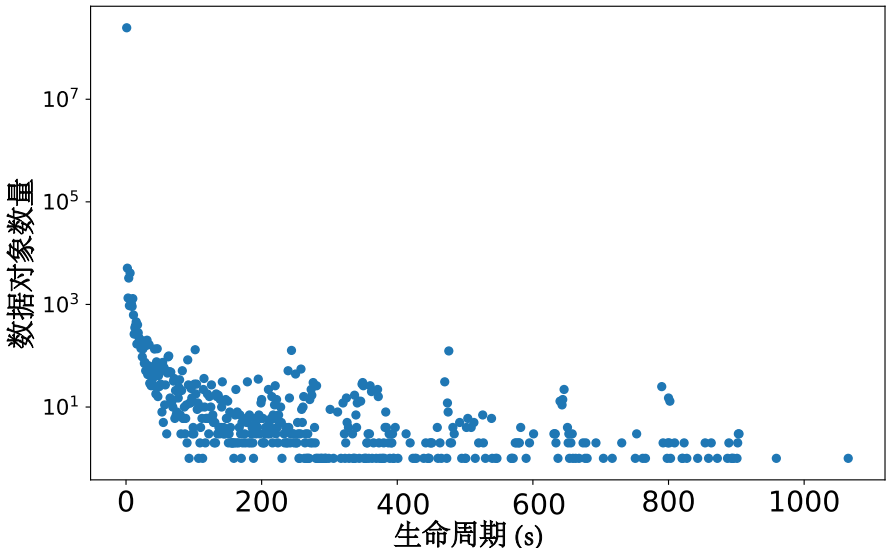


图 5-2　数据对象生命周期的分布，采样时间为 20 分钟

5.3.2　攻击实例

为了更深入阐明攻击者可以利用状态切换风险绕过现有的分隔化技术，我们选择了 CVE-2022-0995[218](即 bonfee) 公开的攻击利用代码作为研究实例。类似的 CVE-2022-0185[219], CVE-2022–32250[220], CVE-2022-27666[221]等漏洞可以被改造用来协助状态切换风险绕过分隔化技术。

CVE-2022-0995 漏洞是在 watch_queue 事件通知子系统中发现的，该子系统通过 `CONFIG_WATCH_QUEUE` 配置选项启用。它提供了一个从类型为 `struct watch_filter` 的漏洞数据对象开始的 slab 越界写入能力。公开的攻击利用代码通过在 t_0 之后主动分配漏洞数据对象，然后执行堆喷射[132]，使攻击负载数据对象 `struct msg_msg` 紧邻漏洞数据对象 `struct watch_filter`。攻击利用代码可以在此布局下触发越界写入来修改 `msg_msg->m_list.next` 指针，进而泄露内核基址以绕过 KASLR 并劫持控制流。但这种攻击利用代码不能绕过分隔化，因为漏洞数据对象 `struct watch_filter` 在一个单独的区域中被隔离或被基于插桩分隔化机制追踪和管理。因此，任何试图突破边界以篡改 `struct msg_msg` 的尝试都被阻止，如图 5-3 所示。

然而，通过利用状态切换风险，攻击者即使在 t_0 分隔化被执行之后也能利用上文中提到的攻击利用代码攻击内核。因为攻击者可以利用在 t_0 时刻之前已经分配且尚未释放 (即，情况 3°) 的数据对象实现攻击。当 t_0 时刻执行分隔化之后，watch_queue 子系统的运行在分隔区之中，但是已经存在的数据对象在分隔

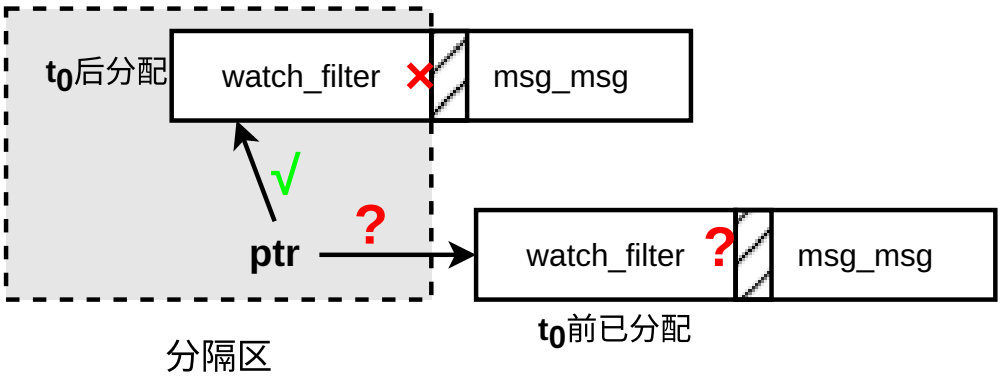


图 5-3 攻击者利用状态切换风险触发 CVE-2022-0995 漏洞[218]，成功绕过分隔化。现有的分隔化机制无法判断未被追踪的 `struct watch_filter` 和 `msg_msg` 是否可以被分隔区对象访问

区之外且未被分隔化机制追踪和管理，如图 5-3所示，此时攻击利用代码通过一个指针解引用分隔区之外的漏洞数据对象 `struct watch_filter`，触发越界写漏洞破坏 `struct msg_msg` 数据对象达成攻击目标。在此期间，分隔化机制能够观察到分隔区之内的代码访问了分隔区之外的一块内存，但由于没有追踪记录，无从判断该内存是否可以合法访问。

5.3.3 频繁被修改的数据对象

防止上文中攻击实例的关键在于识别出数据对象的类型，进而区分合法和非法访问的数据对象。对于 `struct watch_filter`，根据最小特权原则，它属于 watch_queue 子系统的合法访问范围，无论该数据对象是否在分隔区内，watch_queue 子系统的代码都可以合法访问。而 `struct msg_msg` 数据对象不属于 watch_queue 子系统的合法访问范围，对此类数据对象的访问违反了数据完整性。

不幸的是，根据我们对情况 3° 中数据对象的深入研究，数据对象类型的区分是非常有挑战性的。通过结合静态和动态分析我们发现内核中存在 35,012 种不同类型的数据对象，这些数据对象在分配时会被初始化为 0，在整个生命周期中平均会被修改 22.87 次。数据对象内容的多样性和变化性，使得我们很难手动通过数据对象内容的特征识别出对应的类型。因此，最先进的 Gibraltar[222]只能推断出具有特殊特性 (如链表) 的数据对象类型，并且缺乏支持多种数据对象类型的通用性。

5.4　概述

我们设计并实现了 O2C，一种审计状态切换风险的探索性工作，努力为实现实时分隔化铺平道路。O2C 首先通过动态插桩技术构建了分隔化技术所需的运行时机制，并提供了常见的分隔化访问控制策略。在此基础上，O2C 加入了一个基于机器学习的审计方案，能够区分不可信组件访问的数据对象类型，审计在状态切换阶段发生的攻击利用尝试，促进离线的诊断和取证。

5.4.1　安全模型

本章节的安全模型与相关工作类似，假设某个不可信内核组件因为存在漏洞需要被隔离到分隔区中，除了该组件的内核作为一个可信的整体。分隔化运行时机制和策略是由特权用户进行管理的，例如系统管理员。攻击者作为非特权用户，无权关闭或修改系统管理员配置的分隔区。

攻击者的目标是利用分隔区内的漏洞来侵犯内核的其他部分，并最终提升权限。他们被允许使用最新的利用技术[132,142]，但硬件侧信道和物理攻击除外，因为这些超出了内核层面分隔化的范围。

防御者对不可信的组件强制执行分隔化，例如低质量的第三方设备驱动程序，而不是核心内核代码。通过坚持最小权限原则，防御者旨在保护内核的其他部分免受分隔区内漏洞带来的潜在威胁，保持系统服务。特别的，在实时隔离的过渡阶段，防御者应审计尽可能多的异常操作，以便离线诊断和取证。

	内核			**分隔区**		
	read	**write**	**exec**	**read**	**write**	**exec**
内核代码	√		√	√		
内核数据	√	√		√		
内核堆	√	√		√		
内核栈	√	√		√		
O2C	√	√	√	√		
分隔区代码	√	√	√	√		√
分隔区数据	√	√		√	√	
分隔区堆	√	√		√	√	
分隔区栈	√	√		√	√	

表 5-1　内核和不可信分隔区强制执行的访问控制表。√ 代表当前主体有权限读、写、执行相关客体

表 5-1 展示了 O2C 强制执行的访问控制策略。在这一策略中，分隔区被剥

夺了写入内核数据、堆和栈的权限。任何可能与内核交互的操作，如函数调用和内存写入，必须经过 O2C 的严格检查。请注意，O2C 允许分隔区读取内核，因为攻击者仅通过外部读取不能提升权限。这也解释了在许多先前的工作中，内存读取被广泛认为是低风险的[87-89,99,223]。

5.4.2 工作流程

如 §5.3 所讨论，缓解状态切换风险的关键在于识别不可信组件访问的数据对象类型，并确定这些类型是否为分隔区允许的合法类型。为实现此目的，O2C 在内核中嵌入了一个机器学习模型，以根据数据对象内容分类数据对象类型。机器学习模型通过利用 eBPF 生态系统动态地嵌入到内核中。根据模型的输出，O2C 审计对未被追踪数据对象的访问。此外，O2C 使用 BPF map 和辅助函数构建私有栈和堆，从而隔离在 t_0 时候部署分隔化后开始其生命周期的对象，实现完整的分隔化。

两阶段工作流程

O2C 分为两个阶段工作：阶段 0 用于准备，阶段 1 用于审计和分隔化，如图 5-4。

在阶段 0，内核像往常一样运行，不存在攻击者正在进行攻击利用。代码分析器分析内核源代码及其运行时二进制文件，识别需要审计和部署分隔化运行时检查机制的指令。数据对象分析器收集并标记内核数据对象的内容和类型，以便 O2C 可以训练机器学习模型。所有审计和部署分隔化运行时检查被整合成一组 eBPF 程序，用于阶段 1 部署分隔化。

从 t_0 开始的阶段 1，O2C 将 eBPF 程序应用于阶段 0 由代码分析器识别的指令，审计不可信组件的访问，为新数据对象的分配创建私有栈和堆，控制主体转换，并验证其他分隔化属性，如控制流完整性。值得注意的是，在检测到审计中的异常时，O2C 记录必要的运行时信息以供离线诊断和取证。我们的实验表明，审计是轻量级的。因此，O2C 永远不会停止审计，尽管未跟踪的对象最终将在未来的某个时候终止。

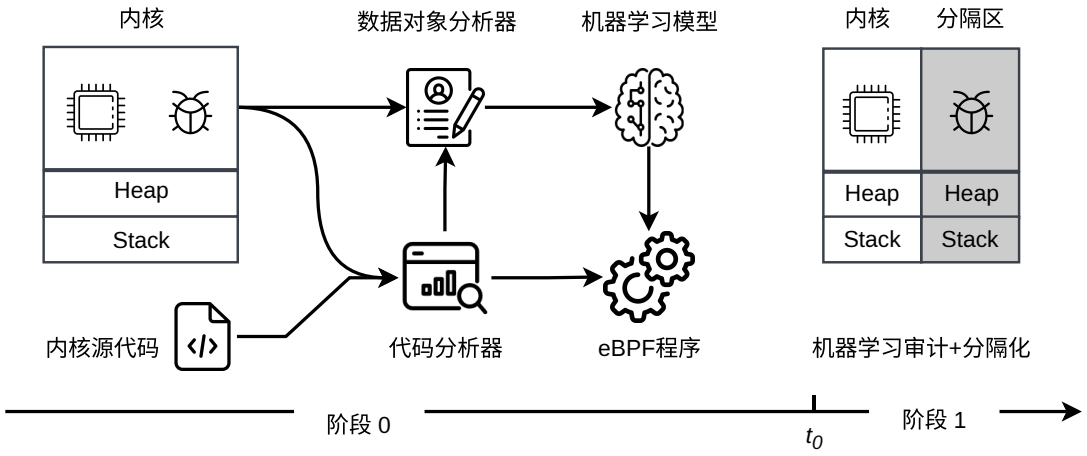


图 5-4　O2C 的两阶段工作流程

5.5　阶段 0 的代码分析器

O2C 对三类指令进行了动态插装，以强制执行审计和分隔化。在本节中，我们将描述代码分析器是如何识别这些指令的。

5.5.1　识别强制执行机制相关指令

间接跳转指令

第一类强制执行机制相关指令与间接控制流跳转有关，包括间接调用、间接跳转和函数返回。在阶段 1，O2C 将仔细检查这些指令的目标地址，以确保它们符合不可信组件的合法控制流图。为此，在阶段 0 识别这些指令时，代码分析器不仅记录它们自己的地址，格式为 `func+offset`，还标记了目标地址存储的位置。例如，对于指令 `call *%rax`，代码分析器记录目标地址可以从`%rax` 寄存器中检索，对于 `jmp -0x7dabaac0(,%rax,8)`，目标地址位于 `(-0x7dabaac0+%rax*8)` 内存单元中。特别是对于 `ret` 指令，目标地址在 x86 架构中隐式放在`%rbp+0x10` 中，在 ARM 中放在`%r14` 寄存器中。

写内存指令

第二类指令涉及内存写入，包括移动、存储、加载、比较和算术操作，以及操作内核栈的 push/pop 指令。它们的源操作数或目标操作数涉及内存。

与第一类相似，代码分析器记录指令地址和被写入的内存地址。例如，在指令 `mov 0x29c23e4(%rip),%eax` 中，写入的地址是`%rip+0x29c23e4`，这表示一个

全局变量。当栈被写入时，地址是根据`%rbp/%rsp`寄存器计算的，因为它们分别代表栈的底部和顶部。

在阶段 1 中，O2C 将通过检查允许写入数据对象的合法类型和实际类型之间是否匹配，来审计状态切换风险。因此，代码分析器进一步利用在[224-225]中开发的数据对象别名静态分析来识别对象类型。对于没有定义类型的普通数组，代码分析器用唯一 ID 作为类型进行注释。

为了支持私有堆，代码分析器还识别堆分配 `call` 指令，以便 O2C 可以在阶段 1 拦截分配请求并从私有堆中分配内存。

主体切换

最后一类与主体切换相关，当分隔区调用内核其余部分的函数时，或内核进入分隔区内时，就会发生主体切换。一方面，由于分隔区不可信，O2C 需要防止它向外部内核函数传递恶意参数或返回恶意值，避免内核被攻击。另一方面，分隔区在其自己的私有堆和栈上操作，确保任何内部崩溃不会向外扩散。因此，随着主体的切换，O2C 也对应切换栈和堆。代码分析器识别可以从外部调用的分隔区内的函数，以及分隔区内调用分隔区外的内核函数的指令。当内核第一次调用分隔区函数时，O2C 创建一个私有栈。

5.5.2 性能优化

O2C 采用了多种优化措施以保持轻量级。这些优化导致需要插桩的指令数量总体减少了 24.07%。

跳过确定性地址

访问全局内核变量的指令通常呈现为 `mov offset(%rip), %rax`。因此，当指令执行时，访问的地址总是固定的。这种确定性减少了 O2C 对全局变量访问指令进行插桩的需要，占总插桩数量的 1.21%。

类似的优化也可以应用于访问栈变量的指令。在函数序言 (prologue) 中，函数的栈帧通过 `sub offset1, %rsp` 创建，且在函数的生命周期内`%rbp/%rsp` 的值保持不变。因此，像 `mov x, offset2(%rbp/%rsp)` 这样从`%rbp/%rsp` 有固定偏移量的指令一定是访问当前函数的栈，无需进行额外的运行时检查[226]。然而，在一些特殊情况下，如`%rsp + %rax * offset3`，被访问地址取决于运行时的寄存

器值，O2C 保留检查以确保访问部分位于私有栈内。这种优化导致总插桩数减少了 12.39%，栈访问插桩数减少了 99.32%。

消除冗余检查

由于局部性原理，顺序访问的内存地址往往是邻近的，共享相同的基地址但在偏移量上有所不同。这在数组索引和对数据结构成员的操作中经常出现。我们将多个访问检查进行合并，使用 eBPF 程序检查最小和最大偏移量——代表数据对象的两个端点。

另一个冗余检查也与栈访问有关。因为返回地址的覆写通常是由栈溢出引起的，并且栈访问已经经过严格审查，因此为返回地址添加额外的完整性检查变得多余。我们可以安全地消除这类插桩，减少 10.47% 的插桩数。

5.6　阶段 0 的机器学习模型训练

在本章我们将介绍训练机器学习模型的细节。模型学习内核对象类型中的不变量。在运行时，给定一个访问位置，模型提取被访问堆数据对象的内容并对其类型进行分类。如果分类的类型与数据对象分析器的静态分析结果不匹配，O2C 会审查这种恶意活动以供离线诊断和取证。在本节中，我们将讨论哪种机器学习模型合适数据对象分类任务，以及如何在内核环境中嵌入机器学习模型。

5.6.1　训练数据收集

在 O2C 中，数据对象分析器使用一组 eBPF 程序来提取堆数据对象的内容作为训练数据。在每个分配点，O2C 用一个 eBPF 程序记录分配数据对象的地址，和分配点对应的调用路径记录到一个 BPF map 中。在每个释放点，O2C 用另一个 eBPF 程序来获取被释放数据对象的内容，并通过查找 BPF map 找到对应的数据对象分配时的调用路径。在线下分析时，数据对象分析器根据其分配的调用路径确定对象的类型。我们选择在数据对象的释放点而不是其他访问点获取数据对象的内容，因为内核代码在数据对象的生命周期内不断对其进行更改，数据对象的内容也逐渐积累了特定于其类型的值，使不同类型的数据对象越来越有区分性。使用在释放点的内容进行训练能产生最高的准确性。为了确保有足够的训练数据，收集不仅限于要被分隔的组件，而是包含整个内核。此外，我

们定制了模糊测试 (fuzzing) 模板以尽可能覆盖更多对象类型。

5.6.2 决策树模型

可用于基于对象内容分类对象类型的机器学习模型种类繁多，如支持向量机和神经网络等。但在众多选项中，我们基于以下见解选择决策树作为我们的默认模型。

首先，与最先进的深度学习方法相比，决策树模型在处理表格型数据方面有明显的优势[227]。我们在 §5.9.3 将决策树模型与其他模型进行了比较，验证了它对 O2C 特定任务的有效性。其次，决策树模型是可解释的，并且具有确定的分类时间[228]，使其足够可靠，可以用于对可靠性和快速响应要求极高的内核。第三，如 §2.4 所讨论的，当前的 eBPF 技术缺乏对浮点计算和动态堆的支持，对最大指令数和栈大小的也存在较大限制。因此随机森林[229]或 XGboost[230]等复杂的树集成机器学习模型很难通过目前的 eBPF 技术进行实现。在 §5.10 中，我们将讨论扩展 eBPF 生态系统以包含更多机器学习模型的计划。

在训练模型时，考虑到一些对象的大小可达两个页面，我们将对象内容分割成 8 字节数组来形成特征向量，而不是对完整的内容进行训练。虽然这种方法增加了输入维度，但决策树擅长处理高维数据。此外，为了处理对象大小的变化，我们通过填充零值来标准化它们的长度。

为了将模型嵌入到 eBPF 程序中，我们利用 scikit-learn[231] 将模型转换为五个数组：childrenLeft、childrenRight、feature、threshold 和 value。childrenLeft 和 childrenRight 数组构建树结构。feature 数组指定在树节点使用的特征向量中的哪个 8 字节单元。threshold 数组表示树节点的阈值，能够决定接下来选择左子树或右子树。值得注意的是，eBPF 程序不支持浮点计算，决策树模型在绕过这一限制方面显示出独特的优势。受到先前工作[232]的启发，我们将浮点阈值向下取整为整数而不失去精度。这是因为决策树中的所有比较都是 $\leqslant$ 操作，且我们模型的特征向量中的所有 8 字节单元都是整数。最后，value 数组在叶节点存储分类输出，即数据对象类型。这五个数组被加载到 BPF map 中，这些 map 在所有 eBPF 程序之间共享。这些程序执行树遍历逻辑，并确定分类类型是否与代码分析器的结果匹配。

除了克服浮点计算的限制外，我们还解决了 BPF 栈大小有限和缺乏对动态

堆支持的问题。为了使用决策树模型，eBPF 程序需要提取堆数据对象的内容。然而，某些数据对象对象大于 eBPF 分配的栈空间，当前的 eBPF 生态系统中也不存在动态堆。为了解决这个问题，我们在指定的 BPF map 中预留了一个巨大的池，并将运行时内容存储在 map 中。通过一个指向池中数据对象的指针，我们读取特征向量并将其输入到决策树中进行类型分类。

5.7 阶段 1 的审计和分隔化强制执行

5.7.1 审计状态切换风险

在阶段 0，O2C 已经识别出所有的访问指令，以及部署审计操作的释放点。此外，O2C 还训练了一个机器学习模型来根据数据对象的内容对数据对象类型进行分类。在阶段 1 中 O2C 配置了将机器学习模型载入到这些访问点的 eBPF 程序，审计异常情况——合法的数据对象类型和实际访问的数据对象类型之间的不匹配。

5.7.2 完整的分隔化强制执行

除了审计外，O2C 还包含了基于动态插桩的完整分隔化。图 5-5 总结了内核、eBPF 程序和分隔区之间的交互协议。当内核需要请求在分隔区中的其他组件的服务时，内核直接调用分隔区中隔离的代码，分隔区中的内核组件处理服务请求并将结果返回给内核。在分隔区中的内核组件处理服务请求时，可能也会调用部分合法的内核函数、访问部分合法的内核数据协助完成服务。eBPF 程序在发生内核-分隔区和分隔区-内核主体切换时，都会提前更换私有或内核的堆和栈，同时检查目标函数、内存访问地址、参数、返回值是否合法。接下来我们展开介绍。

控制流完整性

O2C 通过将 eBPF 程序应用于检查间接跳转类型的指令来保证控制流完整性。更具体地说，代码分析器采用目前最先进的静态分析技术[105,107]，获得一个可靠且足够精确的控制流图。对于每一个间接调用、间接跳转和返回指令，这个控制流图都提供了一组合法目标。O2C 使用一个 BPF map 来存储这些信息，将

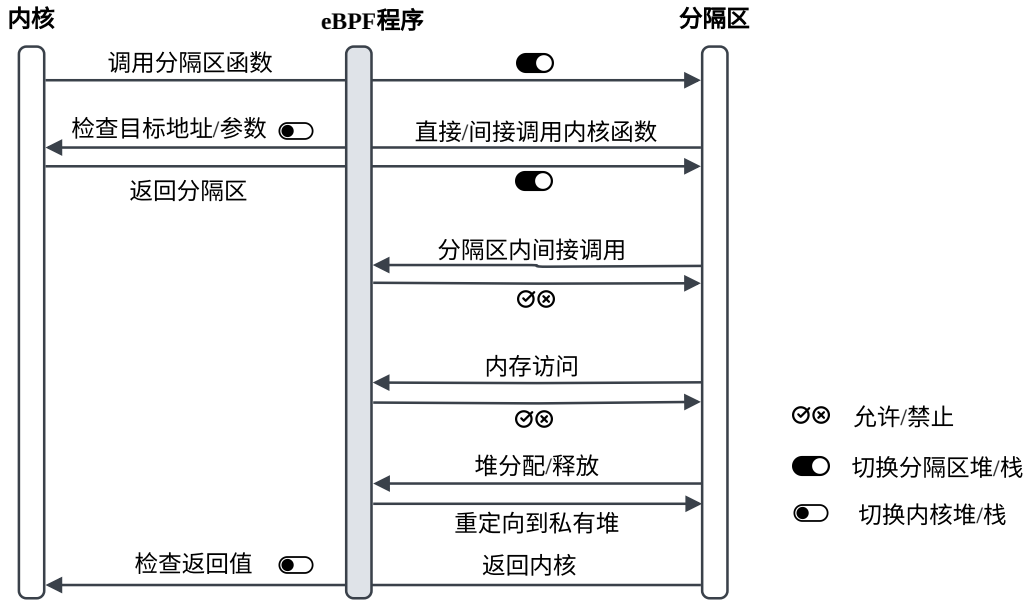


图 5-5 内核-eBPF 程序-分隔区的交互协议

指令地址设为 key，目标数组作为 value。这个 map 在所有用于实现控制流完整性的 eBPF 程序之间共享。通过检索运行时目标地址，例如间接调用 `call *%rax` 中的`%rax` 寄存器，eBPF 程序将其与 map 中存储的目标地址进行比较，确定执行跳转是否违反了控制流完整性。

私有栈和堆

O2C 为保证数据完整性的强制执行创建了私有栈和堆。这些私有内存区域通过共享的 BPF map 维护。

当不可信的组件在 t_0 之后首次执行时，安装到函数入口的 eBPF 程序通过从管理整个计算机物理内存的 buddy 分配器获取内存来创建一个私有栈。栈的地址记录在 BPF map 中，以便检索和重用。

同样地，为了管理私有堆，O2C 将 eBPF 程序部署到调用分配和释放数据对象的指令上，将内存的分配和释放重定向到分隔区的内部分配器上。在分配点，根据需要的内存类型的不同，eBPF 程序采取不同的行动。如果需要多个物理连续的页面，eBPF 程序将分配请求转向 buddy 分配器，就像创建私有栈一样。如果分配了小于一页的对象，eBPF 程序将分配请求转向 slab/slub 分配器。在 slab/slub 分配器内部，内存以 slab cache 的单位组织。一个 slab cache 被划分为统一大小的槽，以便同一个 cache 中的对象属于同一类型或有相同大小。当第一次为特定类型的对象分配内存时，eBPF 程序首先创建一个私有 slab cache，并从私有 cache

中获取一个槽来保存该数据对象。私有 cache 的地址记录在共享的 BPF map 中，以便后续的分配可以从中进行。

对于同时属于分隔区和内核共有的对象，不违反访问控制策略的情况下，eBPF 程序将持有该对象的 cache 视为私有。如果需要大块的虚拟地址连续内存，eBPF 程序将请求转向 vmalloc 分配器。在释放点同理，O2C 根据数据对象的分配器类型，将内存释放请求转向对应的 buddy、slab 或 vmalloc 的释放函数。

数据完整性

通过将 eBPF 程序挂载到与内存访问和主体切换有关的指令中，O2C 能够限制内存访问并强制执行数据完整性检查。

如果阶段 0 的代码分析器确定访问的目标地址在私有栈内，那么在运行时的任何非私有栈地址的访问都表明发生了恶意事件。如果访问的地址落在私有堆区域内，O2C 直接查询一个共享的 BPF map，该 map 记录了在 t_0 之后分配的所有堆数据对象以验证该数据对象类型。对于由 buddy 分配的数据对象，共享 BPF map 记录其分配位置，该位置充当类型标识符。对于处于 slab cache 中的数据对象，O2C 查找共享 BPF map 以获取该数据对象可以被合法分配的 slab cache 集合。从这个集合中的 slab cache 中，O2C 提取类型信息进行比较。对于由 vmalloc 分配器分配的数据对象，共享 BPF map 存储相应的 `struct vm_struct` 的地址，也可以用于标记类型。上述检查也适用于与内核共有的数据对象。

如果访问超出了分隔区的边界，O2C 依赖审计来确定访问的对象是否有效，如 §5.7.1 所述。

合法的参数和返回值

为了禁止混淆代理攻击[101]和 Iago 攻击[215]，O2C 支持对从分隔区流向内核的参数和返回值进行运行时检查。由于接口的复杂性和多样性，参数和返回值的合法值范围可能有很大不同。例如，文件打开操作的预期返回值应该是 0 表示成功，或`-ERROR CODE` 表示失败，而文件读取的返回值应该是读取的字节数。

之前的分隔化工作，如 LXFI[89], LVD[94], 以及 KSplit[95]，需要使用特殊的 IDL(interface defnition languages) 标记这些检查。O2C 在这些努力的基础上，提供了一种更直接的方法——在 eBPF 程序中表达检查。

5.8 实现

实现 O2C 的原型包含了 324 行 C 代码的 eBPF 程序模板，260 行 C 代码的内核中新的辅助函数，以及 1371 行 Python 代码用于识别执行指令、机器学习训练和框架集成。此外，还有 8553 行 C++ 代码用于静态分析。O2C 在开源的 Linux 上实现了原型系统。由于 eBPF 在不同平台上的设计一致性，O2C 也具备迁移到其他开源操作系统内核的潜力。

识别强制执行相关指令

我们使用 Ghidra[233] 来反汇编内核镜像。作为一款成熟的逆向工程工具，Ghidra 能够提供全方位的分析结果，包括段、函数边界、交叉引用、变量识别等。

我们遍历了分隔区内的所有函数，并使用 Ghidra 提供的 `CodeUnits` 迭代器和每条指令的指令助记符 (mnemonic) 对收集到的指令进行分类。指令助记符为 `mov`, `xchg`, `stos`, `out`, `rep` 都是潜在的内存访问指令。根据 x86 的指令规范，指令的操作数类型可以用于判断该指令用于内存的读/写。对于控制流相关指令，直接和间接跳转的区别在于 `operand-0` 的类型，前者为一个确定的地址，而后者可能为寄存器或从内存地址中获得。

辅助函数和 BPF maps

现有的 eBPF 生态系统不支持寄存器修改或内存分配。为了实现私有的堆和栈，我们额外增加了 4 个 BPF 辅助函数，这些辅助函数都包含了严格的检查。第一个辅助函数是能够用于修改寄存器的 `bpf_set_regs`，包括栈寄存器`%rsp/%rbp`，O2C 可以通过修改栈寄存器切换内核栈和私有栈。其他的三个辅助函数用于支持私有堆，`bpf_create_slab_cache` 用于创建私有的 slab cache，`bpf_cache_alloc/free` 从私有的 cache 中分配或释放分隔区的私有数据对象。

除了上述 4 个 BPF 辅助函数，我们还额外增加了一些简单的辅助函数来简化 O2C 的 eBPF 程序。例如 `bpf_get_slab` 和 `bpf_get_vmstruct` 可以直接获取保存 slab cache 和 vmalloc 的数据结构，而不需要使用 eBPF 程序遍历保存 slab 数据对象的所有内存页或保存 `vm_struct` 的红黑树。

BPF 使用了不同类型的 BPF map，除了保存决策树模型的 `BPF_MAP_TYPE_ARRAY`，O2C 还大量使用了 `BPF_MAP_TYPE_HASH` 类型的 hash map 来实现分隔化的动态插

装机制。这些 hash map 还被用来保存 slab cache，`vm_struct` 数据结构和 buddy 分配器数据对象。数据对象分析器使用了 `BPF_MAP_TYPE_RINGBUF` 类型的用户和内核之间交互的环形缓冲区，将收集到的内核数据对象的内容发送给用户程序训练决策树模型。

5.9 实验评估

在本节中，我们使用真实世界的漏洞案例全面评估 O2C。我们首先对 O2C 进行安全性分析，然后进行机器学习模型的比较实验。最后，我们测量 O2C 的开销并与相关工作 HAKC[14]进行对比。

5.9.1 安全性分析

我们收集了真实世界中 84 个不同的漏洞，以评估 O2C 对抗攻击的有效性。

缓解状态切换风险

如图 5-2 统计结果所示，虽然“长尾效应”揭示了相当数量的数据对象生命周期可以横跨 t_0 分隔化载入时刻，但无法忽视的是大多数数据对象生命周期非常短，在数据对象最终释放点进行审计的机器学习模型几乎可以覆盖所有分隔化机制无法确定的内存访问。同时，正如我们将在 §5.9.2 中看到的，机器学习模型的准确性非常高。因此，如果攻击者触发漏洞以非法篡改数据对象，它逃避审计的机会很低。在我们的实验中，我们观察到机器学习审计能够成功记录由公开的利用或 PoC 触发的至少一例数据完整性的攻击行为。尽管如此，机器学习模型不可避免地会产生假阴性。作为一项探索性工作，O2C 在审计中的目标是尽可能捕捉到更多违规行为，而不是确保一定没有攻击行为被遗漏。我们希望 O2C 能成为管理状态切换风险更成熟解决方案的倡导者。

控制流完整性

攻击者利用分隔区内的漏洞可以劫持控制流执行代码复用攻击，并最终危及整个系统。为了对抗这一风险，O2C 确保所有间接调用和跳转的目标都是合法的。此外，O2C 检查分隔区中对栈的写操作，以防止越界写入。因此，任何试图篡改栈上返回值的行为都会在它能造成任何不利影响之前被捕获。

数据完整性

除了控制流劫持外，攻击者还可以通过篡改全局数据、栈上的局部变量和堆上的安全敏感对象，发起仅限数据攻击 (data-only attack)[108]。O2C 通过向所有内存写操作装载 eBPF 程序，运行时检查目标地址来阻止这一威胁。具体来说，内核代码受到 W⊗X 的保护。不可信的组件是除核心内核代码外的低质量第三方设备驱动程序，这些驱动程序不包含特权指令。因此，它们被禁止修改 CPU 状态、控制寄存器和页面表项。对于全局数据，分隔区只允许在合法情况下访问允许的内容，遵循最小权限原则。O2C 为分隔区创建了一个私有栈，在其执行期间内核栈禁止访问。最后，分隔区对堆数据对象的访问受到严格限制，只有分隔区私有或与内核合法共有的对象才被授权访问。

参数和返回值

系统管理员作为 O2C 的用户，可以使用静态分析技术来标注参数和返回值的合法值范围，并在 eBPF 程序中表达相应的检查。例如，在利用 `memcpy(dst, size, src)` 的混淆代理攻击中，O2C 确保安全敏感的内核数据对象的地址不会恶意地作为目的地参数传递。当分隔区向内核中的调用者函数返回值时，O2C 确保返回的值与函数的预期行为一致，并且可以被调用者对应地处理。

真实世界漏洞

我们在表 5-2 中详细介绍了 8 个代表性的漏洞，以说明 O2C 如何防止分隔区中的漏洞危害整个系统。在表中，CVE-2021-3715 和 CVE-2022-2588 是内核堆的释放后使用漏洞，CVE-2022-27666 是堆数据对象溢出漏洞。O2C 通过确保分隔区对正确类型的内核对象的访问来缓解这些威胁。CVE-2023-0394 和 syz-76d0b8 访问未授权的内核地址，从而触发 CPU 通用保护故障。O2C 通过验证内存访问的目标地址来反制这一类漏洞。整数溢出漏洞，如 syz-490321，和栈溢出漏洞，如 syz-e73923，有可能导致内核栈溢出并篡改函数返回地址。O2C 为分隔区创建私有栈，并使用栈访问检查来保护返回地址。最后，syz-caed28 示例了非法释放 (invalid free)，其中引用错误类型的指针作为 `kfree` 的参数传递。O2C 通过在 kfree 函数上执行参数授权来防止这种情况。综上，O2C 提供了对各种真实世界漏洞的全面防御。

漏洞 ID	漏洞根源	O2C 的防御措施
CVE-2021-3715	释放后使用漏洞 `sched/cls_route.c`	O2C 限制了分隔区访问其他内核数据对象
CVE-2022-2588	释放后使用漏洞 `sched/cls_route.c`	
CVE-2022-27666	堆溢出漏洞 `ipv6/esp6.c`	
CVE-2023-0394	空指针解引用 `ipv6/raw.c`	O2C 检查了内存访问的目标地址，避免了空指针和非法地址
syz-76d0b8[234]	#GP 错误 `netfilter/nft_tunnel.c`	
syz-e73923[235]	栈溢出漏洞 `ipv6/sit.c`	分隔区使用私有栈，返回地址无法被恶意修改
syz-490321[236]	整数溢出漏洞 `netfilter/nfnetlink.c`	
syz-caed28[237]	非法释放漏洞 `netfilter/nf_tables_api.c`	O2C 检查 `kfree` 函数的返回地址，避免释放内核数据对象

表 5-2 O2C 防止分隔区内的 IPv6, sched, 和 netfilter 内核模块中的漏洞破坏内核

5.9.2 机器学习模型评估

实验设置

我们通过交叉验证评估了机器学习模型的性能。考虑到类型众多的数据对象类型，传统的交叉验证 (cross-validation) 过程会在模型训练中漏掉某些类型。因此，我们采用了 5-fold 分层交叉验证 (5-fold Stratified cross-validation)，以包含训练集中的所有对象类型。关于评价指标，我们不仅考虑准确率，还考虑 Marcro F1 分数，它衡量了模型对于训练样本有限的对象类型的性能。

数据源	介绍	数据量
ipv6	内核运行正常服务时 + ipv6 protocal 服务的数据对象	7,010,295
net/sched	内核运行正常服务时 + 网络包调度 服务质量 (QoS) 的数据对象	6,190,592
netfilter	内核运行正常服务时 + 网络包过滤和 网络地址转换协议 (NAT) 的数据对象	6,489,411

表 5-3 机器学习模型的训练数据概述

训练目标	类型	介绍
数据对象类型	训练数据	由数据分析器收集的内核数据对象，将数据对象的内容分成 8 字节粒度的数组，每个 8 字节元素代表一个特征
	标签	数据对象的类型
是否数据分隔区	训练数据	由数据分析器收集的内核数据对象，将数据对象的内容分成 8 字节粒度的数组，每个 8 字节元素代表一个特征
	标签	数据对象是否属于分隔区

表 5-4 机器学习模型的特征和标签

我们训练了两种粒度的模型实例。一种输出具体的对象类型 (类型粒度)，另一种直接区分访问对象的类型是否属于分隔区 (分隔区粒度)，如表 5-4 所示。

为了说明决策树为什么是 O2C 特定任务最合适的模型，我们将为 eBPF 程序定制的决策树模型，与不受限制的随机森林模型和不受限制的神经网络进行了比较。特别是，神经网络有三个隐藏层[238-239]，每个隐藏层的神经元数量分别为 256、128 和 64。我们使用 Adam 作为优化器，训练了神经网络 50 个周期 (epochs)。最后，我们调整了特征向量的长度和最大深度，以确定决策树的最优参数。模型的训练数据总结见表 5-3。

实验结果

表 5-5 显示，决策树的类型粒度实例实现了 80% - 96% 的准确率，而其分隔区粒度实例在所有数据集上超过了 99% 的准确率。这些结果与不受限制的随机森林模型 (决策树的更高级版本) 的结果相当。此外，两个实例的 Macro F1 分数也与随机森林相当。这表明，即使在为 eBPF 程序定制决策树模型之后，它仍然保持了对数据有限的对象类型的有效性。相比之下，神经网络模型中的 Macro F1 分数要低得多，因为作为多层感知器，它无法处理不平衡的表格数据。考虑到由于当前 eBPF 生态系统的限制，随机森林无法嵌入，决策树模型成为 O2C 更合适的模型。

表 5-6 展示了不同参数值对决策树模型性能的影响。结果显示，特征向量的长度不是决定性的，256 字节已经足够好。关键参数是树的深度，特别是考虑到 Macro F1 分数。较浅的深度会导致模型忽视训练样本有限的对象类型。在 eBPF 生态系统施加的限制条件下，我们确定树深度为 14 时得到最佳结果。此外，比较表 5-5 和表 5-6 中两个实例的结果，我们可以观察到，粗粒度实例 (分隔区粒度) 具有更高的准确率。然而，在生产场景中，这种准确性引入了允许攻击者妥协分隔区内数据的风险。这种风险的出现是因为指针可以引用错误类型，只要错误类型可以在分隔区中使用。O2C 在其实现中提供了两个实例，从而使用户能够根据风险偏好进行权衡。

	类型-粒度		分隔区-粒度	
	Accuracy	**Macro F1**	**Accuracy**	**Macro F1**
IPV6				
DT	96.88 ± 0.65	75.56 ± 1.84	99.99 ±0.02	99.98 ± 0.03
RF	96.91 ± 0.63	78.81 ± 0.73	100 ± 0.01	99.99 ± 0.01
NW	89.63 ± 1.29	38.76± 2.70	99.99 ± 0.01	99.99 ±0.01
net/sched				
DT	80.48 ± 0.76	71.04 ± 1.77	99.93 ± 0.14	97.74 ± 4.22
RF	80.61 ± 0.69	76.28 ± 0.49	100 ± 0	99.99 ± 0.01
NW	65.98 ± 6.91	39.18 ± 1.48	99.66±0.03	89.47±1.20
netfilter				
DT	89.47 ± 0.23	78.17 ± 4.88	99.92 ± 0.07	99.51 ± 0.46
RF	89.54 ± 0.15	81.87 ± 1.86	99.96 ± 0.05	99.77 ± 0.29
NW	72.9 ± 2.23	37.98 ± 2.83	97.16 ±0.17	74 ± 2.56

表 5-5　为 eBPF 程序定制的决策树 (DT) 模型，不受限制的随机森林 (RF)，和不受限制的神经网络 (NW) 模型之间的性能比较

	类型-粒度		分隔区-粒度	
	Accuracy	**Macro F1**	**Accuracy**	**Macro F1**
特征向量的长度 (8 字节)				
64	89.15 ± 0.33	77.24 ± 4.21	99.91 ± 0.07	99.47 ± 0.45
128	89.18 ± 0.29	77.44 ± 4.33	99.85 ± 0.1	99.46 ± 0.64
256	89.26 ± 0.29	77.34 ± 5.06	99.92 ± 0.08	99.51 ± 0.49
1024	89.47 ± 0.23	78.17 ± 4.88	99.92 ± 0.07	99.51 ± 0.46
最大深度				
3	61.18 ± 2.45	1.72 ± 0.19	97.47 ± 0.4	79.34 ± 3.03
7	76.59 ± 2.38	8.48 ± 0.58	99.44 ± 0.21	96.44 ± 1.32
10	83.54 ± 2.19	21.06 ± 2.19	99.65 ± 0.14	97.78 ± 0.86
14	89.47 ± 0.23	78.17 ± 4.88	99.92 ± 0.07	99.51 ± 0.46

表 5-6　不同参数对决策树模型性能的影响

5.9.3　性能开销

通过实验，我们旨在回答以下问题：❶O2C 对整个系统的整体性能开销是什么？❷O2C 在分隔区内本身造成的性能开销是什么？在回答这些问题时，我们改变了分隔区的规模，并在有无机器学习审计的情况下评估了性能。

实验设置

我们对 IPv6 进行了分隔化，IPv6 是在最先进的 HAKC[14]中使用的测试对象。我们还对 net/sched 和 netfilter 进行了分隔，因为最近有几个公开的漏洞攻击针对它们[240-241]。IPv6 执行互联网协议版本 6，net/sched 管理网络数据包调度和服务质量 (QoS)，netfiler 处理数据包过滤和网络地址转换 (NAT)。所有实验都在配置为 CPU Intel core i9-13900HX、64 GB 内存和 1TB 硬盘的机器上进行，运行 Ubuntu-22.10 和内核版本 v6.1(LTS 最新长期支持版本)。此外，我们将内核设置为性能模式并锁定了 CPU 频率，以排除由 Intel Turbo Boost 引起的噪声。为了

最小化波动，我们反复运行基准测试，直到最近五次执行的开销的标准差小于3.5% - Phoronix 中的默认设置。

对于问题❶，我们使用 LMbench[158] 测量系统调用，并使用 Phoronix 测试套件[159] 运行代表性的真实世界应用程序，如 Perf、Apache/Nginx-IPv4、Git、SQLite、Redis、XZ 压缩和内核编译。我们改变了每个分隔区的分隔化规模，从单个核心文件到整个子系统。最后将所有三个子系统都被分隔化，我们总共有七种不同的情况。对于这些情况中的每一种，我们进一步测量了有无机器学习审计的性能 (见表 5-7 中 `w ML / w/o ML` 两列)。

对于问题❷，我们选择了 IPv6 作为目标子系统，以便我们可以与 HAKC ——目前最先进的分隔化技术进行比较。本文与 HAKC 对齐了实验设置，使用 Apache Bench 全面测试 IPv6，以每秒请求次数 (Requests/Sec) 和传输率 (Transfer Rate(KBytes/Sec)) 为指标。我们启动了一个绑定 `[::1]:8000` 地址的本地 IPv6 Web 服务器，并对服务器的 IPv6 子系统进行了分隔化。通过 IPv6 地址，Apache Bench 访问了三种不同大小的文件 100KB、1MB 和 10MB，各 1000 次。请注意，我们将 O2C 与硬件辅助的 HAKC 进行了比较，而不是其他基于 SFI 的分隔化工作，如 XFI[99]、BGI[88]和 LXFI[89]。这是因为 XFI 和 BGI 是为 Windows 操作系统设计的，不适合直接比较。LXFI 没有开源，它对 e1000 英特尔网络驱动进行了分隔化，相应的网卡已经不再市场上销售。

问题❶的结果

表 5-7 展示了以原生内核为基准的 O2C 全系统性能开销的样本结果。从表中我们可以观察到，对于 LMbench，O2C 的整体开销范围从-1.26% 到 3.81%，这可能是由 cache 命中率变化带来的正常波动[161]。对于 Phoronix，开销稍高，尤其是内核编译和 Apache。这是因为这些应用程序或频繁创建子进程，或持续处理传入的网络数据包。这些操作导致了大量的内存分配，以上所有操作都需要被 O2C 跟踪，从而导致性能损失。

关于机器学习审计的影响，表中所有的基准测试的结果相比未使用机器学习审计都没有明显增加。此外，随着分隔区规模的增加，从单个文件 `ip6_output.c` 开始，代码行数为 1,470 行 (LOC)，到 IPv6 子系统，代码行数为 78,213 行，最后是三个子系统共 255,330 行，整个系统的性能没有明显下降，证明了 O2C 在

LMBench	ip6_output.c (1,470 loc)		IPv6 (78,213 loc)		cls_route.c (681 loc)		sched (49,901 loc)		nf_tables_api.c (10,615 loc)		netfilter (97,570 loc)		IPv6+ sched+netfiler (255,330 LOC)	
	w ML	w/o ML	w ML	w/o ML	w ML	w/o ML	w ML	w/o ML	w ML	w/o ML	w ML	w/o ML	w ML	w/o ML
Simple syscall	-0.17%	-0.35%	-0.28%	-0.17%	0.05%	-0.17%	0.23%	0.23%	-0.14%	-0.20%	-0.18%	-0.14%	-0.04%	-0.09%
Simple read	-0.15%	-0.01%	0.38%	-0.15%	0.12%	0.07%	0.12%	0.06%	-0.45%	0.53%	0.24%	0.01%	0.02%	0.66%
Simple write	-0.23%	-0.12%	-0.07%	-0.23%	0.04%	0.08%	0.11%	0.11%	-0.27%	1.10%	1.06%	0.07%	0.05%	-0.03%
Simple stat	0.44%	0.29%	1.63%	0.44%	1.27%	0.91%	0.91%	1.67%	-0.25%	1.70%	1.01%	8.52%	0.47%	-0.91%
Simple fstat	-0.14%	0.48%	0.10%	-0.14%	0.50%	0.13%	1.00%	0.99%	-0.85%	1.42%	0.73%	8.61%	0.02%	0.65%
Simple open/close	1.79%	1.08%	1.58%	1.79%	2.23%	2.12%	2.59%	1.95%	1.00%	3.27%	2.49%	6.82%	3.45%	2.40%
Select on 10 fd's	-1.57%	-1.00%	-0.43%	-1.57%	-0.52%	-0.73%	-1.60%	-0.98%	-5.66%	-1.05%	-5.11%	1.73%	-0.56%	-1.07%
Select on 100 fd's	-0.20%	-0.28%	0.04%	-0.20%	0.66%	0.08%	-0.01%	0.51%	-0.51%	0.69%	0.38%	7.35%	0.14%	-0.50%
Select on 250 fd's	0.47%	0.17%	0.61%	0.47%	0.26%	0.07%	0.28%	0.67%	0.12%	3.21%	2.87%	5.09%	0.08%	-0.04%
Select on 500 fd's	3.21%	3.10%	3.59%	3.21%	3.70%	2.63%	3.18%	3.13%	0.22%	3.91%	0.94%	5.06%	3.92%	3.41%
Select on 10 tcp fd's	-0.29%	0.34%	-0.12%	-0.29%	-0.06%	-0.28%	0.23%	0.24%	-0.89%	0.32%	-0.27%	3.06%	0.12%	-0.02%
Select on 100 tcp fd's	-0.05%	-0.29%	0.32%	-0.05%	0.57%	-0.19%	-0.43%	-0.26%	-0.28%	3.14%	2.91%	6.53%	-0.14%	-0.32%
Select on 250 tcp fd's	-0.81%	-0.97%	-0.05%	-0.81%	0.09%	-0.84%	-0.74%	-0.37%	-1.01%	-0.40%	-0.60%	-0.50%	-0.33%	-0.95%
Select on 500 tcp fd's	1.45%	1.45%	1.66%	1.45%	1.18%	1.07%	1.80%	2.20%	1.05%	5.07%	4.69%	1.61%	2.21%	1.33%
Signal handler install	0.13%	1.51%	1.03%	0.13%	0.47%	1.26%	3.06%	1.09%	0.17%	-0.55%	-0.51%	0.76%	1.50%	0.24%
Signal handler overhead	0.06%	-0.06%	-0.27%	0.06%	-0.25%	0.15%	-0.26%	0.01%	0.11%	-0.15%	-0.10%	7.11%	-0.06%	0.93%
Protection fault	1.17%	-0.05%	-0.86%	1.17%	-2.09%	-1.23%	1.29%	0.19%	1.17%	-0.19%	-0.19%	11.41%	1.19%	0.45%
Pipe latency	-0.64%	-1.26%	3.32%	-0.64%	-0.17%	-0.09%	-1.89%	0.05%	-0.60%	-0.23%	-0.19%	1.47%	-0.41%	-0.92%
Pagefaults	1.95%	2.14%	2.03%	1.95%	2.73%	2.97%	1.93%	2.31%	1.63%	4.19%	3.89%	2.95%	3.18%	3.53%
UDP latency	-0.66%	0.65%	3.81%	-0.66%	0.95%	-0.50%	2.22%	1.53%	-0.41%	5.01%	5.25%	3.57%	3.65%	1.45%
TCP latency	0.37%	1.38%	0.44%	0.37%	0.22%	-0.01%	2.57%	1.11%	0.30%	2.04%	1.97%	3.12%	2.14%	1.31%
average	0.29%	0.39%	0.88%	0.29%	0.57%	0.36%	0.79%	0.78%	-0.26%	1.56%	1.01%	4.01%	0.98%	0.55%
Phoronix	**w ML**	**w/o ML**	**w ML**	**w/o ML**	**w ML**	**w/o ML**	**w ML**	**w/o ML**	**w ML**	**w/o ML**	**w ML**	**w/o ML**	**w ML**	**w/o ML**
perf-bench	0.13%	-0.35%	0.53%	-0.55%	0.06%	0.28%	0.33%	0.73%	-0.13%	-0.21%	-0.17%	0.38%	0.04%	-0.39%
OSBench	1.87%	5.97%	3.94%	5.85%	3.20%	4.05%	9.86%	6.38%	11.43%	0.90%	-0.07%	0.07%	7.23%	7.32%
Kernel Compilation	3.06%	3.30%	3.25%	2.70%	3.81%	3.39%	2.99%	4.65%	3.48%	3.29%	2.73%	2.80%	5.48%	6.31%
XZ Compression	-1.65%	-1.58%	-0.83%	-1.75%	-0.59%	-0.55%	-1.68%	-0.73%	-0.16%	0.17%	-1.35%	-1.63%	-0.41%	-0.38%
OpenSSL	2.04%	2.06%	1.30%	1.37%	0.57%	0.30%	1.82%	2.30%	0.90%	0.98%	0.91%	0.01%	-0.14%	1.83%
SQLite Speedtest	3.73%	1.32%	-2.90%	5.32%	-0.10%	-0.41%	-0.13%	0.20%	-0.65%	-0.36%	2.15%	1.45%	-0.21%	-0.04%
Nginx-ipv4	1.81%	2.39%	2.20%	2.66%	1.61%	1.75%	1.67%	1.59%	2.12%	2.12%	20.41%	17.62%	3.02%	2.40%
Apache-ipv4	3.22%	3.76%	3.93%	3.69%	4.77%	4.16%	4.43%	4.07%	3.44%	3.13%	11.08%	10.10%	7.23%	5.89%
Git	2.49%	1.99%	2.67%	1.80%	1.29%	1.37%	1.92%	2.49%	1.74%	2.19%	1.37%	1.45%	1.78%	1.07%
Average	1.21%	1.71%	0.96%	1.36%	1.63%	1.19%	1.46%	1.51%	1.92%	0.71%	4.06%	3.15%	1.96%	2.16%

表 5-7　O2C 在不同分隔区配置下的系统性能开销，以原生内核测试结果为基线

系统层次的出色可扩展性。

问题❷的结果

图 5-6 展示了 O2C 在分隔区内造成的性能开销。所有实验结果都与原生内核进行了标准化。从图中我们可以观察到，两个指标每秒请求数和传输率的结果显示出类似的趋势。更具体地说，当分隔区的规模为单个文件时，对于所有文件大小 (100KB、1MB 和 10MB)，即使进行机器学习审计，性能开销也是可以忽略不计的。然而，如果规模是整个 IPv6 子系统，100KB 文件的性能下降了 5%，这仍然是可以接受的，但对于 1MB 和 10MB 文件，性能急剧下降了近 20-35%。

在这个规模上，机器学习审计也对性能产生了影响。这种变化可以归因于文件大小。文件越大，eBPF 程序执行越频繁，机器学习审计中未跟踪的对象就越多。我们将 O2C 与 HAKC 进行了比较。由于 HAKC 使用的硬件特性，即 ARM MTE，尚未公开可用，我们只能使用论文中报告的实验结果进行比较。从图中我们可以看到，处理 10MB 文件时，HAKC 的性能优于 O2C。这是因为 HAKC 利用硬件特性在许多内存访问位于同一区域时降低检查开销。然而，随着文件

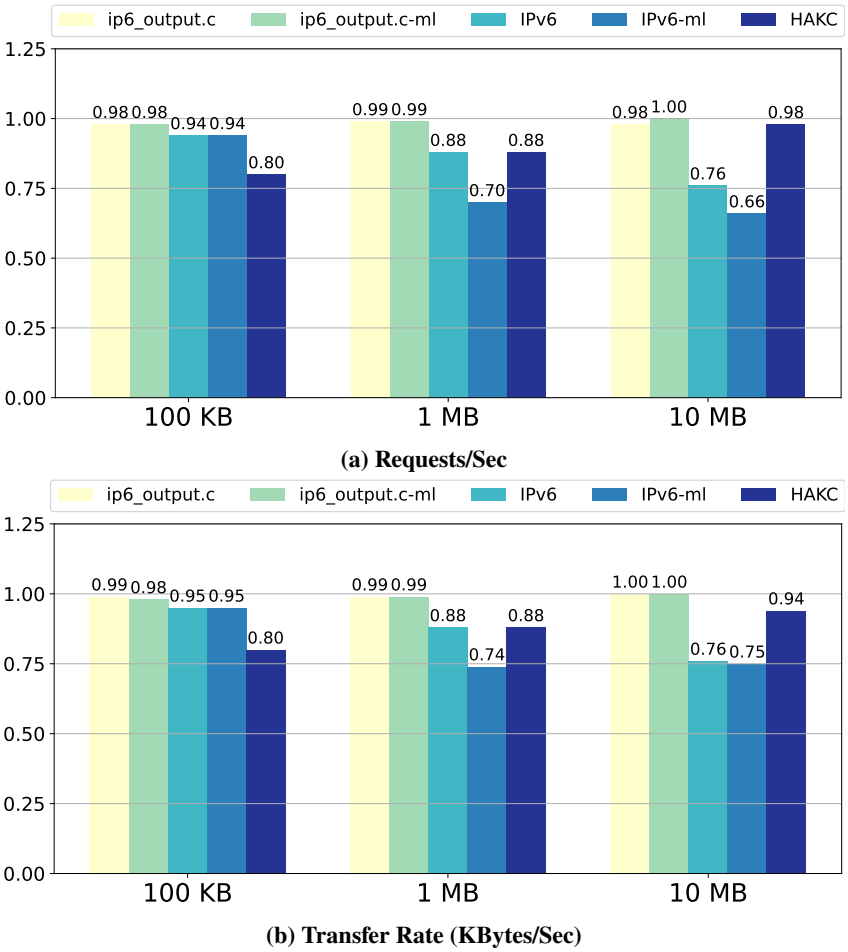


图 5-6 O2C 对比 HAKC，Apache Bench 访问三种不同大小的文件 100KB、1MB 和 10MB - 共 1000 次的性能开销

大小的减小，这种硬件特性的优化效果减弱。这就是为什么 O2C 在处理 100KB 文件时显示出优势。在性能比较之外，HAKC 缺乏对实时分隔化的支持，而这是 O2C 的主要贡献。

5.10 讨论

审计的假阳性。尽管 O2C 中的决策树模型准确率已经大于 99% 了，但是实际工作中也不可避免地存在假阳性情况。作为一个探索性的工作，O2C 的目标是捕获尽可能多的攻击行为，而不是保证分隔区一定不会出现任何违反安全目标的行为。在未来的工作中我们将会探索可行性更强的方案，例如在数据对象分配时直接用元数据记录对应的类型。对于该方案可能带来的内存和性能开销的增加，我们也将探索如何将其与 eBPF 更高效地结合。

读内存指令。安全模型假设了不可信的分隔区有权限取读取可信的内核数据，这

是由于之前的相关工作通常认为读内存的安全威胁通常较低，而且仅有读内存漏洞也无法完成内核提权攻击，与此同时内存读指令的数量是写指令数量的约 3 倍，对其进行插桩可能会造成过大运行时开销。我们计划在 O2C 的改进计划中加入对读权限的限制，我们可以结合内存权限划分的硬件机制，例如 PKS(MPK)，在限制读内存漏洞的同时保证较低开销。

性能优化。 O2C 的运行时性能还有进一步提升空间，我们可以借鉴相关工作中 ASan–[226], Carat[162] 和 CaratCake[242]中提到的跳出循环、去掉重复检查等技术进行优化。在此基础上，我们还将结合操作系统内核的特性对以上检查进行进一步优化。

与内核共有的数据对象。分隔区与内核之间的数据共享是复杂的。共有对象可以像链表那样组织在数据结构中，或者像以前的工作如 LVD[94]和 KSplit[95] 那样，进行层次管理。虽然这些工作提供了维护和同步共有对象多个副本的机制，但确定同步时的数据副本是否是恶意的，仍然是一个开放的挑战。O2C 遵循最小权限原则。因此，为了保持分隔区的正常功能，保留了对某些资源 (包括共享对象) 的访问。尽管存在恶意分隔区通过共享对象利用内核的潜在风险，但通过对内存访问和对参数和返回值的检查，O2C 大大降低了这一威胁。

丰富训练数据。为了更好地训练机器学习模型，O2C 可以使用谷歌公司开发的内核模糊测试工具 syzkaller[1]来执行内核，获取更多的训练数据，并配合 slake[134]中的 syzkaller 模板集中执行分隔区中的代码，尽可能全面地获取分隔区内数据对象的类型。O2C 还可以使用 GREBE[188]这类数据对象驱动的模糊测试技术，让内核分配更多指定类型的数据对象。此外，我们还可以通过在不同的机器上重复上述模糊测试，丰富数据对象内容的多样性以获得更高质量的训练数据。

5.11 小结

在这项工作中，我们调查并揭示了状态切换风险是实现实时分隔化的主要挑战。我们设计并实现了 O2C 作为一种探索性解决方案来审计状态切换风险。O2C 展示了将机器学习模型嵌入到内核中，以实现更智能、更安全操作系统的潜力。同时，O2C 可以利用 eBPF 生态系统进行基于动态插桩的分隔化强制执行。我们的评估显示 O2C 是有效且轻量级的，并具备了卓越的可扩展性。

第六章　总结与展望

6.1　总结

内核漏洞的实时防御将内核安全的关注点聚焦到了内核漏洞从公开到完成修复的关键时间窗口上。在此期间，内核漏洞利用前阶段的热补丁、漏洞预防，漏洞利用阶段的内核缓解机制、内核裁剪、系统调用过滤，以及漏洞利用后阶段的分隔化和完整性保护等相关工作，均无法对尚无修复补丁的漏洞进行有效防御。为应对这一挑战，本研究由浅入深的针对漏洞利用的中、前、后三个阶段，设计了创新的安全防御策略，并通过对 eBPF 生态系统的升级与改造，实现了这些防御策略的实时部署。本文的核心工作总结和贡献概述如下：

1. **ERA: 基于 eBPF 的内核堆漏洞缓解框架**。本文提出了 ERA，一种基于 eBPF 的内核堆漏洞动态缓解框架，在内核漏洞利用阶段真正有效地提升了攻击者利用堆漏洞的难度。ERA 率先将 eBPF 技术应用于漏洞防御领域，配合对 eBPF 生态系统的改造，实现了基于随机化的堆安全分配器，在分配时将漏洞数据对象动态替换为随机化保护的数据对象，有效破坏了攻击者依赖的数据对象重叠条件。实验结果显示 ERA 能有效防御常见类型内核堆漏洞，压力测试情况下仅对系统造成 1% 的性能开销和微不足道内存开销。
2. **PET: 基于 eBPF 的防内核漏洞触发框架**。本文提出了 PET，一种基于 eBPF 的防内核漏洞触发框架，用于在漏洞利用前阶段阻止内核漏洞的触发而无需任何修复补丁。PET 能够为最先进内核漏洞消毒器报告的整数溢出、越界访问、释放后使用、未初始化访问以及数据竞争多种类型的内核漏洞构造相应的 eBPF 防御程序，且构造过程可以在 5 分钟内完成，有效防御了内核漏洞修复窗口的恶意攻击。经过实验分析，PET 展现了对常见的漏洞类型的防御能力，在同时防御多个内核漏洞时也仅对系统造成 3% 以内的轻量级开销，而且在漏洞触发后能继续保持内核稳定运行。此外，PET 通过改造 eBPF 生态系统支持漏洞类型无关运行时机制，有潜力在未来实现更多类型内核漏洞的实

时防御

3. **O2C: 基于 eBPF 和机器学习审计的实时内核分隔探索**。本文提出了 O2C，是相关领域第一个尝试解决实时内核分隔化的探索性工作，用于在漏洞利用后阶段降低安全风险。O2C 首先揭示了实时分隔化的主要挑战在于解决状态切换风险，并在此基础上提出了基于决策树机器学习模型的审计方案。同时 O2C 使用 eBPF 实现了动态插装和状态切换风险审计，构造了完整的分隔化原型系统。实验评估显示 O2C 能有效隔离不可信内核组件的安全风险，在同时隔离超过 255,000 行代码时也仅对系统造成了小于 4% 的性能开销，具备出色的可扩展性。

通过这些创新性的防御方案，本文不仅在策略层面实现了高级别的内核漏洞安全防御，及时响应了主流操作系统内核的漏洞防御需求。同时在机制层面通过 eBPF 生态系统的改造和提升，有效实施了防御策略的实时部署，展现了 eBPF 在安全领域应用的全新思路和实践。

6.2 未来工作展望

增强 PET 稳定性。 PET 提供了一个具备强大可扩展性的防内核漏洞触发框架，它能在漏洞触发前跳过漏洞触发指令并将内核恢复到平稳运行状态。但恢复操作不得不面对状态泄露 (state spill) 问题，这使得 PET 的恢复在机制层面上存在局限性，可能在未来某时导致系统崩溃。为增强 PET 的稳定性，我们可以参考编程语言领域的最弱前置条件 (weakest precondition)，利用静态分析、符号执行等技术手段，将判断漏洞能否触发的位置从漏洞触发点沿着代码执行路径前移。前移的目标位置可以是系统调用，在攻击程序进入内核前就判断出漏洞触发风险并进行拦截，也可以是内核中大量存在的 LSM Hook(Linux Security Module)，这些良好定义的 Hook 接口也能保证内核稳定结束当前任务。

eBPF 程序保护。 eBPF 是近些年 Linux 内核社区的革命性技术，它支持将用户定义的程序运行在内核级沙箱，实现内核观测、安全、包过滤等需求。但有研究表明，eBPF 程序也可以作为攻击目标，其正在运行的观测、安全等任务极有可能被攻击者破坏。对此，我们可以结合 hypervisor/硬件机制构造可信基，将 eBPF 程序运行的沙箱 (sandbox) 强化成安全箱 (safebox)，赋予 eBPF 程序更强的安全

保证，使其能够在更严苛的威胁环境下，例如内核不可信，可靠地实现对系统的监控。

开源操作系统社区贡献。本文的主要工作均围绕开源操作系统内核 Linux 开展，因此贡献的解决方案可以直接应用到内核中。我们将继续努力完善工程细节，将文中对 eBPF 的改进和增强，以及实用性较强 ERA 和 PET 两项工作向社区提交，帮助操作系统内核及时应对未被修复的漏洞，增强内核的安全性和稳定性。

参考文献

[1] syzbot — syzkaller.appspot.com[Z]. https://syzkaller.appspot.com/.

[2] Linux Linux Kernel : Security vulnerabilities, CVEs — cvedetails.com[Z]. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html.

[3] TALEBI S M S, YAO Z, SANI A A, et al. Undo Workarounds for Kernel Bugs [C]//30th USENIX Security Symposium (SEC'21). 2021.

[4] ALEXOPOULOS N, BRACK M, WAGNER J P, et al. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes[C]//31st USENIX Security Symposium (SEC'22). 2022.

[5] RICE L. Cloud Native SecurityCon Europe 2022: Real Time Security - eBPF for Preventing Attacks[Z]. https://cloudnativesecurityconeu22.sched.com/event/zsUt/real-time-security-ebpf-for-preventing-attacks-liz-rice-isovalent. 2022.

[6] GREGG B. BPF Internals[C]//Usenix Large Installation System Administration Conference (LISA'21). 2021.

[7] "Introducing kpatch: Dynamic Kernel Patching." [Z]. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching. 2014.

[8] JONES R W, KELLY P H. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs.[C]//AADEBUG: vol. 97. 1997: 13-26.

[9] AKRITIDIS P. Cling: A Memory Allocator to Mitigate Dangling Pointers[C]//19th USENIX Security Symposium (SEC'10). 2010.

[10] SONG D, LETTNER J, RAJASEKARAN P, et al. SoK: Sanitizing for Security [C]//2019 IEEE Symposium on Security and Privacy (SP'19). 2019.

[11] Kernel Self Protection Project - Linux Kernel Security Subsystem — kernsec.org[Z]. https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project. 2022.

[12] GU Z, SALTAFORMAGGIO B, ZHANG X, et al. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine[C]//44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14). 2014.

[13] LI Y, DOLAN-GAVITT B, WEBER S, et al. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path[C]//2017 USENIX Annual Technical Conference (ATC'17). 2017.

[14] MCKEE D, GIANNARIS Y, PEREZ C O, et al. Preventing kernel hacks with hakc[C]//29th Network and Distributed System Security Symposium (NDSS'22). 2022.

[15] SZEKERES L, PAYER M, WEI T, et al. Sok: Eternal war in memory[C]//2013 IEEE Symposium on Security and Privacy (SP'13). 2013.

[16] Applying Patches To The Linux Kernel The Linux Kernel documentation — kernel.org[Z]. https://www.kernel.org/doc/html/v4.10/process/applying-patches.html. 2016.

[17] TIAN Y, LAWALL J, LO D. Identifying Linux Bug Fixing Patches[C]//34th International Conference on Software Engineering (ICSE'12). 2012.

[18] ZHANG Z, ZHANG H, QIAN Z, et al. An Investigation of the Android Kernel Patch Ecosystem[C]//30th USENIX Security Symposium (SEC'21). 2021.

[19] Live Kernel Patching Using kGraft | SUSE Linux[Z]. https://documentation.suse.com/sles/12-SP5/html/SLES-kgraft/index.html.

[20] ARNOLD J, KAASHOEK M F. Ksplice: Automatic Rebootless Kernel Updates [C]//4th European Conference on Computer Systems (EuroSys'09). 2009.

[21] Ubuntu Livepatch Service | Security | Ubuntu. [Online][Z]. https://ubuntu.com/security/livepatch.

[22] CHEN H, CHEN R, ZHANG F, et al. Live updating operating systems using virtualization[C]//2nd International Conference on Virtual Execution Environments (VEE'06). 2006.

[23] CHEN Y, ZHANG Y, WANG Z, et al. Adaptive Android Kernel Live Patching [C]//26th USENIX Security Symposium (SEC'17). 2017.

[24] XU Z, ZHANG Y, ZHENG L, et al. Automatic Hot Patch Generation for Android Kernels[C]//29th USENIX Security Symposium (SEC'20). 2020.

[25] SINIAVINE M, GOEL A. Seamless Kernel Updates[C]//43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13). 2013.

[26] ZHOU L, ZHANG F, LIAO J, et al. KShot: Live Kernel Patching with SMM and SGX[C]//50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'20). 2020.

[27] NAGARAKATTE S, ZHAO J, MARTIN M M K, et al. SoftBound: highly compatible and complete spatial memory safety for c[C]//2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09). 2009.

[28] AINSWORTH S, JONES T M. MarkUs: Drop-in Use-after-free Prevention for Low-level Languages[C]//2020 IEEE Symposium on Security and Privacy (SP'20). 2020.

[29] ERDOS M, AINSWORTH S, JONES T M. MineSweeper: a "clean sweep" for drop-in use-after-free prevention[C]//27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22). 2022.

[30] Van der KOUWE E, NIGADE V, GIUFFRIDA C. DangSan: Scalable Use-after-free Detection[C]//12th European Conference on Computer Systems (EuroSys'17). 2017.

[31] LEE B, SONG C, JANG Y, et al. Preventing Use-after-free with Dangling Point-

ers Nullification[C]//22nd Annual Network and Distributed System Security Symposium (NDSS'15). 2015.

[32] SHIN J, KWON D, SEO J, et al. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++[C]//26th Annual Network and Distributed System Security Symposium (NDSS'19). 2019.

[33] DANG T H, MANIATIS P, WAGNER D. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers[C]//26th USENIX Security Symposium (SEC'17). 2017.

[34] WICKMAN B, HU H, YUN I, et al. Preventing Use-After-Free Attacks with Fast Forward Allocation[C]//30th USENIX Security Symposium (SEC'21). 2021.

[35] TIAN D, ZENG Q, WU D, et al. Kruiser: Semi-synchronized Non-blocking Concurrent Kernel Heap Buffer Overflow Monitoring.[C]//19th Annual Network and Distributed System Security Symposium (NDSS'12). 2012.

[36] CHO H, PARK J, OEST A, et al. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection[C]//27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22). 2022.

[37] WANG D, ZHANG Z, ZHANG H, et al. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning[C]//30th USENIX Security Symposium (SEC'21). 2021.

[38] ZHAO B, LI Z, QIN S, et al. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing[C]//31st USENIX Security Symposium (SEC'22). 2022.

[39] CHEN W, WANG Y, ZHANG Z, et al. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers[C]//2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21). 2021.

[40] The Kernel Address Sanitizer (KASAN) The Linux Kernel documentation —

kernel.org[Z]. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[41] The Kernel Memory Sanitizer (KMSAN) The Linux Kernel documentation — docs.kernel.org[Z]. https://docs.kernel.org/next/dev-tools/kmsan.html.

[42] The Kernel Concurrency Sanitizer (KCSAN) The Linux Kernel documentation — kernel.org[Z]. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[43] UBSan: run-time undefined behavior sanity checker [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/617364/.

[44] x86-64: Stack protector and percpu improvements [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/876081/.

[45] ACCARDI K C. Function Granular KASLR [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/824307/. 2020.

[46] Per-system-call kernel-stack offset randomization [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/816085/.

[47] BERGER E D, ZORN B G. DieHard: probabilistic memory safety for unsafe languages[C]//ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06). 2006.

[48] NOVARK G, BERGER E D. DieHarder: Securing the Heap[C]//5th USENIX Workshop on Offensive Technologies (WOOT'11). 2011.

[49] SILVESTRO S, LIU H, CROSSER C, et al. FreeGuard: A Faster Secure Heap Allocator[C]//2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17). 2017.

[50] SILVESTRO S, LIU H, LIU T, et al. Guarder: A Tunable Secure Allocator[C]//27th USENIX Security Symposium (SEC'18). 2018.

[51] LIU B, OLIVIER P, RAVINDRAN B. SlimGuard: A Secure and Memory-Efficient Heap Allocator[C]//20th International Middleware Conference (Middleware'19). 2019.

[52] Lwn.net. mm: SLUB Freelist randomization [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/688749/. 2016.

[53] COOK K. slub: Improve bit diffusion for freelist ptr obfuscation - Patchwork — keescook[Z]. https://patchwork.kernel.org/project/linux-mm/patch/202003051623.AF4F8CB@keescook/. 2020.

[54] [PATCH v2 2/2] slab: Add naive detection of double free - Kees Cook[Z]. https://lore.kernel.org/lkml/20200625215548.389774-3-keescook@chromium.org/.

[55] Kernel Electric-Fence (KFENCE) The Linux Kernel documentation[Z]. https://docs.kernel.org/dev-tools/kfence.html. 2014.

[56] HUSSEIN N. Randomizing structure layout [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/722293/. 2017.

[57] EDGE J. Hardened Usercopy [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/695991/. 2016.

[58] HUA Z, DU D, XIA Y, et al. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs[C]//2018 USENIX Annual Technical Conference (ATC'18). 2018.

[59] LIPP M, SCHWARZ M, GRUSS D, et al. Meltdown: Reading Kernel Memory from User Space[C]//27th USENIX Security Symposium (SEC'18). 2018.

[60] WU W, CHEN Y, XU J, et al. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities[C]//27th USENIX Security Symposium (SEC'18). 2018.

[61] KEMERLIS V P, PORTOKALIDIS G, KEROMYTIS A D. kGuard: Lightweight Kernel Protection against Return-to-User Attacks[C]//21th USENIX Security Symposium (SEC'12). 2012.

[62] XIE M, WU C, ZHANG Y, et al. CETIS: Retrofitting Intel CET for generic and efficient intra-process memory isolation[C]//2022 ACM SIGSAC Conference on Computer and Communications Security (CCS'22). 2022.

[63] ZHANG Z, CHENG Y, NEPAL S, et al. KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels[C]//21st International Symposium on Research in Attacks, Intrusions and Defenses

(RAID'18). 2018.

[64] ABUBAKAR M, AHMAD A, FONSECA P, et al. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening[C]//30th USENIX Security Symposium (SEC'21). 2021.

[65] HU Z, LEE S, PEINADO M. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis[C]//2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23). 2023.

[66] GHAVAMNIA S, PALIT T, BENAMEUR A, et al. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction[C]//23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'20). 2020.

[67] GHAVAMNIA S, PALIT T, MISHRA S, et al. Temporal System Call Specialization for Attack Surface Reduction[C]//29th USENIX Security Symposium (SEC'20). 2020.

[68] GHAVAMNIA S, PALIT T, POLYCHRONAKIS M. C2C: Fine-grained Configuration-driven System Call Filtering[C]//2022 ACM SIGSAC Conference on Computer and Communications Security (CCS'22). 2022.

[69] XING Y, WANG X, TORABI S, et al. A Hybrid System Call Profiling Approach for Container Protection[J]. IEEE Transactions on Dependable and Secure Computing, 2023.

[70] BULEKOV A, JAHANSHAHI R, EGELE M. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists[C]//30th USENIX Security Symposium (SEC'21). 2021.

[71] GAIDIS A J, ATLIDAKIS V, KEMERLIS V P. SysXCHG: Refining Privilege with Adaptive System Call Filters[C]//2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23). 2023.

[72] HUNG H W, LIU Y, SANI A A. Sifter: protecting security-critical kernel modules in Android through attack surface reduction[C]//28th Annual Inter-

national Conference on Mobile Computing And Networking (MobiCom'22). 2022.

[73] ANDERSON. J P. Computer security technology planning study. Volumes I and II.[R]. ESD-TR-73-51. The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, 1972.

[74] SHARIF M I, LEE W, CUI W, et al. Secure in-VM monitoring using hardware virtualization[C]//2009 ACM Conference on Computer and Communications Security (CCS'09). 2009.

[75] SHI B, CUI L, LI B, et al. ShadowMonitor: An Effective In-VM Monitoring Framework with Hardware-Enforced Isolation[C]//21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID'18). 2018.

[76] KWON D, OH K, PARK J, et al. Hypernel: a hardware-assisted framework for kernel protection without nested paging[C]//55th Annual Design Automation Conference (DAC'18). 2018.

[77] XIONG X, LIU P. SILVER: Fine-grained and transparent protection domain primitives in commodity OS kernel[C]//10th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'13). 2013.

[78] SRIVASTAVA A, GIFFIN J T. Efficient Monitoring of Untrusted Kernel-Mode Execution[C]//18th Annual Network and Distributed System Security Symposium (NDSS'11). 2011.

[79] HEDAYATI M, GRAVANI S, JOHNSON E, et al. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries[C]//2019 USENIX Annual Technical Conference (ATC'19). 2019.

[80] LEE D, KOHLBRENNER D, SHINDE S, et al. Keystone: an open framework for architecting trusted execution environments[C]//15th European Conference on Computer Systems (EuroSys'20). 2020.

[81] BOYD-WICKIZER S, ZELDOVICH N. Tolerating malicious device drivers in linux[C]//2010 USENIX Annual Technical Conference (ATC'10). 2010.

[82] SUN Y, CHIUEH T C. SIDE: Isolated and efficient execution of unmodified device drivers[C]//43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13). 2013.

[83] SWIFT M M, BERSHAD B N, LEVY H M. Improving the Reliability of Commodity Operating Systems[C]//19th ACM Symposium on Operating Systems Principles (SOSP'03). 2003.

[84] AZAB A M, SWIDOWSKI K, BHUTKAR R, et al. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM[C]//23th Network and Distributed System Security (NDSS'16). 2016.

[85] GRAVANI S, HEDAYATI M, CRISWELL J, et al. Fast Intra-kernel Isolation and Security with Iskios[C]//24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21). 2021.

[86] DAUTENHAHN N, KASAMPALIS T, DIETZ W, et al. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation[C]//20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15). 2015.

[87] WAHBE R, LUCCO S, ANDERSON T E, et al. Efficient software-based fault isolation[C]//14th ACM symposium on Operating systems principles (SOSP'93). 1993.

[88] CASTRO M, COSTA M, MARTIN J P, et al. Fast byte-granularity software fault isolation[C]//22nd ACM SIGOPS symposium on Operating systems principles (SOSP'09). 2009.

[89] MAO Y, CHEN H, ZHOU D, et al. Software Fault Isolation with API Integrity and Multi-Principal Modules[C]//23th symposium on Operating systems design and implementation (SOSP'11). 2011.

[90] BHATTACHARYYA A, HOFHAMMER F, LI Y, et al. SecureCells: A Secure Compartmentalized Architecture[C]//2023 IEEE Symposium on Security and Privacy (SP'23). 2023.

[91] ROESSLER N, DEHON A. SCALPEL: Exploring the Limits of Tag-Enforced Compartmentalization[J]. J. Emerg. Technol. Comput. Syst., 2021.

[92] KOROMILAS L, VASILIADIS G, ATHANASOPOULOS E, et al. GRIM: Leveraging GPUs for Kernel Integrity Monitoring[C]//19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'16). 2016.

[93] NARAYANAN V, BALASUBRAMANIAN A, JACOBSEN C, et al. LXDs: Towards Isolation of Kernel Subsystems[C]//2019 USENIX Annual Technical Conference (ATC'19). 2019.

[94] NARAYANAN V, HUANG Y, TAN G, et al. Lightweight Kernel Isolation with Virtualization and VM Functions[C]//16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20). 2020.

[95] HUANG Y, NARAYANAN V, DETWEILER D, et al. KSplit: Automating Device Driver Isolation[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22). 2022.

[96] GUO Y, WANG Z, ZHONG B, et al. Formal Modeling and Security Analysis for Intra-level Privilege Separation[C]//38th Annual Computer Security Applications Conference (ACSAC'22). 2022.

[97] 钟炳南, 邓良, 曾庆凯. 基于硬件虚拟化的内核同层多域隔离模型[J]. 软件学报, 2022.

[98] XIONG X, TIAN D, LIU P, et al. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions[C]//18th Network and Distributed System Security (NDSS'11). 2011.

[99] ERLINGSSON U, ABADI M, VRABLE M, et al. XFI: Software guards for system address spaces[C]//7th symposium on Operating systems design and implementation (SOSP'06). 2006.

[100] ROESSLER N, ATAYDE L, PALMER I, et al. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts [C]//24th International Symposium on Research in Attacks, Intrusions and

Defenses (RAID'21). 2021.

[101] LEFEUVRE H, BĂDOIU V A, CHIEN Y, et al. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software[C]//30th Network and Distributed System Security (NDSS'23). 2022.

[102] SESHADRI A, LUK M, QU N, et al. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes[C]//21st ACM Symposium on Operating Systems Principles (SOSP'07). 2007.

[103] CRISWELL J, DAUTENHAHN N, ADVE V S. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels[C]//2014 IEEE Symposium on Security and Privacy (SP'14). 2014.

[104] CRISWELL J, DAUTENHAHN N, ADVE V S. Virtual Ghost: Protecting Applications from Hostile Operating Systems[C]//19th Architectural Support for Programming Languages and Operating Systems (ASPLOS'14). 2014.

[105] YOO S, PARK J, KIM S, et al. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication[C]//31st USENIX Security Symposium (SEC'22). 2022.

[106] LI J, TONG X, ZHANG F, et al. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels[J]. IEEE Transactions on Information Forensics and Security, 2018.

[107] LU K, HU H. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis[C]//2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19). 2019.

[108] DAVI L, GENS D, LIEBCHEN C, et al. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables[C]//24th Network and Distributed System Security (NDSS'17). 2017.

[109] JANG D, LEE H, KIM M, et al. ATRA: Address Translation Redirection Attack against Hardware-based External Monitors[C]//2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14). 2014.

[110] PROSKURIN S, MOMEU M, GHAVAMNIA S, et al. xMP: Selective Memory Protection for Kernel and User Space[C]//2020 IEEE Symposium on Security and Privacy (SP'20). 2020.

[111] SONG C, LEE B, LU K, et al. Enforcing Kernel Security Invariants with Data Flow Integrity[C]//23rd Annual Network and Distributed System Security Symposium (NDSS'16). 2016.

[112] CHEN X, GARFINKEL T, LEWIS E C, et al. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems[C]//13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08). 2008.

[113] HOFMANN O S, KIM S, DUNN A M, et al. InkTag: secure applications on an untrusted operating system[C]//Architectural Support for Programming Languages and Operating Systems (ASPLOS'13). 2013.

[114] KAFFES K, HUMPHRIES J T, MAZIÈRES D, et al. Syrup: User-Defined Scheduling Across the Stack[C]//ACM SIGOPS 28th Symposium on Operating Systems Principles,(SOSP'21). 2021.

[115] ZHONG Y, LI H, WU Y J, et al. XRP:In-Kernel Storage Functions with eBPF [C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22). 2022.

[116] GHIGOFF Y, SOPENA J, LAZRI K, et al. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing[C]//18th USENIX Symposium on Networked Systems Design and Implementation, (NSDI'21). 2021.

[117] PARK S, ZHOU D, QIAN Y, et al. Application-Informed Kernel Synchronization Primitives[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22). 2022.

[118] ZHOU Y, WANG Z, DHARANIPRAGADA S, et al. Electrode: Accelerating Distributed Protocols with eBPF[C]//20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23). 2023.

[119] YANG Z, LU Y, LIAO X, et al. λ-IO: A Unified IO Stack for Computational Storage[C]//21st USENIX Conference on File and Storage Technologies (FAST'23). 2023.

[120] QI S, MONIS L, ZENG Z, et al. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing[C]//ACM SIGCOMM 2022 Conference (SIGCOMM '22). 2022.

[121] BRUNELLA M S, BELOCCHI G, BONOLA M, et al. hXDP: Efficient Software Packet Processing on FPGA NICs[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20). 2020.

[122] BONOLA M, BELOCCHI G, TULUMELLO A, et al. Faster Software Packet Processing on FPGA NICs with eBPF Program Warping[C]//2022 USENIX Annual Technical Conference (ATC'22). 2022.

[123] RIVITTI A, BIFULCO R, TULUMELLO A, et al. EHDL: Turning EBPF/XDP Programs into Hardware Designs for the NIC[C]//28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23). 2023.

[124] TIAN D J, HERNANDEZ G, CHOI J I, et al. LBM: A Security Framework for Peripherals within the Linux Kernel[C]//2019 IEEE Symposium on Security and Privacy (SP'19). 2019.

[125] HE Y, ZOU Z, SUN K, et al. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices[C]//31st USENIX Security Symposium (SEC'22). 2022.

[126] MOHAMED M H N, WANG X, RAVINDRAN B. Understanding the Security of Linux eBPF Subsystem[C]//14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'23). 2023.

[127] JIN D, ATLIDAKIS V, KEMERLIS V P. EPF: Evil Packet Filter[C]//2023 USENIX Annual Technical Conference (ATC'23). 2023.

[128] HE Y, GUO R, XING Y, et al. Cross Container Attacks: The Bewildered eBPF

on Clouds[C]//32nd USENIX Security Symposium (SEC'23). 2023.

[129] Blackhat USA, With Friends Like eBPF, Who Needs Enemies[Z]. https://www.blackhat.com/us-21/briefings/schedule/#with-friends-like-ebpf-who-needs-enemies-23619. 2021.

[130] NAGARAKATTE S, ZHAO J, MARTIN M M K, et al. CETS: compiler enforced temporal safety for C[C]//9th International Symposium on Memory Management (ISMM'10). 2010.

[131] 刘翔, 童薇, 刘景宁, 等. 动态内存分配器研究综述[J]. 计算机学报, 2018.

[132] ZENG K, CHEN Y, CHO H, et al. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability[C]//31st USENIX Security Symposium (SEC'22).

[133] XU W, LI J, SHU J, et al. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel[C]//22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15). 2015.

[134] CHEN Y, XING X. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel[C]//2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19).

[135] 杨松涛, 陈凯翔, 王准, 等. 面向缓解机制评估的自动化信息泄露方法[J]. 软件学报, 2018.

[136] PALIT T, MONROSE F, POLYCHRONAKIS M. Mitigating data leakage by protecting memory-resident sensitive data[C]//35th Annual Computer Security Applications Conference (ACSAC'19). 2019.

[137] mm: SLAB freelist randomization [LWN.net][Z]. https://lwn.net/Articles/685047/.

[138] Linux kernel heap quarantine versus use-after-free exploits | Alexander Popov." [Online][Z]. https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html.

[139] "Introduce struct layout randomization plugin [LWN.net]." [Z]. https://lwn.net/Articles/719732/.

[140] grsecurity - How AUTOSLAB Changes the Memory Unsafety Game[Z]. https ://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[141] "[PATCH] slub: Improve bit diffusion for freelist ptr obfuscation - Kees Cook." [Z]. https://www.uwsg.indiana.edu/hypermail/linux/kernel/2003.0/06194.html .

[142] LIN Z, WU Y, XING X. DIRTYCRED: Escalating Privilege in Linux Kernel [C]//2022 ACM SIGSAC Conference on Computer and Communications Security (CCS'22). 2022.

[143] MANÈS V J M, JANG D, RYU C, et al. Domain Isolated Kernel: A lightweight sandbox for untrusted kernel extensions[J]. Computers and Security, 2018.

[144] GÖKTAS E, RAZAVI K, PORTOKALIDIS G, et al. Speculative Probing: Hacking Blind in the Spectre Era[C]//2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20). 2020.

[145] CHEN W, ZOU X, LI G, et al. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities[C]//29th USENIX Security Symposium (SEC'20). 2020.

[146] CABALLERO J, GRIECO G, MARRON M, et al. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities[C]// International Symposium on Software Testing and Analysis (ISSTA'12). 2012.

[147] CHEN Y, LIN Z, XING X. A Systematic Study of Elastic Objects in Kernel Exploitation[C]//2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20). 2020.

[148] "Weaknesses in Linux Kernel Heap Hardening." [Z]. https://blog.infosectcbr.com.au/2020/03/weaknesses-in-linux-kernel-heap.html.

[149] "Bit Flipping Attacks Against Free List Pointer Obfuscation." [Z]. https://blog .infosectcbr.com.au/2020/04/bit-flipping-attacks-against-free-list.html.

[150] LIU D, ZHANG M, WANG H. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping[C]//2018 ACM SIGSAC

Conference on Computer and Communications Security (CCS'18). 2018.

[151] CHEN P, XU J, LIN Z, et al. A practical approach for adaptive data structure layout randomization[C]//20th European Symposium on Research in Computer Security (ESORICS'15). 2015.

[152] KIM J, JANG D, JEONG Y, et al. POLaR: Per-Allocation Object Layout Randomization[C]//49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19). 2019.

[153] SUN H, SHEN Y, LIU J, et al. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation[C]//2022 USENIX Annual Technical Conference (ATC'22). 2022.

[154] LUNA G A D, ITALIANO D, MASSARELLI L, et al. Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries[C]// 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21). 2021.

[155] eBPF - Introduction, Tutorials & Community Resources[Z]. https://ebpf.io/.

[156] EMAMDOOST N, WU Q, LU K, et al. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning[C]//28th Annual Network and Distributed System Security Symposium (NDSS'21). 2021.

[157] HEELAN S, MELHAM T, KROENING D. Automatic Heap Layout Manipulation for Exploitation[C]//27th USENIX Security Symposium (SEC'18). 2018.

[158] MCVOY L W, STAELIN C, et al. lmbench: Portable Tools for Performance Analysis[C]//USENIX annual technical conference (ATC'96). 1996.

[159] MEDIA P. Phoronix test suites: Open-Source, Automated Benchmarking[Z]. https://www.phoronix-test-suite.com/. 2023.

[160] REN X, RODRIGUES K, CHEN L, et al. An Analysis of Performance Evolution of Linux's Core Operations[C]//27th ACM Symposium on Operating Systems Principles (SOSP'10). 2019.

[161] Van der KOUWE E, HEISER G, ANDRIESSE D, et al. SoK: Benchmarking Flaws in Systems Security[C]//2019 IEEE European Symposium on Security and Privacy (EuroS&P'19). 2019.

[162] SUCHY B, CAMPANONI S, HARDAVELLAS N, et al. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation[C]//41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20). 2020.

[163] Fisher–Yates shuffle - Wikipedia.[Z]. https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle.

[164] CVE-2017-7533[Z]. https://nvd.nist.gov/vuln/detail/CVE-2017-7533.

[165] CVE-2010-2959[Z]. https://nvd.nist.gov/vuln/detail/CVE-2010-2959.

[166] Roessler, Nick and Chien, Yi and Atayde, Lucas and Yang, Peiru and Palmer, Imani and Gray, Lily and Dautenhahn, Nathan. Lossless instruction-to-object memory tracing in the Linux kernel[C]//14th ACM International Conference on Systems and Storage. 2021.

[167] CVE-2017-7308[Z]. https://nvd.nist.gov/vuln/detail/CVE-2017-7308.

[168] CORBET J. Super Long-term Kernel Support [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/749530/. 2018.

[169] The Perenial Nuclear Power Plant" example [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/106179/.

[170] LIU C, CHEN Y, LU L. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel[C]//28th Annual Network and Distributed System Security Symposium (NDSS'21). 2021.

[171] XU M, KASHYAP S, ZHAO H, et al. Krace: Data Race Fuzzing for Kernel File Systems[C]//2020 IEEE Symposium on Security and Privacy (SP'20). 2020.

[172] ZOU X, LI G, CHEN W, et al. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel[C]//31st USENIX Security Symposium (SEC'22). 2022.

[173] HAO Y, ZHANG H, LI G, et al. Demystifying the Dependency Challenge in Kernel Fuzzing[C]//44th IEEE/ACM 44th International Conference on Software Engineering (ICSE'22). 2022.

[174] WU W, CHEN Y, XING X, et al. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities[C]//28th USENIX Security Symposium (SEC'19). 2019.

[175] LEE Y, MIN C, LEE B. ExpRace: Exploiting Kernel Races through Raising Interrupts[C]//30th USENIX Security Symposium (SEC'21). 2021.

[176] XU J, MU D, CHEN P, et al. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump[C]//2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16). 2016.

[177] MU D, DU Y, XU J, et al. POMP++: Facilitating Postmortem Program Diagnosis with Value-Set Analysis[J]. IEEE Trans. Software Eng., 2021.

[178] CUI W, GE X, KASIKCI B, et al. REPT: Reverse Debugging of Failures in Deployed Software[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18). 2018.

[179] GE X, NIU B, CUI W. Reverse Debugging of Kernel Failures in Deployed Systems[C]//2020 USENIX Annual Technical Conference (ATC'20). 2020.

[180] BLAZYTKO T, SCHLÖGEL M, ASCHERMANN C, et al. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation[C]//29th USENIX Security Symposium (SEC'20). 2020.

[181] MILLER M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape[Z]. BlueHat. 2019.

[182] Variable-length arrays and the max() mess [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/749064/.

[183] HALLER I, JEON Y, PENG H, et al. TypeSan: Practical Type Confusion Detection[C]//2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16). 2016.

[184] JEON Y, BISWAS P, CARR S, et al. HexType: Efficient Detection of Type Confusion Errors for C++[C]//2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17). 2017.

[185] POPOV A. Alexander Popov's Blog[Z]. https://a13xp0p0v.github.io/. 2023.

[186] SEREBRYANY K, ISKHODZHANOV T. ThreadSanitizer: Data Race Detection in Practice[C]//Workshop on Binary Instrumentation and Applications. 2009.

[187] MU D, WU Y, CHEN Y, et al. An In-depth Analysis of Duplicated Linux Kernel Bug Reports[C]//29th Annual Network and Distributed System Security Symposium (NDSS'22). 2022.

[188] LIN Z, CHEN Y, WU Y, et al. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs[C]//43rd IEEE Symposium on Security and Privacy (SP'22). 2022.

[189] Syzbot. UBSAN: shift-out-of-bounds in __qdisc_calculate_pkt_len[Z]. https://syzkaller.appspot.com/bug?id=70c77abc177053bca3b6ce82cefe5f05ad67c9f2.

[190] Syzbot. KASAN: use-after-free Read in route4_get[Z]. https://syzkaller.appspot.com/bug?id=5bb09c0c5b65ab2ce628ba26fe7cbd06144bd952.

[191] Syzbot. KMSAN: uninit-value in tcp_recvmsg[Z]. https://syzkaller.appspot.com/bug?id=2039c557a4f369ad05baf0c6d0c9b9b8caf3acd5.

[192] Syzbot. KCSAN: data-race in tcp_send_challenge_ack / tcp_send_challenge_ack [Z]. https://syzkaller.appspot.com/bug?id=f6e95af74472292ab1c50af3d6ac36cd4a683432.

[193] KASIKCI B, ZAMFIR C, CANDEA G. Data Races vs. Data Race Bugs: Telling the Difference with Portend[C]//17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12). 2012.

[194] JEONG D R, KIM K, SHIVAKUMAR B, et al. Razzer: Finding Kernel Race Bugs through Fuzzing[C]//2019 IEEE Symposium on Security and Privacy

(SP'19). 2019.

[195] LENHARTH A, ADVE V S, KING S T. Recovery Domains: An Organizing Principle for Recoverable Operating Systems[C]//14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09). 2009.

[196] SIDIROGLOU S, LAADAN O, PEREZ C, et al. ASSURE: Automatic Software Self-healing Using Rescue Points[C]//14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09). 2009.

[197] KADAV A, RENZELMANN M J, SWIFT M M. Fine-grained fault tolerance using device checkpoints[C]//Architectural Support for Programming Languages and Operating Systems (ASPLOS'13). 2013.

[198] SMITH R, RIXNER S. Surviving Peripheral Failures in Embedded Systems[C]//2015 USENIX Annual Technical Conference (ATC'15). 2015.

[199] GU T, SUN C, MA X, et al. Automatic Runtime Recovery via Error Handler Synthesis[C]//31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16). 2016.

[200] BOOS K, VECCHIO E D, ZHONG L. A Characterization of State Spill in Modern Operating Systems[C]//12th European Conference on Computer Systems (EuroSys'17). 2017.

[201] Add Code-generated BPF Object Skeleton Support [LWN.net] — lwn.net[Z]. https://lwn.net/Articles/806911/.

[202] GitHub - Markakd/LLVM-O0-BitcodeWriter: patch for LLVM to generate O0 bitcode — github.com[Z]. https://github.com/Markakd/LLVM-O0-BitcodeWriter.

[203] POPOV A. CVE-2017-2636: Exploit the race condition in the n_hdlc Linux kernel driver[Z]. https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[204] JORN J. Issue 2247: Linux: unix GC memory corruption by resurrecting a file reference through RCU[Z]. https://bugs.chromium.org/p/project-zero/issues/detail?id=2247.

[205] UBSAN: shift-out-of-bounds in dummy_hub_control[Z]. https://syzkaller.appspot.com/bug?id=b5b251b9bcc4653c39164dfee969dafb903ae25e.

[206] Syzbot. KASAN: stack-out-of-bounds Write in bitmap_from_arr32[Z]. https://syzkaller.appspot.com/bug?id=2c09122a1f7edf61aa6fb5dbb6cd19766b5daaa1.

[207] Syzbot. KASAN: slab-out-of-bounds Write in watch_queue_set_filter[Z]. https://syzkaller.appspot.com/bug?id=797c55d2697d19367c3dabc1e8661f5810014731.

[208] Syzbot. KASAN: slab-out-of-bounds Write in sha512_final[Z]. https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.

[209] Syzbot. KASAN: use-after-free Read in vb2_mmap[Z]. https://syzkaller.appspot.com/bug?extid=be93025dd45dccd8923c.

[210] Syzbot. KMSAN: kernel-infoleak in _copy_to_iter (6)[Z]. https://syzkaller.appspot.com/bug?id=e476b01dd5a1075a281c26069ebf677b019bf6d8.

[211] Syzbot. KCSAN: data-race in netlink_recvmsg / netlink_recvmsg (5)[Z]. https://syzkaller.appspot.com/bug?id=cb2264a0f3b303a24e4c4a88752d551e35bae757.

[212] Syzbot. KCSAN: data-race in netlink_getname / netlink_insert (4)[Z]. https://syzkaller.appspot.com/bug?id=a834b993b63ed43938194af3accb08c0a5042877.

[213] WILLIAMS-KING D, GOBIESKI G, WILLIAMS-KING K, et al. Shuffler: Fast and Deployable Continuous Code Re-Randomization[C]/ / 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16). 2016.

[214] PWN2OWN VANCOUVER 2022 - THE RESULTS[Z]. https://www.zerodayinitiative.com/blog/2022/5/18/pwn2own-vancouver-2022-the-results. 2022.

[215] CHECKOWAY S, SHACHAM H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface[C]//18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13). 2013.

[216] PARMER G, WEST R. Mutable protection domains: Towards a component-based system for dependable and predictable computing[C]//28th IEEE International Real-Time Systems Symposium (RTSS'07). 2007.

[217] PARMER G, WEST R. Mutable protection domains: Adapting system fault isolation for reliability and efficiency[J]. IEEE Transactions on Software Engineering, 2011.

[218] GitHub - Bonfee/CVE-2022-0995: CVE-2022-0995 exploit — github.com[Z]. https://github.com/Bonfee/CVE-2022-0995.

[219] GitHub - Crusaders-of-Rust/CVE-2022-0185: CVE-2022-0185 — github.com [Z]. https://github.com/Crusaders-of-Rust/CVE-2022-0185.

[220] GitHub - theori-io/CVE-2022-32250-exploit — github.com[Z]. https://github.com/theori-io/CVE-2022-32250-exploit.

[221] GitHub - plummm/CVE-2022-27666: Exploit for CVE-2022-27666 — github.com[Z]. https://github.com/plummm/CVE-2022-27666.

[222] BALIGA A, GANAPATHY V, IFTODE L. Automatic Inference and Enforcement of Kernel Data Structure Invariants[C]//2008 Annual Computer Security Applications Conference (ACSAC'08). 2008.

[223] MELL P, SCARFONE K, ROMANOSKY S. The common vulnerability scoring system (CVSS) and its applicability to federal agency systems[J]. National Institute of Standards and Technology, 2007.

[224] LYU Y, FANG Y, ZHANG Y, et al. Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis[C]// 2022 IEEE Symposium on Security and Privacy (SP'22). 2022.

[225] LU K. Practical Program Modularization with Type-Based Dependence Analysis[C]//2023 IEEE Symposium on Security and Privacy (SP'23). 2023.

[226] ZHANG Y, PANG C, PORTOKALIDIS G, et al. Debloating address sanitizer [C]//31st USENIX Security Symposium (SEC'22). 2022.

[227] SHWARTZ-ZIV R, ARMON A. Tabular data: Deep learning is not all you need [J]. Information Fusion, 2022.

[228] MOLNAR C, CASALICCHIO G, BISCHL B. Interpretable Machine Learning–a Brief History, State-of-the-art and Challenges[C]//Joint European Conference on Machine Learning and Knowledge Discovery in Databases. 2020.

[229] BREIMAN L. Random forests[J]. Machine learning, 2001.

[230] CHEN T, GUESTRIN C. Xgboost: A scalable tree boosting system[C]//22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016.

[231] BUITINCK L, LOUPPE G, BLONDEL M, et al. API design for machine learning software: experiences from the scikit-learn project[J]. arXiv preprint arXiv:1309.0238, 2013.

[232] BACHL M, FABINI J, ZSEBY T. A Flow-based IDS Using Machine Learning in eBPF[J]. arXiv preprint arXiv:2102.09980, 2022.

[233] NSA. A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission[Z]. https://ghidra-sre.org/. 2023.

[234] Syzbot. general protection fault in nft_tunnel_get_init[Z]. https://syzkaller.appspot.com/bug?extid=76d0b80493ac881ff77b.

[235] Syzbot. KASAN: stack-out-of-bounds in ipip6_tunnel_locate[Z]. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b95211e066fc.

[236] Syzbot. UBSAN: array-index-out-of-bounds in nfnetlink_unbind[Z]. https://syzkaller.appspot.com/bug?extid=4903218f7fba0a2d6226.

[237] Syzbot. KASAN: invalid-free in nf_tables_newset[Z]. https://syzkaller.appspot.com/bug?id=caed28f292ccd32eef950b27d68cf16852303b7f.

[238] POPOV S, MOROZOV S, BABENKO A. Neural Oblivious Decision Ensembles for Deep Learning on Tabular Data[J]. arXiv preprint arXiv:1909.06312, 2019.

[239] KATZIR L, ELIDAN G, EL-YANIV R. Net-dnf: Effective Deep Modeling of Tabular Data[C]//International Conference on Learning Representations. 2020.

[240] GitHub - pqlx/CVE-2022-1015: Local privilege escalation PoC for Linux kernel CVE-2022-1015 — github.com[Z]. https://github.com/pqlx/CVE-2022-1015.

[241] GitHub - randorisec/CVE-2022-34918-LPE-PoC — github.com[Z]. https://github.com/randorisec/CVE-2022-34918-LPE-PoC.

[242] SUCHY B, GHOSH S, KERSNAR D, et al. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation[C]//27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22). 2022.

简历与科研成果

基本情况

王子成，男，汉族，1996 年 12 月出生，内蒙古呼伦贝尔人。

教育背景

2018.9 ~ 2024.6	南京大学计算机科学与技术系	博士
2014.9 ~ 2018.6	吉林大学软件学院	学士

攻读博士学位期间完成的学术成果

第一作者论文

[1] WANG Z, CHEN Y, ZENG Q. PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel[C]//Usenix Security **(CCF-A)**. 2023.

[2] 王子成, 郭迎港, 钟炳南, 等. 基于 eBPF 的内核堆漏洞动态缓解机制 [J].JOS: 软件学报 **(中文 CCF-A)**, 2023: 1-23. DOI: 10.13328/j.cnki.jos.006923.

[3] WANG Z, GUANG Y, CHEN Y, et al. SeaK: Rethinking the Design of a Secure Allocator for OS Kernel[C]//USENIX Security **(CCF-A)**. 2024.

[4] WANG Z, CHEN T, DAI Q, et al. When eBPF Meets Machine Learning: On-the-fly OS Kernel Compartmentalization. arXiv:2401.05641. 2024.

其他贡献论文

[1] GUO Y, **Zicheng Wang**, ZHONG B, et al. Formal Modeling and Security Analysis for Intra-level Privilege Separation[C]//ACSAC **(CCF-B)**. 2022.

[2] ZHONG B, **Zicheng Wang**, GUO Y, et al. CryptKSP: A Kernel Stack Protection Model Based on AES-NI Hardware Feature[C]//IFIP SEC **(CCF-C)**. 2022.

[3] SUN R, GUO Y, **Zicheng Wang**, et al. AttnCall: Refining Indirect Call Targets in Binaries with Attention[C]//ESORICS **(CCF-B)**. 2023.

攻读博士学位期间参与的科研课题

[1] 基于分散可信基的内核保护方法研究，国家自然科学基金

[2] 新型通信网络系统的安全态势分析基础理论和方法研究，国家自然科学基金

致　　谢

首先，我要感谢我的导师曾庆凯教授。作为我进入学术世界的引路人，他不仅塑造了我的思维模式和研究方法，还启发我理性地认识世界，深入探索事物的基本原理。这些宝贵的知识与启示，将是我一生中持续受益的精神财富。

我也非常感谢科罗拉多大学博尔德分校的陈越琦教授，是他慷慨资助了我在美国的学术交流之旅。这段经历极大地拓展了我的学术视野，让我有机会深入了解国际前沿的研究动态，同时也言传身教让我理解如何做好研究。

对于那些与我风雨同舟的同学们，我也要表达我的诚挚感谢。感谢南京大学的郭迎港、钟炳南、孙锐、苏超、郭锐、尧利利、李佳杰、金乃柱，科罗拉多大学的林明浩、戴钦润，亚利桑那州立大学的陈铁今。每一次的交流与讨论，都是我宝贵的学习经历，让我们一起在科研的道路上不断前行。

我还要特别感谢我的家人，是他们的爱、鼓励和支持，构成了我追求学术梦想的坚强后盾。在我最需要支持的时候，家人总是我的坚实依靠。

最后，希望未来更多漏洞能被实时防御，我们的生活能够事事顺心。