
# 用户空间旁路：加速系统调用密集型应用

***复旦大学的***周哲、毕燕翔、万俊鹏和周扬帆***；***
周莉，***加州大学欧文分校***

https://www.usenix.org/conference/osdi23/presentation/zhou-zhe

本文收录于第 **17** 届 **USENIX** 操作系统设计与实现研讨会论文集。

**2023** 年 **7** 月 **10-12** 日 **-** 美国马萨诸塞州波士顿

978-1-939133-34-2

第 **17** 届 **USENIX** 操作系统设计与实现研讨会论文集》的开放获取由以下机构赞助


# 用户空间旁路：加速系统调用密集型应用

周哲、毕燕翔、万俊鹏、周扬帆

*复旦大学*

*{zhouzhe, 19210240167, 19210240003, zyf}@fudan.edu.cn}*

周莉

*加州大学欧文分校*

*zhou.li@uci.edu*

## 摘要

内核模式和用户模式之间的上下文切换通常会造成显著的开销，从而降低频繁系统调用（或系统调用）应用程序的运行速度，例如那些 I/O 需求较高的应用程序。而 Linux 内核页表隔离（KPTI）等安全机制又进一步加大了开销。为了加快此类应用的运行速度，很多人都在努力从 I/O 路径中移除系统调用，主要方法是将驱动程序和应用程序合并到同一空间或批处理系统调用。然而，这些解决方案要求开发人员重构应用程序，甚至更新硬件，这阻碍了它们的广泛应用。

在本文中，我们提出了另一种方法，即用户空间旁路（UB），通过将用户空间指令透明地移入内核来加速系统调用密集型应用程序。用户空间旁路不需要修改用户空间二进制文件或代码，就能实现完全的二进制兼容。具体来说，为了避免频繁的系统调用造成的开销，内核会识别连续系统调用之间的用户空间执行路径，并将路径中的指令转换为具有软件故障隔离（SFI）保证的代码块。根据我们的评估，在开启 KPTI 的虚拟化环境中执行应用程序时，I/O 微基准性能可提高 30.3 - 88.3%，Redis GET 每秒请求数（RPS）可提高 4.4 - 10.8%（数据大小为 1B - 4KiB）。关闭 KPTI 后，性能提升幅度将减小。

## 1 导言

系统调用（syscall）被用户空间应用程序广泛用于访问主机操作系统（OS）提供的资源，并广泛用于 I/O 操作。然而，当开启 Linux 内核页表异或（KPTI）[47] 等机制时，系统调用可能会产生明显的性能过剩[43]。可以说，系统调用是追求每秒高 I/O 请求（IOPS）的应用程序（如请求超过一百万 IOPS 的应用程序）的主要性能瓶颈之一[7]。

**系统调用重构方法。**在最近的文献中，通过改变 I/O 路径处理系统调用的方式来实现更高的 IOPS 主要有两种方法，我们称之为系统调用重构方法：1) 第一种方法是通过将数据处理逻辑移入内核[26, 36, 53]，或将负责 I/O 的驱动程序移入用户空间（内核旁路）[21, 51]，从而将驱动程序和数据处理逻辑整合到同一地址空间。这样，处理逻辑就可以直接与 I/O 设备对话，避免了在用户模式和内核模式之间切换造成的开销 [51]。2) 第二种方法是批量调用系统调用，允许用户空间进程对多个 I/O 请求进行排队，只需一次系统调用即可发出这些请求 [43]。然而，这些解决方案要求开发人员修改代码，而这通常是一项非同小可的任务。

**我们的方法**在本文中，我们提出了*用户空间旁路*（简称 UB），它可以减少系统调用相关 I/O 带来的开销，同时实现*二进制兼容*（即无需更改或重建应用程序代码）。之所以采用 UB，是因为观察到高 IOPS 应用程序在两次连续系统调用之间不会执行很多指令（见第 3.1 节）。因此，我们可以在预定义的安全要求下，在两次系统调用之间透明地执行指令（即把指令翻译成经过消毒的代码块），然后让内核退出这些代码块，而不返回用户空间。这样就可以避免连续系统调用造成的开销。图 1 举例说明了这一想法。

然而，有几个难题需要解决。首先，只有那些有可能在*频繁*调用的连续系统调用之间执行的指令才值得用户空间旁路。但是，如果没有开发人员提供的明确信息，就很难找到这样的系统调用序列。由于将指令提升到内核也会带来开销，因此需要谨慎选择要优化的系统调用，以抵消这种开销。其次，恶意程序可能会利用 UB 窃取内核数据，甚至执行特权指令。此外，存在漏洞的应用程序可能会污染内核内存。因此，UB 必须保证内核数据的安全性。

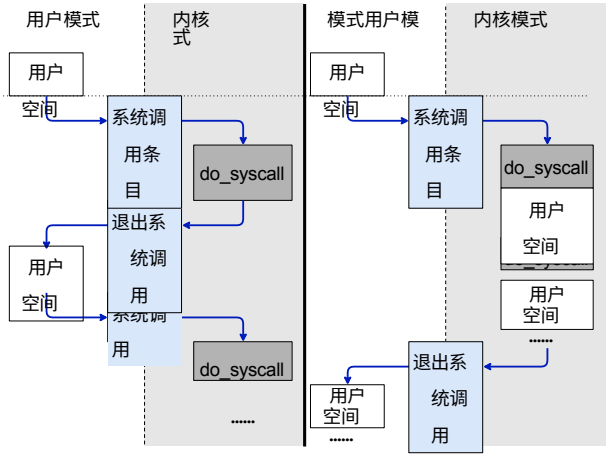



图 1： 在没有 UB 和有 UB 的情况下调用系统调用。

通过对用户空间的代码和数据进行全面的清理，来确保安全性。最后，为了实现二进制兼容性，升级后的应用程序代码应不考虑是在内核模式还是用户模式下执行。应确保有 UB 和无 UB 时的执行结果完全相同，包括多线程应用程序的内存顺序和原子性。

我们通过调整动态二进制转换（DBT）[52] 和基于软件的故障隔离（SFI）[44] 技术来应对这些挑战。首先，我们通过挂钩系统调用条目来了解哪些系统调用调用频繁（即 "热 "系统调用）。受即时编译（JIT）[17, 22, 48]的启发，我们可以获取运行时热门系统调用之后的用户空间指令。如果这些指令属于同一函数，则会被转化为二进制缓存（BTC）。接下来，我们将迭代执行 BTC，并从退出指令开始扩展 BTC，直到遇到下一次系统调用。我们执行指令和地址清除以限制 BTC 的行为，并在 BTC 上实现控制流完整性（CFI）和数据完整性。UB 不对指令重新排序，也不分割内存访问。因此，其他线程可以与经过 UB 优化的线程同时安全地执行。

我们实现了 UB 的原型，并评估了其在 I/O 微基准和实际应用（包括 Redis 和 Nginx）中的性能增益。在默认设置下（测试应用程序在虚拟机（VM）中运行，Linux KPTI 处于开启状态），I/O 微基准线程的加速度为 30.3% 到 88.3%。对于 Redis `GET`，在数据大小为 1B - 4KiB 的情况下，加速率为 4.4% - 10.8%。Nginx 的加速率为 0.4% - 10.9%。UB 可将基于原始套接字的数据包过滤器加速 31.5% - 34.3%。我们还评估了 KPTI 和虚拟化对 UB 性能提升的影响。由于关闭 KPTI 会减少系统调用开销，因此 UB 的效果较差。例如，I/O 微基准测试的加速率从 88.3% 降至

降至最小 I/O 大小的 41.6%。因此，未来的处理器有望在硬件上消除 Meltdown 和 Spectre 漏洞，而 UB 的优势将大打折扣。当应用程序在物理机中运行时，与虚拟机相比，UB 在大多数情况下都能达到更高的上限加速比，因为在这种情况下 IOPS 通常更高，从而产生更多可优化的系统调用。

在实验研究中，我们还将 UB 与其他优化系统调用的系统进行了比较，包括 io_uring [23]、F-Stack DPDK [45] 和 eBPF [34]。结果表明，在微基准测试中，UB 与 io_uring 相比优势较小；在 Redis 宏基准测试中，UB 与 F-Stack 相比优势较小；在原始套接字测试中，UB 与 eBPF 相比优势较小。不过，UB 有一个独特的优势，即应用程序开发人员无需修改代码。

最后，我们承认 UB 可能会在侧信道、未注明的 x86 指令和内核竞赛下引入新的安全风险。因此，我们提出了一些防御建议。

我们的 UB 原型代码发布在 [15] 网站上。我们将本文的贡献总结如下。

- 我们提出了用户空间旁路 (UB)，在内核模式下直接执行系统调用之间的指令，以加速系统调用。
- 我们提供了一种具体的设计，可以将用户空间指令透明地转换为内核安全、经过净化的 BTC。使用这种方法，现有应用程序无需修改即可执行，并享受性能提升。
- 我们实施了一个原型，并针对几个高 IOPS 应用程序进行了评估。结果证明了 UB 的有效性。

## 2 背景介绍

在本节中，我们首先概述了系统调用机制及其引入的开销。然后，我们将介绍之前在减少此类开销方面所做的努力。

### 2.1 系统调用及其成本

Syscall 是用户空间应用程序与内核服务之间的默认接口。可以利用软件中断（如 `int 0x80`，已被淘汰）和特殊指令（如 AMD 创建的 `syscall/sysret` 和 Intel 创建的 `sysenter/sysexit`），在 syscall 之后将控制权从用户空间转移到内核空间，反之亦然。

以往的研究表明，调用系统调用会给各种应用和场景带来显著的开销[16, 35, 43]，包括*直接成本*和*间接成本*[43]。对于第一种情况，由于在用户模式和内核模式之间切换，必须执行额外的程序来保存寄存器、更改保护域和处理已注册的异常。对于后一种情况，处理器结构的状态，包括 L1 缓存数据和指令

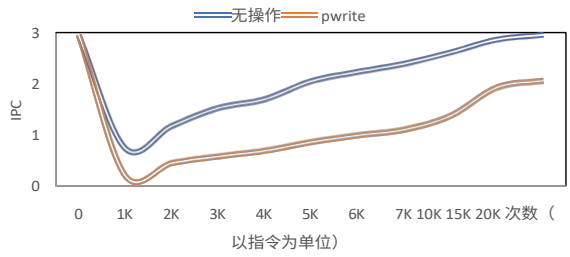



图 2：在我们的平台（英特尔 Skylake 和 Linux）上测得的无操作系统调用和 `pwrite` 系统调用后的 IPC。

系统调用可对高速缓存、翻译侧向缓冲区（TLB）等进行*控制*，CPU 的失序执行（OOE）必须停滞以保证顺序。因此，在系统调用后，用户模式的每周期指令数（IPC）会减少。

被广泛使用的内核页表隔离技术（KPTI）[47] 使系统调用速度更慢。为了抵御瞬时执行攻击（如 Meltdown [29] 和 Spectre v3a [49]），操作系统内核为用户空间和内核空间使用了两套页表。因此，CPU 应在进入系统调用时切换到内核页表，并在返回用户空间时切换回内核页表。除了 KPTI，虚拟化也可能增加上下文切换开销。例如，虚拟机内 TLB 未命中的开销（间接开销的一部分）可能会更大，因为需要检查的页表项比物理机内的更多。

下面我们将总结以往研究的观察结果和我们对具体系统调用开销的测量结果。

- 根据 Mi 等人在英特尔 Skylake 和 seL4[35]上的测量，启用 KPTI 的无操作系统调用会耗费 431 个 CPU 周期。
- 在我们的实验平台（英特尔 Skylake 和 Linux）上测得，内核序幕和尾声（直接成本）的无操作系统调用需要 197 个指令（992 个 CPU 周期），这表明在 Soares 等人的研究[43]十年后，系统调用开销问题依然存在。
- 同样在我们的平台上，`pwrite` 系统调用会将下列用户空间指令的 IPC 从 2.9 降至 0.2（间接成本）。在执行 *20,000 条*指令后，IPC 缓慢回到 2.1。图 2 显示了 IPC 随时间变化的趋势。

| 计划 | 开发费用 | 异步需要 | Accele-定量 | 人口-理智 |
|---|---|---|---|---|
| **eBPF** | ++ | C | ++ | ++ |
| **DPDK** | ++ | C | + | ++ |
| **io_uring** | ++ | C | + | + |
| **统一内核** | +++ | - | +++ | - |
| **FlexSC** | + | - | + | - |
| **UB** | - | - | + | |

## 2.2 系统调用性能优化

研究界正积极致力于减少系统调用带来的开销。下面我们将介绍相关工作，并将其与我们的方法进行比较（表 1 也进行了总结）。

**异步系统调用。**系统调用引入了同步执行模式，因为在系统调用完成后，用户模式的执行将被恢复。布朗提出的非阻塞式 Linux 系统调用[5]可以异步并行完成。

表 1： 优化系统调用方案的比较。

到用户空间执行流程。但是，这种方法并不能将系统调用的调用与执行完全分离。迄今为止，Linux 上的大多数系统调用实现仍然是同步的。

**系统调用批处理。**由于本地性是性能的一个主要因素，因此也有人对批量执行系统调用进行了研究。Ra- jagopalan 等人提出将连续的系统调用分组为一个系统调用（一个系统调用的结果直接反馈给下一个系统调用）[38]。这种方法在两个系统调用之间不发生连接的假设条件下是有效的。Soares 等人建议对多个共同程序的系统调用进行批处理，并要求开发人员将线程模型改为 M-on-N（"M 个用户模式线程在 N 个内核可见线程上执行，M 个用户模式线程在 N 个内核可见线程上执行，N 个内核可见线程在 M 个用户模式线程上执行，M 个用户模式线程在 N 个内核可见线程上执行）。
"N") [38].因此，只有当任务可以分割成多个线程时，它才会起作用。现代内核提供了本地队列，即 io_uring [23]，用于批量处理来自用户空间进程的 I/O 请求，减少系统调用的发生。特别是，用户空间代码可以向队列发出多个请求，并调用一个系统调用让内核处理队列。[1].

**Unikernel**为了减少上下文切换的开销，Unikernel 解决方案在内核空间而不是用户空间运行应用代码。这方面的例子包括可加载内核模块（LKM）[42] 和库操作系统[31, 40]。

**内核沙箱**为减少系统调用引起的上下文切换，内核沙箱允许应用代码在特权模式下运行。例如，eBPF [34] 允许开发人员将代码附加到内核跟踪点。当内核到达这些点时，它将使用虚拟机来执行附加代码。不过，eBPF 对代码有许多限制，内核会在执行前验证是否满足所有要求，在此期间，合法代码可能会因为验证中的误报而被拒绝。最近，Dmitry 等人提出使用内核沙箱，完全在内核中执行应用程序 [27]，这样也可以减少上下文切换的开销。

**内核旁路。**一些研究人员注意到，内核不必总是参与数据包处理等 I/O 任务，因此提出了内核旁路方法。一个突出的例子是数据平面开发套件

[1] io_uring 还支持内核轮询模式（如果应用程序具有 root 权限，则无需调用系统调用）。

(DPDK) [11]，它接管了用户空间中的 I/O 设备。具体来说，I/O 请求通过共享环形缓冲区提交给设备，而不是系统调用。缓冲区在用户空间内维护，不涉及内核的 I/O 活动。

我们发现，现有的方法都需要大量的开发工作。要使用系统调用重构基元，必须遵循不同的编码范式。大多数内核旁路和系统调用批处理解决方案（如 DPDK、RDMA、io_uring）都要求应用代码与队列对异步交互。尽管如此，开发人员仍然喜欢用同步方式编写程序逻辑。改造传统代码也是一项劳动密集型工作。例如，我们将支持 DPDK 的非官方 Redis [2] 与其官方版本（3.0.5 版）进行了比较。我们发现，前者为支持 DPDK 额外增加了 9984 行代码（LoC），占官方版本 LoC 的 *10%*。另一个例子是 Unikernel：它要求开发人员编写内核模式代码，不幸的是，这种代码很难调试，而且容易出现内存损坏等错误（没有内存隔离）。除了修改应用程序代码，绕过内核的解决方案可能还需要特殊的用户空间驱动程序[24]。

## 3 设计概述

为了解决上述问题，我们提出了用户空间旁路（UB）这一系统调用优化的新方法。UB 旨在实现以下三个设计目标 (DG)。

**DG1：最大限度地减少开发人员的手工劳动。**系统调用重构方法要求开发人员更改遗留代码或调整为异步编程，与之不同的是，UB 在*执行时*优化系统调用，同时不影响应用程序的功能。

**DG2：尽量减少对系统架构的改变。**系统重构方法可能会改变当前的系统架构，例如将设备映射和绑定到用户空间。相比之下，UB 保持当前系统架构不变，包括设备驱动程序和 I/O 采集模型。

**DG3：与系统调用重构方法性能相当。**UB 的目标是减少系统调用的直接和间接成本，并实现与系统调用重构方法类似的性能提升。

### 3.1 系统调用密集型应用

我们专注于优化高 IOPS 应用程序，如 Redis 和 Nginx，它们也是系统集成密集型应用程序。通过分析它们的代码和运行时行为，我们发现以下两点对 UB 的设计具有指导意义。

**I/O 线程中的轻量级用户空间指令。**我们发现，I/O 事件之间的计算工作量对于所研究的应用程序来说通常是轻量级的。此外，两个连续系统调用之间的指令数量也很低。

通常很小。一种解释是，此类应用程序采用流行的 I/O 模型，将 I/O 密集型工作负载与 CPU 密集型工作负载分隔在不同的线程中。例如，Redis 服务器有一个主线程，该线程向 I/O 线程分配所接受的套接字[10]，I/O 线程执行内核间的 I/O，让主线程完成 CPU 密集型计算。在这种设计下，I/O 事件之间的指令只需处理缓冲区的移动。我们还分析了 Redis 调用的系统调用（共计 3M），发现其中一半在下一次系统调用之前的用户空间指令少于 400 条（IPC 为 2 时约为 200 个周期），这比执行系统调用本身还要快（如第 2.1 节所述的 431 个周期 [35]）。

**放大的直接和间接成本。**第 2.1 节概述了一般系统调用的直接和间接成本，这些成本在系统调用密集型应用中会被放大。如图 1 所示，进入和退出的频率会随着系统调用频率的增加而增加。由于 TLB 未命中、OOE 停顿和缓存未命中造成的间接成本也不容忽视，尤其是当系统调用处理的任务较轻时（如图 2 所示，IPC 在无操作系统调用中降至 0.74，在 `pwrite` 中降至 0.21）。

### 3.2 UB 模块

基于上述考虑，我们设计 UB 的动机是让它能够检测系统调用的发生，并通过二进制转换将连续系统调用之间的用户空间指令提升到内核。图 1 展示了我们的想法。虽然从高层次上看，这个想法似乎很简单，但要将 UB 应用于现实世界的成熟应用程序，还需要解决一些难题。

- 与内核代码相比，应用代码的可信度较低。因此，在转移到内核后，应执行必要的*隔离*以限制其能力。然而，识别不信任区域并用正确的策略对其进行管理并非易事。
- 考虑到隔离会产生额外成本，转换每块用户空间指令并不总是有益的。但是，何时进行转换以及如何减少转换的开销都是未知数。

UB 通过三个关键部分应对这些挑战。

1) 热门 "系统调用标识符，用于监控目标程序的执行情况，描述调用的系统调用，并确定何时需要提升用户空间指令；

2) 3) *内核 BTC 运行时*，执行翻译后的代码。图 3 概述了 UB 的发展历程。

请注意，UB 中的组件从根本上来说并不是新概念。BTC 是动态二进制转换 (DBT) [3, 22] 的标准组件。JIT 译码器在以下方面遵循基于软件的故障隔离（SFI）[44] 准则

代码工具和隔离策略。然而，我们发现现有系统无法直接用于我们的问题设置。下面我们将简要讨论 UB 的主要模块。

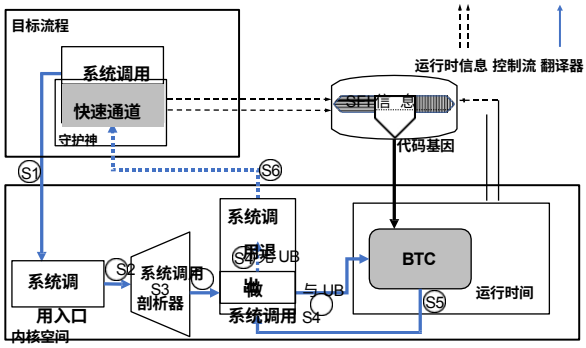



图 3： UB 框架概览。每次线程调用系统调用（S1）时，热系统调用标识符都会将其挂钩（S2），并将其打补丁到 `do_syscall`（S3），之后，如果系统调用不是热的，内核可能会返回用户模式（S6），或者将其发送到 BTC 运行时进行 UB（S5）。

**热门系统调用标识符。**该模块在内核模式下运行，并钩住每个系统调用。通过分析运行时的统计数据，它可以识别哪些系统调用指令是*热门指令*，即很有可能在短时间内紧跟着另一个系统调用的指令。在两个连续的热门系统调用之间的用户空间指令将被提升到内核，并在应用程序下次运行时被加速。为避免运行时监控带来大量开销，该模块会间歇运行。

**BTC 翻译器。**BTC 译码器将之前模块标记的用户空间指令转换为 BTC，并由内核 BTC 运行时执行。根据 SFI 准则，它将危险指令（如间接控制流传输）转换为安全指令（如直接跳转），并通过工具检查来限制内存访问和控制传输行为。翻译器在独立的用户空间进程中运行，以避免将其代码引入内核。翻译不会阻止应用程序的执行，*下次*访问相同的代码路径时，翻译后*的*代码会被执行。

除了优化一对热系统调用之间的用户空间指令外，我们还考虑对一连串热系统调用进行加速。我们称这种封权衡 BTC 翻译器的翻译和仪表化开销。我们根据不同的用户空间路径长度（即指令数量）来衡量性能增益，并考虑路径较*短*的区域。主要原因是，由于需要监控的指令数量较多，因此较长路径的检测成本会迅速增加。我们认为
1 000 个指令（称为 $T_{path}$ ）作为短时
闭的用户空间代码为*快速路径*。UB 的目标是将这些用户空间代码串联起来，并对它们进行加速。快速路径是通过观察跳转目标逐步发现的。具体细节将在第 5 节讨论。

## 4 热系统调用标识符

**绕过用户空间的标准。**当用户空间内结构化区域的性能增益超过用户空间旁路区域的性能增益时，就应提升该区域的性能。

路径长度[2].通过实证研究，我们发现使用这一路径长度可以明显提高性能（超过 20%）（见第 6.2 节）。

**模块设计。**该模块旨在发现包含较短用户空间路径的热门系统调用。我们通过在线分析实现无缝剖析。具体来说，该模块挂钩系统调用入口，并计算两个连续系统调用之间的指令数。当指令数小于 $T_{path}$ 时，这两个系统调用就被归类为热系统调用候选。下面我们将介绍具体步骤。

- **系统调用抽样。**监控每一次系统调用都会给应用程序的执行带来很高的性能损失。因此，我们对系统调用进行抽样，只在线程（如 I/O 线程）频繁发出系统调用时才进行后续分析。根据我们对系统调用密集型应用程序（如 Re- dis 和 Nginx）的测量，每秒至少发出 10 万次系统调用（称为 $T_{sys}$ ）（每分钟 600 万次），而我们选择对 $T_{sys}$ 中不到 10% 的系统调用（每分钟最多 50 万次）进行剖析。因此，大多数系统调用不会被采样，也不会受到干扰。
- **粗粒度剖析。**为进一步降低预处理开销，我们会检查受监控线程是否频繁调用系统调用。如果线程每秒调用的系统调用次数少于 50K（$T_{sys}$ 的一半），模块将不进行下一次细粒度系统调用分析。在这种情况下，IOPS 低的线程将被跳过。
- **细粒度剖析。**对于频繁调用系统调用的线程，该模块会进一步分析哪些系统调用指令被频繁调用。频繁调用的指令值得绕过用户空间，因为这样可以获得更多性能提升。我们监控每轮的 15K 次系统调用（占 $T_{sys}$ 的 15%），并为每条调用的系统调用指令维护一个表格，记录其位置寄存器（`RIP`）和在 4 微秒（约为执行 $T_{path}$ 指令的时间）内下一条系统调用指令被调用次数的计数器。当计数器大于 900 次（占 15000 次系统调用的 6%）时，我们认为系统调用频繁。这些系统调用及其附带的用户空间指令将在下一阶段由 BTC 译码器处理。

有人可能会问，该模块的性能是否对参数选择敏感。为了测试其敏感性，我们

[2] Soares 等人认为，如果一个系统调用每 2,000 个或更少指令被调用一次，则该调用是频繁的[43]。为了适应不同平台，我们使用了一个更为保守的数字。

检查在三台不同机器上能否正确发现 Redis 和 Nginx（我们实验中使用的两个应用程序）的热系统调用：一台使用 Core i5 10500 的 PC（2021 年），两台使用 Xeon 8175 和 8260 的服务器（2017 年和 2019 年）。所有热系统调用都能被正确识别，这表明在大多数情况下可以跳过参数调整。

## 5 BTC 运行时和翻译器

本节将介绍 BTC 转换器如何将用户空间指令转换为内核 BTC 并满足安全要求。我们的 BTC 译码器遵循动态二进制转换（DBT）程序 [19, 22, 48]。一般来说，给定一个由二进制基本块组成的路径，并触发一个事件（例如，在我们的案例中的热系统调用），DBT 将其分解，用 SFI 规则手册进行翻译，并编译成 BTC 供将来执行。由于采用了 SFI，翻译后的代码中的恶意或不需要的行为可以被遏制，并由 BTC 运行时安全地运行。

### 5.1 BTC 运行时间

翻译后的代码块由内核中的 BTC 运行时执行。BTC 运行时在内核堆栈中保存局部变量，BTC 中的仪器指令可以访问这些局部变量，以执行策略和进行上下文切换。局部变量包括1) 保存的内核上下文，即调用保存的寄存器，2) 保留寄存器的值，以及
3) 间接跳转目的地信息，用于建立快速路径。

在执行 BTC 之前，运行时会为用户空间准备好返回状态，即恢复系统调用入口（如 x86_64 的 `pt_regs`）上保存的用户空间上下文。程序块执行完毕后，运行时会处理 BTC 的返回状态，并采取进一步措施。当跳转目标丢失（例如遇到新路径）时，BTC 的执行可能会中途退出运行时。在这种情况下，运行时会记录这次跳转的信息，并立即返回用户空间，即跳转目标。我们允许 BTC 运行时访问用户空间内存，因此内存上的所有变化都会被保留下来。对寄存器的更改会更新到用户空间上下文（即 x86_64 的 `pt_regs`），当内核重新转向用户空间时，这些更改将写入寄存器。因此，BTC 所做的用户空间状态更改也会被保留下来，并对其他线程可见，从而确保应用程序逻辑不会在 UB 下发生更改。当遇到系统调用指令时，BTC 的执行也可能退出。在这种情况下，两个连续系统调用之间的快速路径已在线程中完全执行，这表明用户空间旁路成功。BTC 运行时通过在系统调用表中查找系统调用编号并将系统调用参数分派给相应的 `do_syscall` 函数（即执行系统调用）来模拟系统调用陷阱。在 `do_syscall` 返回后，BTC run-

运行时会检查下一条系统调用指令是否又是热的。如果答案是肯定的，运行时将尝试执行另一个用户空间旁路。这样，`do_syscall` 和用户空间旁路就可以串联起来，这与 DBT 的直接分支串联类似。在理想情况下，*整个线程都可以在内核中执行*。

**快速路径发现。**UB 的性能在很大程度上取决于快速路径识别的准确性，我们利用增量、JIT 类型的方法来实现高准确性。给定一个入口地址（即热系统调用旁边的指令），BTC 译码器首先从入口地址迭代分解目标线程的代码段，从而发现快速路径的一部分。在每次迭代中，翻译器都会跳过可能无法到达的路径。具体来说，翻译器只跟随*直接跳转*，并在调用指令处停止，这就迫使翻译器在一次迭代中只处理函数内的代码，并将其视为快速路径。当以后确实发生间接跳转或调用时，BTC 运行时将收集目标信息并发送给翻译器，以便在替换跳转指令后扩展快速路径（见第 5.2.1 节）。这种方法类似于 QEMU [9]，但我们并不将二进制提升为中间表示。

## 5.2 BTC 翻译器

下面我们将介绍如何将安全策略植入用户空间代码。我们遵循 SFI 原则，在*内核*中*提供数据访问策略*和*控制流策略*[44]，并继承和扩展了 Nacl [52]的实现，后者在浏览器中对不受信任的 x86 本地代码进行了沙箱处理。值得注意的是，Nacl 假定源代码可用，因此 SFI 规则可在静态编译下执行。相比之下，UB 对二进制文件执行 DBT。因此，必须调整和扩展 SFI 规则。

**威胁模型。**我们假设用户空间代码是不可信任的，可能包含*任意*代码和数据，其副作用包括对内核内存、特权函数等的无中介访问。UB 的目标是确保用户空间代码在提升到内核后不能获得更多权限（并造成更多危害），即保护内核控制流的完整性。值得注意的是，这一目标与保证用户空间应用的控制流完整性[1]不同（详见第 5.3 节）。在设计 UB 时，我们采取了一种*保守的*方法，在无法立即确定后果时（例如，在翻译过程中跳转目标未知），避免提升快速路径。我们将重点放在 x86_64 平台上，但所提出的技术可以很容易地推广到其他平台上。下面我们将介绍与跳转、寄存器、指令和内存访问相关的实现方法，以确保该威胁模型下的安全性。

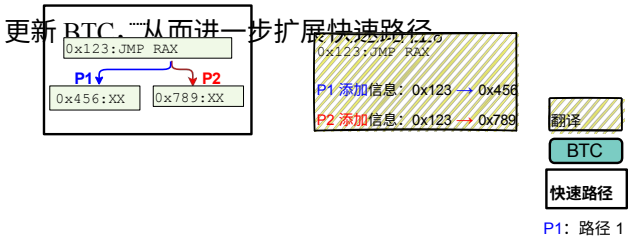
### 5.2.1 跳转卫生设施

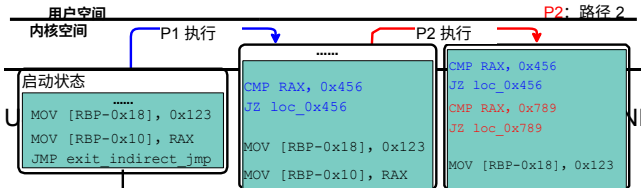
Nacl 的内部沙箱检查调用和跳转所表达的*显式*控制流，并禁止对间接跳转和调用指令进行内存删除。跳转的目标被限制在沙箱内。相比之下，在 UB 下，整个内核内存空间都对提升的用户空间代码开放。因此，我们采取了不同的方法来净化跳转。

**直接跳转。**为防止 BTC 中的代码跳转到任意地址，只有在跳转目标已知的情况下才进行转换。换句话说，只有目标已知的直接跳转才会被处理。地址净化将在第 5.2.3 节中介绍。

**间接跳转。**然而，用户空间快速路径可能包含间接跳转，我们阻止 BTC 在目标已知之前处理此类路径。特别是，翻译器会插入检查，在首次遇到相关代码时将目标*地址与目标地址表*（类似于 jumptable [22]）进行比较。如果目标地址不在表中，控制流将退出 BTC 运行时。在执行 BTC 代码块期间触发这种退出时，BTC 运行时会将跳转指令地址（即地址的 `RIP`）和目标地址发送给 BTC 翻译器，并扩展快速路径，如第 5.1 节所述。

我们在图 4 中展示了一个示例。间接跳转（跳转到 `RAX`，位于 `0x123`）最初被翻译为写下跳转目标（将 `RAX` 保存到堆栈）并退出 BTC 运行时（跳转到 `exit_indirect_jmp`）。当路径 `P1` 首次执行时，BTC 运行时会学习目标 `0x456`，并将信息发送给翻译器，翻译器通过添加目标表项更新 BTC。之后，路径 `P1` 被添加到 BTC 中，下次不会再触发 `exit_indirect_jmp`。如果稍后到达 `P2`，则可以学习另一个目标 `0x789`，并更新 BTC，从而进一步扩展快速路径。




整个快速通道。插入 BTC 的检查之所以能高效执行，是因为1) 根据我们对系统调用密集型应用的经验分析和先前的研究[18]，间接控制流传输指令并不经常出现；2) 允许 CPU 在无序执行下推测性地跳转到目的地，而无需等待目的地检查。

### 5.2.2 重新映射寄存器

为了保护内核寄存器和堆栈，BTC 译码器不允许 BTC 代码访问堆栈寄存器（即 `RSP`、`RBP` 和 `RIP`）。此外，有些寄存器是为 BTC 运行时保留的，BTC 代码也不能访问。因此，我们开发了这个模块来管理这些寄存器。

具体来说，BTC 转换器使用 BTC 中的 $M$ *个*预留寄存器来处理对 $N$ *个*寄存器的潜在访问（$N = M + 3$，其中 3 个用于堆栈寄存器）。当 $M < N$ 时，转换器需要对寄存器进行调度。$N$ *个*寄存器的值存储在局部变量中，译者从 $M$ *个*保留寄存器中选择一个暂时充当特殊寄存器，并*重新命名*。译者还插入代码，使 $N$ *个寄存器*与栈上的局部变量同步。因此，BTC 代码的行为与用户空间中的快速路径相同。

**保留寄存器。**翻译器将 R12- `R15`（$M = 4$）保留给 BTC 运行时使用，因为它们在普通用户空间应用中使用频率最低（使用频率低于 1%[18]）。当它们出现在快速通道中时，重命名就会发生。我们还对常用特殊寄存器（即 `RSP`）的重命名机制进行了优化，让翻译器固定保留寄存器以保持其值。这样可以减少代价高昂的寄存器同步操作。

### 5.2.3 教学消毒

不允许在 BTC 中使用特权指令（如 `sysret`），以避免利用 UB 的恶意代码造成特权升级。在翻译过程中，如果包含任何特权指令，翻译器会避免将快速路径提升到内核。

由于寄存器重映射，有些指令必须重写。对于像

PU
SH
/P
OP
这

样
的
堆
栈



图 4： 跳转消除下的翻译示例。

随着应用程序运行时间的延长，可以学习到更多的间接跳转焦油。由此产生的 BTC 最终可以覆盖

操作指令，翻译器会用多条指令来代替它们。以 POP 为例。翻译器首先添加一条指令，将操作数从保留寄存器（即堆栈指针）插入的内存中 MOV 到弹出的目标，然后用新的堆栈指针值（即加 8 ）更新保留寄存器。

#### 5.2.4 内存访问净化

为防止未经授权访问内核内存，翻译器会对所有内存访问指令进行消毒处理。对于 ev-

与 SFI [44] 的地址屏蔽类似，翻译器将地址左移一位，然后右移一位，以满足地址要求。为此引入了两条额外的指令（即 SHL 和 SHR），但我们的评估表明，额外的开销可以忽略不计（0.4%）。需要注意的是，增加的检查并不能防止 BTC 访问*未映射的*内存区域并触发页面故障，我们将通过下面描述的程序来处理。

**页面故障处理。**我们修改了页面故障处理程序，以监控页面故障事件。对于次要故障和主要页面故障，页面故障处理程序在内核模式和用户空间模式下的表现相同。因此，由用户空间应用程序引起的故障与没有 UB 时的解决方式相同。当无效页面故障（即非法访问某些内存区域）发生时，BTC 代码的执行将被中止。

Nacl 还借助 x86 CPU 提供的分段功能，隔离了扩展程序与主机浏览器之间的内存空间。因此，扩展程序的指令只能访问段内的内存，不允许修改段状态的指令。不过，虽然 x86_64 仍提供分段功能，但它只在地址中添加分段偏移量，而不检查分段边界，因此不能直接用于内存隔离。

## 5.3 安全保证

翻译后的 BTC 具有以下安全属性（称为 SP），它们共同使 UB 实现内核上的 SFI 策略[44]。

**SP1：用于 BTC 的内核控制流完整性（CFI）。之所以**能保证这一特性，是因为当 BTC 运行时将控制流移交给用户空间时，执行只能通过退出点终止。更重要的是，当运行时执行 BTC 时，线程不会跳转到翻译器未知的位置。对于直接控制流传输，目的地只能是已翻译的已知基本模块的标签。通过替换目的地，间接控制流传输都会被翻译成直接传输。因此，BTC 可以防止恶意代码在提升内核控制流后对其进行劫持。

我们要指出的是，UB 并未声称增加了针对控制流劫持（如 ROP、JOP、COOP [4, 6, 8, 37, 41]）的例外保护，它们仍然可能发生在用户空间。虽然攻击者可以在通过目的地检查时构建小工具，但绝不允许从 BTC 跳转到内核代码段，因为这会被翻译器检测到并终止。

**SP2：内核数据（内存和寄存器）完整性。**对于内核上下文（或寄存器），我们设计的 BTC 运行时符合调用惯例，并且调用方（内核）

在跳转到 BTC 之前，上下文会保存在堆栈中，在返回内核指令之前，上下文会被恢复。上下文切换是轻量级的，因为它不会导致权限转移。

对于内核内存，访问禁制可确保任何内核内存都不会被禁制指令访问，因此内核堆栈不会被玷污。虽然运行时局部变量必须能被 BTC 中的指令访问，但恶意程序不能利用它们来触及内核堆栈。只有有意插入的指令才能触及堆栈基指针引用的局部变量，而堆栈基指针存储的是运行时信息，如交换出的寄存器（见第 5.2.2 节）。由于内核 CFI 是有保证的，因此执行时绝不会跳转到这些指令。

**SP3：BTC 中没有特权指令。**第 5.2.3 节对此有解释。

**SP4：死循环中断。**我们还考虑了针对系统资源*可用性的*攻击和漏洞。例如，用户空间应用程序可能会因错误或故意而陷入死循环。作为对策，翻译器会在 BTC 运行时维护一个计数器，以跟踪已执行指令的数量。一旦计数器超过阈值，执行流就会退出运行时，进而返回用户空间，从而避免内核被 BTC 代码阻塞。

## 5.4 螺纹安全

应特别注意多线程用户空间应用程序，因为除了提升到内核的线程外，UB 无法控制其他线程。必须保持内存顺序和原子性，以避免数据竞赛。幸运的是，线程安全是由翻译器自动保证的，我们将在下文对此进行说明。

**内存顺序。**为了保持内存顺序，译码器将所有用户空间内存视为*易失性*内存，只在用户空间指令之间插入结构，而不对程序块进行优化（例如，对指令重新排序或在寄存器中缓存内存修改）。然而，CPU 仍可根据其内存模型对内存加载和存储重新排序。用户空间应用程序放置的原始内存栅栏会被全部继承，翻译器不会插入额外的栅栏。

**原子性。**在使用多条指令模拟一条用户空间指令时，翻译器会采取特别措施保证原子性。在翻译指令时，翻译器倾向于使用与原始指令具有相同操作码的指令。因此，原始指令的原子性会自动得到保留。例如，带有锁前缀的指令会被翻译成仍带有锁的指令（如 `LOCK MOV`）。如果需要仿真多条指令，内存加载或存储必须在一条指令中完成。例如，在翻译 `PUSH RIP` 时，必须将下一条指令的偏移地址移至

用户空间栈顶。从翻译器的角度来看，当 BTC 运行时到达该指令时，`RIP` 的值是已知的，并成为一个即时数字。然而，x86_64 并没有直接将 64 位中间值移动到内存的指令。因此，翻译器生成的指令首先将立即值移至 64 位保留寄存器，然后将 64 位寄存器移至用户空间堆栈顶部。

| | 测试 | 虚拟机 | 物理 |
|---|---|---|---|
| w/ PTI | **内存** | 30.3% - 88.3% | 38.4% - 112.9% |
| | **Redis GET** | -3.7% - 10.8% | -5.4% - 6.4% |
| | **Redis SET** | -0.4% - 12.4% | -3.2% - 16.1% |
| | **Nginx** | 0.4% - 10.9% | -1.4% - 13.4% |
| | **插座** | 31.5% - 34.3% | 30.9% - 38.6% |
| 无 PTI | **内存** | 14.3% - 41.6% | 16.4% - 52.0% |
| | **Redis GET** | -2.0% - 4.6% | -6.4% - 3.9% |
| | **Redis SET** | -5.5% - 4.9% | -0.9% - 2.8% |
| | **Nginx** | -1.2% - -0.3% | -0.2% - 3.0% |
| | **插座** | 14.5% - 17.8% | 9.2% - 19.8% |

## 6 评估

我们在 Linux 内核 5.4.44 中实现了 UB 的原型。BTC 运行时以内核模块的形式实现，包含 416 行 C 代码，它挂钩系统调用后记以进行系统调用识别和管理 BTC 运行时。翻译器在用户空间由 786 行 Python 代码实现（除了依赖的 Python 反汇编器 `miasm` 和 gcc 汇编器 `as`），它通过 sys 文件与 BTC 运行时内核模块通信。只需在 syscall 条目中添加 6 行代码即可修改内核，使模块能够挂钩 syscall。

我们在 I/O 微基准和两个实际应用（Redis 和 Nginx）的宏观基准中评估了我们的原型。我们还将其与 DPDK、io_uring 和 eBPF 等相关技术进行了比较。为了评估这些应用，我们建立了一个虚拟化环境和一个裸机环境。裸机环境由一台客户机和一台服务器组成。[3] 连接在 40G 以太网局域网内。虚拟化环境在服务器上运行，并启用网卡直通功能。对于微基准 I/O 实验，我们直接在服务器上运行测试，因为它不需要网络。在其他情况下，我们在客户端机器上运行客户端应用程序，在服务器机器上运行服务器应用程序，因此流量通过物理网络。虚拟化和 KPTI 会影响系统调用性能（如第 2.1 节所述），为显示虚拟化和 KPTI 的影响，我们在四种情况下运行每个服务器应用程序：KPTI 开/关 × 虚拟机/物理机。当 KPTI 打开时，Linux 会打开 PCID 以减轻性能下降。以下所有测试均进行 10 轮，并显示平均 IOPS 或每秒请求（RPS）值。对于第 6.1 节至第 6.4 节展示的结果，我们将重点放在开启 KPTI 的虚拟机设置上，并简要介绍其他设置下的结果变化。表 2 列出了不同设置下的加速比。

[3] 服务器机器配备英特尔至强 8175 CPU（24 核）、192GB 内存、三星 980 pro NVME 固态硬盘和 Mellanox Connectx-3 网卡。它运行内核为 5.4.44 的 Ubuntu 20.04。当设置为虚拟机时，它使用 QEMU-

表 2：不同设置下的加速比范围。"In-mem " 指内存文件访问基准。

## 6.1 I/O 微型基准

我们首先考虑加速一个纯粹通过阻塞系统调用执行文件 I/O 请求的线程作为微基准，它近似于 UB 的最佳情况。该线程运行一个紧密循环，通过 `READ` 系统调用从内核到用户空间缓冲区顺序读取文件 839 万次。现实世界中的应用程序可能会表现出不同的模式，如在连续的 I/O 请求之间执行更多的指令，从而降低 UB 的加速率。为了进行比较，我们使用 io_uring (liburing-2.2) 执行相同任务（即紧循环 `READ` 系统调用）并比较 IOPS。

**内存文件访问。**我们在 ramfs 中创建了一个大文件，以避免可能出现的磁盘瓶颈，从而更准确地评估 UB 如何加速系统调用。诚然，这种设置使得微基准测试不那么真实。我们逐渐增大每次读取的缓冲区大小，并评估 UB 在不同缓冲区大小下的加速比。

图 5 显示了结果。在开启 KPTI 的虚拟环境中，UB 可将基于系统调用的 I/O 加速 88.3% 。$_{\pm 0.75\%}$[4] 当 I/O 大小较小时（64B ），UB 可将基于系统调用的 I/O 加速 88.3% 。当 I/O 大小增大时，由于调用的系统调用次数减少，UB 和基线的 IOPS 都会下降，加速比下降到 30.3%$_{\pm 0.96\%}$（4KiB I/O 大小）。关闭 KPTI 会增加 IOPS，但由于系统调用开销减少，UB 的加速比降至 14.3%$_{\pm 1.83\%}$ - 41.6%$_{\pm 1.73\%}$。由于物理机上的 IOPS 较高，UB 节省了更多上下文切换开销，因此物理机上的加速度更高，尤其是当 I/O 大小较小时（例如，当 I/O 大小为 64B 时，开启 KPTI 后，$_{\pm 1.78\%}$，加速度为 112.9%）。

对于 io_uring ，我们首先检查了从 1 到 1024 的不同队列深度（即可以分批处理多少个请求），发现如图 6 所示，在深度达到 128 后，IOPS 保持稳定。因此，我们将深度设为 128 ，以便与 UB 进行比较。从图 5 可以看出，在大多数缓冲区大小下，io_uring 的 IOPS 都更高。当运行在物理

KVM 1:4.2-3，并为虚拟机分配了 24 个内核。客户机有一个英特尔至强 8260 CPU、128GB 内存和 Mellanox Connectx-5 网卡。

[4] 我们报告加速度比和标准偏差。

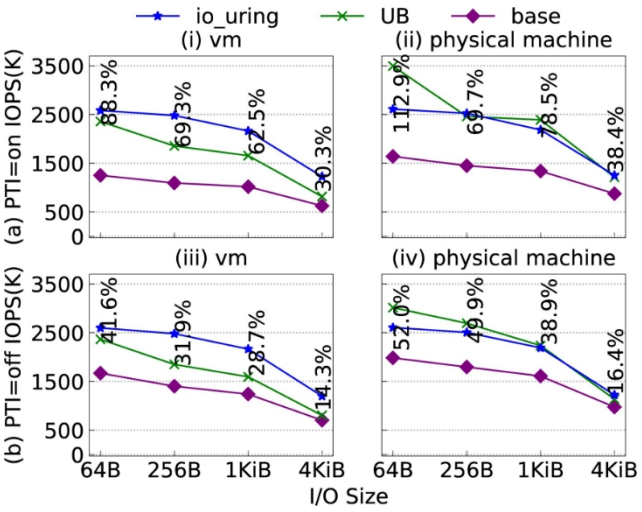



图 5：针对不同缓冲区大小的 READ syscalls、io_uring 和 UB syscalls 的 IOPS。百分比数字是 UB 与基线的加速比。图 7、图 8、图 9 和图 10 遵循了这一风格。

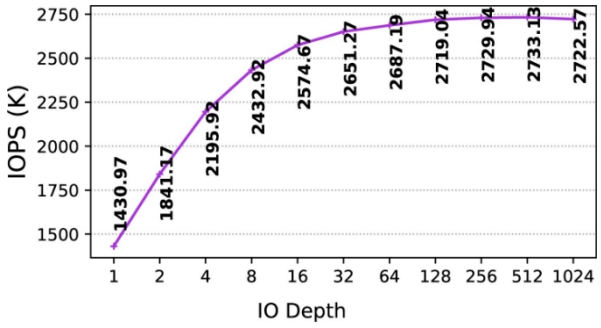



图 6： 不同 io_uring 深度的 IOPS。

在小尺寸（64 字节）的机器上，UB 的 IOPS 要高于 io_uring，尽管我们预计 io_uring 的性能应该总是优于 UB。我们没有找到很好的解释，但我们注意到，当 Linux 内核从 5.4.44（我们测试环境使用的版本）升级到 5.15 时，io_uring 的 IOPS 增加了 13%。因此，io_uring 有可能在更新的 Linux 系统上持续优于 UB。

**NVMe 上的文件访问。**我们还测试了在 NVMe 磁盘上以 1KiB 块大小读取文件的情况，对比结果见表 3（"w/o sum"）。我们只考虑了物理机设置，因为当虚拟机访问虚拟 NVMe 磁盘中的文件时，文件会自动事先缓存到内存中，其行为与内存中的文件访问类似。IOPS 可从 $779_{\pm 5}$ K 提高到 $852_{\pm 3}$ K，从而使$_{\pm 0.3\%}$ 加速 9.4%（KPTI 开启）。当 KPTI 关闭时，基线 IOPS 增加到 $810_{\pm 22}$ K，而 UB 则略微增加到 $858_{\pm 10}$ K，使加速比更小。

我们还考虑了 I/O 线程执行以下操作的情况

轻量级计算，如解析数据包。当连续 I/O 请求之间的关系具有依赖性时，就不能对请求进行分批处理。具体来说，我们设置 I/O 线程在从内核获取缓冲区后，将其视为 64 位整数数组来计算缓冲区的总和。

| | 无和 | 带和 |
|---|---|---|
| **KPTI 关于** | $779_{\pm 5}$ (852 )$_{\pm 3}$ | $630_{\pm 4}$ (793 )$_{\pm 3}$ |
| **KPTI 关闭** | $810_{\pm 22}$ (858 )$_{\pm 10}$ | $686_{\pm 65}$ (795 )$_{\pm 6}$ |

表 3： 在物理机上读取 NVMe 磁盘上文件（1KiB 大小）的 KIOPS（不含求和），以及通过整数求和读取文件的 KIOPS（含求和）。括号中显示的是 UB 加速数。

如表 3（"w/sum"）所示，即使是轻量级计算也能减少大量 IOPS。基线 IOPS 下降了 149K，而开启 UB 时只下降了 59K，因为用户空间中的轻量级计算可以完全移植到内核中执行，因此其 IPC 受系统调用的影响较小。

## 6.2 Redis

我们选择流行的键值存储引擎 Redis 作为一个宏基准，以测试 UB 如何处理真实世界的工作负载。我们使用内置的 Redis- 基准工具 [39] 评估 Redis 6.2.6，以生成工作负载。我们以默认配置运行 Redis 服务器，并以 2 个线程启动 Redis- 基准。连接数保持默认值 50。在每一轮中，客户端发出 100 万个请求。

默认情况下，Redis 的大部分工作都在主线程内完成，主线程不仅负责 I/O，还负责散列等计算任务。在正常工作流程中，主线程会调用 `EPOLL` 获取可读套接字列表。对于每个可读套接字，线程都会读取套接字，然后处理请求。因此，READ 之后的用户空间路径很长（从 3k 到 20k），因为计算任务都在这里进行。最后，Redis 会逐个将响应写入相应的套接字，中间只需少量指令（约 300 条）。

**结果**图 7 显示了 `GET` 和 `SET` 数据的 RPS（有 UB 和无 UB），数据大小从 1B 到 16KiB。在开启 KPTI 的虚拟机中测试 `GET` 时，当数据大小小于或等于 4KiB 时，加速比从 $4.4\%_{\pm 1.52\%}$ 到 $10.8\%_{\pm 2.69\%}$。当数据大小上升到 16KiB 时，加速比降至$-3.7\%_{\pm 0.51\%}$。关闭 KPTI 后，加速比降至 $-2.0\%_{\pm 1.32\%}$ 和 $4.6\%_{\pm 1.96\%}$ 之间。负加速比表明，UB 带来的开销超过了其本身节省的系统调用开销。在物理机上执行的情况则不同：在 KPTI 上执行的加速比为$-5.4\%_{\pm 1.17\%}$ 到 $6.4\%_{\pm 2.01\%}$，在$_{\pm 3.02\%}$ 上执行的加速比为-6.4% 到 3.9%。$_{\pm 1.67\%}$
为 KPTI 关闭。值得注意的是，Redis 的 RPS 要小得多

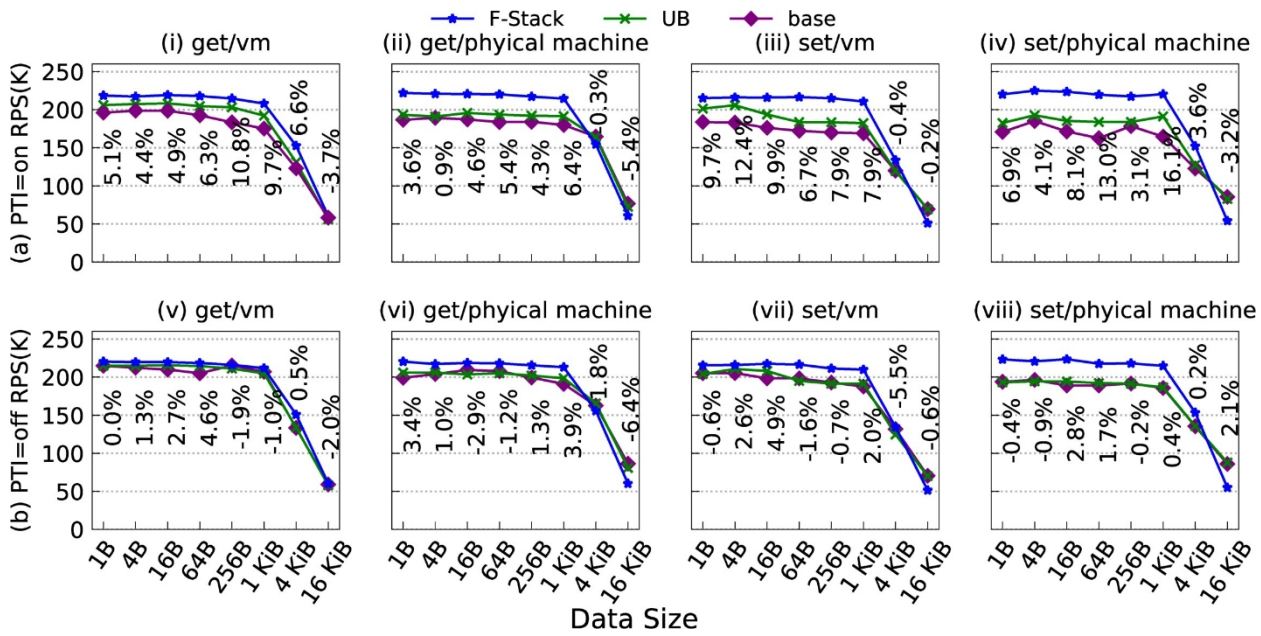



图 7：不同数据量下 Redis `GET` 和 `SET` 的 RPS。

因此，系统调用所产生的费用并不是主要因素。因此，加速比要小得多。

在 `SET` 方面，加速度比从 $-0.4\%_{\pm 2.19\%}$ 至 $12.4\%_{\pm 3.96\%}$ 的 VM 与 KPTI 上。模拟当 KPTI 关闭并在物理机上运行时，也能观察到这一趋势。值得注意的是，当 `SET` 和 `GET` 的数据大小超过 1KB 时，Redis RPS 会显著下降，Redis 基准的官方文档[39]中也有类似的观察报告。

令人惊讶的是，我们发现虚拟机的 RPS 经常高于物理机，尽管虚拟设置本应产生较低的 RPS。我们无法很好地解释 Redis 为什么会出现相反的情况。

| | **Redis 服务器** | **BTC** | **用户空间** | **做写** |
|---|---|---|---|---|
| **无 UB** | 108.57 | - | 34.29 | 36.73 |
| **w/ UB** | 102.98 | 2.42 | 28.38 | 36.07 |

表 4：用于 Redis 各部分的时间（VM+KPTI、`SET`）。

**剖析性能增益。**我们让 BTC 运行时使用 `RDTSCP` 指令对 BTC 执行和用户空间执行进行剖析。我们运行了 20M Redis `SET` 事务（约 100 秒），其中包括 UB 和不包括 UB。表 4 显示了结果。我们可以看到，通过将快速通道提升到 BTC，可以节省 5.91 秒的用户空间时间，而 BTC 仅花费 2.42 秒。其中的差异（3.49s）可归因于用户空间 IPC 的增加（间接开销）。总共节省了 5.59 秒（"Redis 服务器 "一栏），通过调用更少的系统调用直接节省了 2.1 秒（即 5.59 秒 - 3.49 秒）。

**内存检查的开销。**当不需要强内核内存安全性时，例如当二进制文件经过形式验证时，用户可能会选择删除为检查内存边界而插入的指令（即 `SHL` 和 `SHR`），以追求更高的性能增益。我们评估了如果要求翻译器不插入此类指令，能获得多少 RPS 增益。结果显示，RPS 只提高了 0.4%。

**与 DPDK 的比较。**我们将 UB 在 Redis 上的加速比与在 DPDK 上的加速比进行了比较，因为有一些开源实现可以为 Redis 赋能，如 Redis-DPDK [2] 和 F-Stack Redis [45]。我们选择了 F-Stack，因为 Redis-DPDK 的维护工作已于 2017 年停止，而且它无法在最新的 CPU 上运行。F-Stack 支持最新的 Redis 6.2.6 [46] 以及最新的 DPDK 20.11。对比结果也如图 7 所示。

事实证明，F-Stack 可持续为小容量（不大于 4KiB）提供更高的加速比。有趣的是，我们发现对于 16KiB，F-Stack 的性能比 UB 和 Redis 基线差。一种可能的解释是，F-Stack 没有从我们的多核设置中获益。当我们测量 CPU 使用率时，F-Stack 的 CPU 使用率总是 100%，但 UB 和基线的 CPU 使用率可高达 124%，这意味着使用了多个内核。因此，当我们将内核数限制为 1 时，F-Stack 的性能可能会持续优于 UB。

## 6.3 Nginx

除 Redis 外，我们还使用 Nginx（1.20.0 版）作为另一个宏基准，它是一种流行的静态网络服务器，具有很高的 RPS。表 5 显示了每个系统调用所遵循路径的指令数。表 5 显示了每个系统调用所遵循指令的路径数量。

| 系统调用 | 从 | openat | fstat | 设置时钟选项 | 写 | 发送文件 | 关闭 | 设置时钟选项 |
|---|---|---|---|---|---|---|---|---|
| # 遵循指示 # # 遵循指示 # # 遵循指示 # # 遵循指示 # # 遵循指示 # # 遵循指示 | 4,328 | 38 | 4,412 | 43 | 177 | 541 | 477 | 509 |

表 5： Nginx 每次系统调用后的指令数。少于 1,000 条指令的调用为热调用。两个 setsockopt 调用不同。

因此，8 个中有 6 个可以加速。因此，8 个中有 6 个可以加速。我们在客户端机器上运行 HTTP 基准工具 wrk [50]（4.1.0 版，有 8 个线程和 1024 个连接），向 Nginx 服务器发出 12 秒的请求，以检验 Nginx 能处理

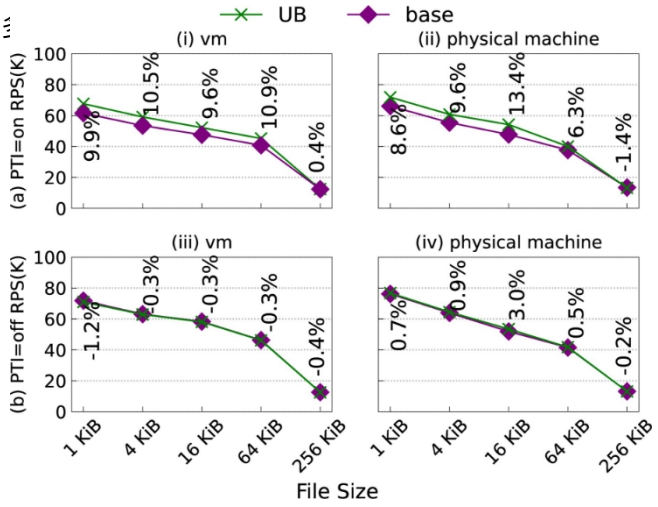



图 8： Nginx 的 RPS 与不同文件大小（字节）的对比。

**结果**我们逐步增加 wrk 请求的文件大小，图 8 显示了 UB 加速前后的 RPS。在开启 KPTI 的虚拟机中进行测试时，Nginx 对 1KB 至 64KB 文件的加速率为 $9.6\%_{\pm 1.81\%}$ 至 $10.9\%_{\pm 0.22\%}$，但对 256KB 文件的加速率则降至 $0.4\%_{\pm 0.86\%}$。对于物理机，1KB 至 64KB 文件的加速比率为 $6.3\%_{\pm 0.17\%}$ 至 $13.4\%_{\pm 3.32\%}$，但 256KB 文件的加速比率也降至 $-1.4\%_{\pm 0.28\%}$。这些结果表明，大文件的瓶颈从系统调用转移到了 I/O。当关闭 KPTI 时，UB 不会产生明显的加速效果。

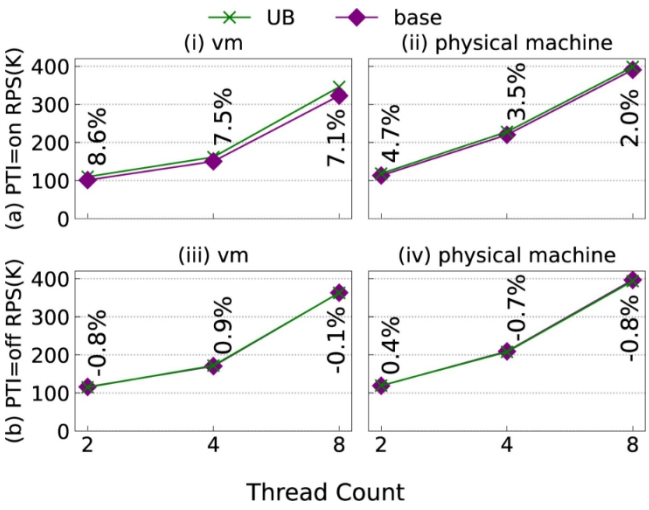



图 9：Nginx 的 RPS 与不同线程数的关系。

**多个工作线程。**我们评估了多线程对加速比的影响。我们逐步增加 Nginx 的工作线程数，并评估文件大小为 4KB 的情况。图 9 显示了 RPS。我们可以看到，随着工作线程数从 2 个增加到 8 个，当 KPTI 启用时，加速比明显下降（虚拟机从 $8.6\%_{\pm 0.22\%}$ 降至 $7.1\%_{\pm 0.17\%}$，物理机从 $4.7\%_{\pm 0.15\%}$ 降至 $2.0\%_{\pm 0.26\%}$）。工作线程越多，用于线程同步的周期就越多，因此每个线程可处理的请求就越少，从而减少了 UB 所节省的系统调用开销。

## 6.4 原始插座与 eBPF

为了避免系统调用开销，eBPF 是另一种流行的解决方案，如第 2.2 节所述。我们将展示，在 UB 的帮助下，开发人员只需在用户空间中使用*原始套接字*编写处理逻辑，并将每秒数据包数（PPS）与 eBPF 进行比较。

我们在客户端机器上运行一个程序，向服务器发送 UDP 数据包，服务器通过原始套接字或 XDP（用于数据包处理的 eBPF 库）处理传入的数据包，每轮 12 秒。客户端运行 15 个线程，会使服务器饱和。处理任务包括通过将数据包视为整数数组来计算数据包数量和数据包总和。

**结果**图 10 显示了 3 种数据包大小（128B、512B 和 1472B）的结果。对于开启 KPTI 的虚拟机，eBPF 在处理小数据包时的性能比原始套接字高出 $368.4\%_{\pm 8.92\%}$。对于 MTU 大小（即 1472B）的数据包，eBPF 的 PPS 仍高出 $236.7\%_{\pm 4.15\%}$。UB 对原始套接字的加速度为 $31.5\%_{\pm 0.25\%}$ - $34.3\%_{\pm 0.72\%}$，远低于 eBPF。在不同数据包大小的情况下，原始套接字的 PPS 是相似的。但是，eBPF 对数据包大小非常敏感，我们认为这是因为原始套接字的瓶颈是协议栈处理，而 eBPF 绕过了这一瓶颈，其瓶颈可能是数据移动，而数据移动的耗时与数据包大小有关。当 KPTI 关闭时，对于不同大小的数据包，UB 的加速比降至 $14.5\%_{\pm 0.45\%}$ - $17.8\%_{\pm 0.44\%}$。在物理机上，UB 的加速率为 0.5%。

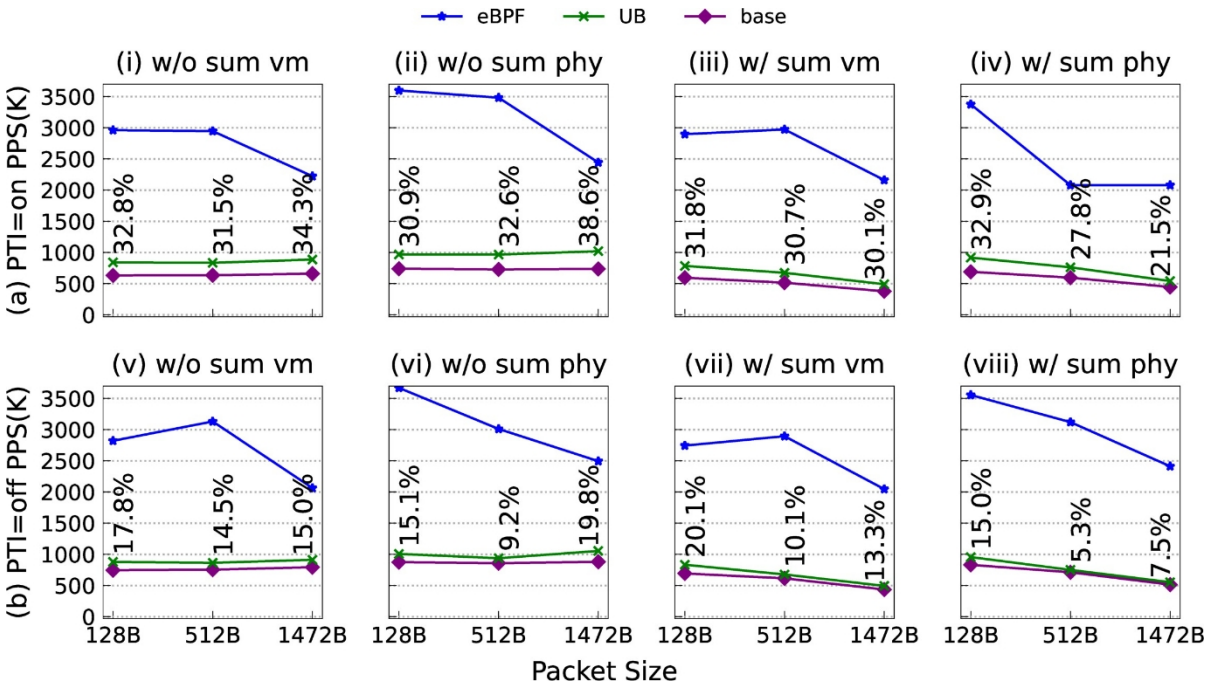



图 10：服务器处理不同大小 UDP 数据包的 PPS。

范围较大（KPTI 开启时，$_{\pm 0.87\%}$ - 38.6%$_{\pm 0.56\%}$；KPTI 关闭时，$_{\pm 0.14\%}$ - 19.8%$_{\pm 0.31\%}$）。

**计算。**我们还考虑添加轻量级计算工作负载，即数据包求和，就像 NVMe 文件访问实验一样（第 6.1 节）。在虚拟机中，当数据包大小增加时，原始套接字的 PPS 下降幅度更大，但 UB 仍能以类似的比率加速原始套接字（KPTI 为 30.1%$_{\pm 0.20\%}$ - 31.8%$_{\pm 0.50\%}$，KPTI 为 10.1%$_{\pm 0.18\%}$ - 20.1%$_{\pm 0.15\%}$）。eBPF 在不进行数据包求和的情况下也能保持相似的 PPS。在物理机上，原始套接字、UB 和 eBPF 也观察到类似的趋势，只是 eBPF 在 512B 数据包大小和 KPTI 开启时 PPS 有相当大的下降。

**剖析执行性能。**我们分别分析了 BTC 和 eBPF 在使用 `RDTSCP` 进行数据包求和时的执行时间，就像我们在 Redis 上的实验一样（第 6.2 节）。在开启 KPTI 的虚拟机中，BTC 处理 3385 万个 128B 的传入数据包耗时 5.86 秒。相比之下，eBPF 花费了 9 秒。可以看出，BTC 的执行性能优于 eBPF 虚拟机。但是，基于之前的结果，UB 仍然无法达到与 eBPF 相似的 PPS。根据我们的分析，原因在于 eBPF 在 softirq 中运行，因此数据包可以被分派到不同的内核中。相比之下，原始套接字协议栈有用于并发访问的内核锁。特别是，我们为套接字读取增加了更多线程，但 PPS 并没有增加。我们还尝试评估 eBPF 在没有多线程的情况下是如何工作的，方法是将网卡的 IRQ 限制为单核，并在开启 KPTI 的虚拟机中重复总和实验。经 UB 加速的套接字分别达到 1M、0.96M 和 1M。

三种数据包大小的 PPS 分别为 0.93M、0.93M 和 0.91M，而 eBPF 的 PPS 分别为 0.96M、0.93M 和 0.91M。因此，我们认为，如果内核针对并发访问优化其协议栈，原始套接字的 PPS 可以得到显著改善。一种可能的方法是构建更好的 UB 运行时，以便通过系统调用暴露更多更深入的内核跟踪点。

# 7 讨论

## 7.1 UB vs. eBPF

除了对 UB 和 eBPF 的性能进行比较外，我们还在此对它们的限制和安全保证进行比较。由于 eBPF 主要是为数据包处理和内核跟踪而开发的，因此它对应用代码有许多限制。例如，eBPF 并非图灵完备，因为它不允许无限循环 [33]。由于对代码的大量限制，eBPF 校验器容易产生误报，即合法代码被视为非法代码[14]。UB 不会对开发者增加任何限制，并能透明地翻译用户空间代码。

在性能方面，UB 只能加速系统调用后的路径，而 eBPF 可以连接到内核中的许多跟踪点，这使得它更灵活，更有能力克服内核瓶颈。我们相信，如果内核通过系统调用公开更多的跟踪点，UB 可以实现与 eBPF 类似的性能。

在安全性方面，eBPF 依赖于与内核虚拟机的隔离，而 UB 则依赖于 SFI 翻译器的策略。如第 7.2 节所述，针对 eBPF 的攻击可能对 UB 也有效。正式验证 eBPF 和 UB 的实现可以缓解这些问题，但验证 eBPF 可能比验证 UB 更容易，因为 eBPF 有官方规范，而且它使用的指令集更少。

## 7.2 安全风险

尽管我们遵循 SFI 原则来设计 UB，但可能会引入新的安全风险。首先，UB 可能会受到侧信道攻击的影响，因为侧信道攻击会根据微架构状态的变化来推断秘密。例如，Spec- tre 攻击证明 eBPF 可被用来窃取内核内存，因为 eBPF VM 可将用户空间代码编译成内核代码[25]。UB 的 BTC 也可能被用于类似的攻击。为降低这种风险，应考虑针对投机攻击的防御措施，例如编译器放置投机阻塞指令[25]。其次，我们的 BTC 译码器可能无法清除未编译的 X86 特权指令。为了降低引入的风险，翻译器可以允许指令白名单。当遇到白名单之外的指令时，UB 应该放弃提升其快速路径。第三，先前的研究表明，内核竞赛会导致从检查时间到使用时间（TOCTOU）的攻击[28]。由于 BTC 运行时并不强制快速路径的检查点和使用点之间保持原子性，恶意用户空间代码可以利用内核竞赛。缓解方法可以依靠现有的主动检测内核竞赛的防御措施 [20]。

## 7.3 其他限制

诚然，当开发人员采取正确措施将 DPDK 等内核旁路框架集成到用户空间应用程序中时，它们可以获得比 UB 更好的性能。更好的性能不仅来自于上下文切换开销的减少，还来自于用户空间驱动程序的简化和高效。例如，用户空间驱动程序可以避免不必要的缓冲区复制、中断等。相比之下，UB 只减少了上下文切换的开销。UB 的主要优势在于不需要开发人员对应用程序进行任何修改（见表 1）。因此，我们认为，如果开发人员愿意重构代码或设计新的应用程序时考虑到内核旁路，内核旁路就会受到欢迎。

UB 并不旨在取代异步 I/O。诚然，当应用程序既是计算密集型又是 I/O 密集型时，异步 I/O 可以帮助开发人员在不同的线程中将 I/O 与计算解耦，从而更好地利用多核。与异步任务相比，UB 并不能为同步 I/O 任务提供更多的 IOPS，但它可以与异步 I/O 结合使用。在某些情况下，异步 I/O 的 I/O 线程会比同步 I/O 的 IOPS 高。

慢性任务仍然需要大量调用系统调用来提交 I/O，而 UB 可以加速这些任务。

## 8 相关工作

第 2 节介绍了有关系统调用优化的相关工作。下面我们将介绍其他相关工作。

**动态二进制转换（DBT）。**DBT 是一种强大的调试和检测方法 [3、19、22、48]。Ke- dia 等人在内核中提出了一种快速 DBT，用于检测内核代码 [22]。我们的翻译器在间接分支处理方面与他们有一些相似之处，但我们的翻译器在内存保护和寄存器重命名方面有很大不同。此外，他们运行时的某些功能需要回滚。相比之下，我们的运行时从不回滚。

**基于软件的故障隔离（SFI）。**在内核中执行 SFI 并不是一个全新的想法。XFI 最早提出用 SFI 隔离内核模块，后来 LXFI 增加了内核 API 检查，以限制通过内核 API 传播的故障 [13, 32]。UB 以不同的方式将 SFI 用于快速路径。

**加速进程间通信（IPC）。**最近提出了一些利用硬件辅助加速 IPC 的方案。与加速系统调用类似，这些方案也试图尽量减少上下文切换开销。Gu 等人提出利用英特尔处理器的最新技术（即 MPK）加速 IPC [16]。Mi 等人借用为虚拟化设计的硬件功能来加速 IPC [35]。Du 等人提出在不涉及内核的情况下为 CPU 的上下文切换添加新功能[12]。他们在 RISC-V FPGA 处理器上实现了原型。

## 9 结论

系统调用带来的开销在高 IOPS 应用程序中非常突出，但现有的方法并没有完全解决这个问题，因为它们需要努力重构代码。为了保持二进制兼容性，我们提出了直接在内核中执行用户空间指令的用户空间旁路（UB）方案。UB 采用 JIT 翻译器，将系统调用之间的用户空间指令翻译成经过净化的代码块。这些代码块受到限制，以避免引入额外的危害，因此可以直接在内核中执行。使用 UB，I/O 微基准测试可加速 30.3% - 88.3%；在 `GET` 下，当应用程序在开启 KPTI 的虚拟机中执行时，对于 1B - 4KiB 的数据大小，Redis 等实际应用可加速 4.4% - 10.8%。

## 鸣谢

感谢牧羊人 Dan Tsafrir 提出的宝贵建议。复旦作者受国家重点研发计划（批准号：2022YFB3102901）和上海市自然科学基金（批准号：23ZR1407100）资助。

## 参考资料


[1] Martín Abadi、Mihai Budiu、Ulfar Erlingsson 和 Jay Ligatti。控制流完整性原理、实现和应用。*ACM Transactions on Informa- tion and System Security (TISSEC)*，13(1):1-40，2009.

[2] ansyun 。 DPDK-Redis. https://github.com/ansyu n/dpdk-redis 。访问日期：2021-05-05。

[3] 法布里斯-贝拉德QEMU，快速、可移植的动态翻译器。In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46.美国加利福尼亚州，2005 年。

[4] Tyler Bletsch、Xuxian Jiang、Vince W Freeh 和 Zhenkai Liang.面向跳转的编程：一类新的代码复用攻击。*第 6 届 ACM 信息、计算机和通信安全研讨会论文集》*，第 30-40 页，2011 年。

[5] 扎克-布朗异步系统调用。*渥太华 Linux 研讨会论文集》（OLS）*，第 81-85 页，2007 年。

[6] Erik Buchanan、Ryan Roemer、Stefan Savage 和 Hovav Shacham。面向返回的编程：无代码注入的漏洞利用。*黑帽*，2008 年 8 月。

[7] JeffCaruso. 万 IOPS 演示。 https://www.networkworld.com/article/2244085/1-million-iops-demonstrated.html。访问日期：2021-12-01。

[8] Stephen Checkoway、Lucas Davi、Alexandra Dmitrienko、Ahmad-Reza Sadeghi、Hovav Shacham 和 Marcel Winandy。面向返回的无返回编程。*第 17 届 ACM 计算机与通信安全会议论文集》*（CCS '10），第 559-572 页，美国纽约州纽约市，2010 年。
计算机协会。

[9] Vitaly Chipounov 和 George Candea.使用 QEMU 将 x86 动态转换为 LLVM。技术报告，EPFL，2010 年。

[10] 阿里巴巴云。通过多线程处理提高 Redis 性能。 https://alibaba-cloud.medium.com/improving-redis-performance-through-multi-thread-processing-ca4d8353523f.访问时间：2020-11-30。

[11] DPDK。数据平面开发工具包。 https://www.dpdk.org/。访问日期：2021-05-01。

[12] Dong Du、Zhichao Hua、Yubin Xia、Binyu Zang 和 Haibo Chen.XPC：为安全和

高效跨进程调用。*第 46 届国际计算机体系结构研讨会论文集》*，第 671-684 页，2019 年。

[13] Ulfar Erlingsson、Martín Abadi、Michael Vrable、Mihai Budiu 和 George C Necula。XFI：系统地址空间的软件防护。*第 7 届操作系统设计与实现研讨会论文集》*，第 75-88 页，2006 年。

[14] Elazar Gershuni、Nadav Amit、Arie Gurfinkel、Nina Narodytska、Jorge A Navas、Noam Rinetzky、Leonid Ryzhyk 和 Mooly Sagiv。对不受信任的 Linux 内核扩展进行简单而精确的静态分析。In *Pro- ceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069-1084, 2019.

[15] GlareR。该项目的代码库。https://github.com/GlareR/UserspaceBypass。访问时间：2022-09-25.

[16] 顾金玉、吴欣悦、李文泰、刘念、米泽宇、夏玉斌、陈海波。用高效的内核内隔离和通信协调微内核的性能和隔离。*2020年USENIX年度技术大会（USENIXATC 20）*，第401-417页、2020.

[17] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien- Min Wang, and Yeh-Ching Chung.HQEMU：多核上的多线程和可重定向动态二进制翻译器。*第十届代码生成与优化国际研讨会论文集》*，第 104-113 页，2012 年。

[18] Amr Hussam Ibrahim、Mohamed Bakr Abdelhalim、Hanadi Hussein 和 Ahmed Fahmy。优化系统软件的 x86-64 指令集分析。*规划视角》*，第 152 页，2011 年。

[19] 安德鲁-杰弗里在 QEMU 中使用 LLVM 编译器基础架构进行优化的异步动态翻译。*阿德莱德大学荣誉论文*，2009 年。

[20] Dae R Jeong、Kyungtae Kim、Basavesh Shivakumar、By- oungyoung Lee 和 Insik Shin。Razzer：通过模糊查找内核竞赛漏洞。In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754-768.IEEE, 2019.

[21] EunYoung Jeong、Shinae Wood、Muhammad Jamshed、Haewon Jeong、Sungghwan Ihm、Dongsu Han 和 Ky- oungSoo Park。mTCP：适用于多核系统的高度可扩展用户级 TCP 栈。*第 11 届 USENIX 网络系统设计与实现研讨会（NSDI 14）*，第 489-502 页，2014 年。

[22] Piyus Kedia 和 Sorav Bansal.内核的快速动态二进制翻译。*第二十四届 ACM 操作系统原理研讨会论文集》*，第 101-115 页，2013 年。

[23] Kernel.dk. 使用 io_uring 实现高效 IO。https://kern el.dk/io_uring.pdf。访问日期：2021-12-01。

[24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim.Nvmedirect：在 NVMe SSD 上针对特定应用进行优化的用户空间 I/O 框架。*第 8 届 USENIX 存储和文件系统热点话题研讨会（HotStorage 16）*，2016 年。

[25] Paul Kocher、Jann Horn、Anders Fogh、Daniel Genkin、Daniel Gruss、Werner Haas、Mike Hamburg、Moritz Lipp、Stefan Mangard、Thomas Prescher、Michael Schwarz 和 Yuval Yarom。幽灵攻击：利用投机执行。In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1-19.IEEE, 2019.

[26] Hsuan-Chi Kuo、Dan Williams、Ricardo Koller 和 Sibin Mohan。穿上单内核服装的 Linux。In *Proceed- ings of the Fifteenth European Conference on Computer Systems*, pages 1-15, 2020.

[27] 德米特里-库兹涅佐夫和亚当-莫里森Privbox：通过沙箱特权执行加快系统调用。*2022 USENIX 年度技术会议（USENIX ATC 22）*，2022 年。

[28] Yoochan Lee, Changwoo Min, and Byoungyoung Lee.ExpRace: Exploiting kernel races through raising inter- rupts.*第 30 届 USENIX 安全研讨会（USENIX Security 21）*，第 2363-2380 页，2021 年。

[29] Moritz Lipp、Michael Schwarz、Daniel Gruss、Thomas Prescher、Werner Haas、Anders Fogh、Jann Horn、Stefan Mangard、Paul Kocher、Daniel Genkin、Yuval Yarom 和 Mike Hamburg。Meltdown：从用户空间读取内核内存。*第 27 届 USENIX 安全研讨会（USENIX Security 18）*，第 973-990 页，2018 年。

[30] 罗志云 Redis 如何处理请求？（转译）。https://www.luozhiyun.com/archives/674.访问日期：2022-09-25。

[31] Anil Madhavapeddy、Richard Mortier、Charalampos Rotsos、David Scott、Balraj Singh、Thomas Gazagnaire、Steven Smith、Steven Hand 和 Jon Crowcroft。Unikernels：云图书馆操作系统。*ACM SIGARCH Computer Architecture News*，41（1）：461-472，2013.

[32] 毛延东、陈皓刚、周东、王曦、Nickolai Zeldovich 和 M Frans Kaashoek。软件

利用 api 完整性和多 principal 模块实现故障隔离。*第二十三届 ACM 操作系统原理研讨会论文集》*，第 115-128 页，2011 年。

[33] Andrea Mayer、Pierpaolo Loreti、Lorenzo Bracciale、Paolo Lungaroni、Stefano Salsano 和 Clarence Filsfils。使用H^2进行性能监控：基于SRv6的混合SDN的混合内核/eBPF数据平面。*计算机网络*，185：107705，2021。

[34] Steven McCanne 和 Van Jacobson.BSD 数据包过滤器：用户级数据包捕获的新架构。见 *USENIX 冬季刊*，第 46 卷，1993 年。

[35] 米泽宇、李定基、杨子涵、王欣然和陈海波。天桥：微内核的快速安全进程间通信。In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1-15, 2019.

[36] Pierre Olivier、Daniel Chiba、Stefan Lankes、Changwoo Min 和 Binoy Ravindran。二进制兼容的 Uniker- nel。*第 15 届 ACM SIGPLAN/SIGOPS 虚拟执行环境国际会议论文集》*，第 59-73 页，2019 年。

[37] M.Prandini 和 M. Ramilli.面向返回的编程。*IEEE Security and Privacy*, 10(6):84-87, 2012.

[38] Mohan Rajagopalan、Saumya K Debray、Matti A Hiltunen 和 Richard D Schlichting。Cassyopia：Com- piler assisted system optimization.在 *HotOS* 中，第 3 卷，第 1-5 页，2003 年。

[39] Redis。Redis Benchmark。https://redis.io/docs/reference/optimization/benchmarks/。访问日期：2022-09-25。

[40] Vasily A Sartakov、Lluís Vilanova 和 Peter Pietzuch。Cubicleos：实用隔离软件组件化的库操作系统。*第 26 届 ACM 国际编程语言和操作系统架构支持大会论文集》*，第 546-558 页，2021 年。

[41] Felix Schuster、Thomas Tendyck、Christopher Liebchen、Lucas Davi、Ahmad-Reza Sadeghi 和 Thorsten Holz。假冒的面向对象编程：论防止 C++ 应用程序中代码重用攻击的难度。In *2015 IEEE Symposium on Security and Privacy*, pages 745-762.IEEE, 2015.

[42] Amol Shukla、Lily Li、Anand Subramanian、Paul AS Ward 和 Tim Brecht。评估用户空间和内核空间网络服务器的性能。*CASCON 第* 4 卷，第 189-201 页，2004 年。

[43] Livio Soares 和 Michael Stumm.FlexSC：无异常系统调用的灵活系统调用调度。*第 9 届 USENIX 操作系统设计与实现会议论文集*，OSDI'10，第 33-46 页，美国，2010 年。USENIX 协会。

[44] 谭刚*基于软件的故障隔离原理与实现技术》*。现在出版社，2017.

[45] 腾讯。F-Stack. https://github.com/F-Stack/f-stack。访问日期：2022-09-25。

[46] 腾讯. F-Stack Redis. https://github.com/F-Stack/f-stack/tree/dev/app/redis-6.2.6。获得时间：2022-09-25。

[47] 内核开发社区。Page table isola- tion (PTI)。https://www.kernel.org/doc/html/latest/x86/pti.html。访问日期：2021-12-01。

[48] Nigel Topham 和 Daniel Jones.使用 JIT 二进制翻译的高速 CPU 仿真。2007 年*修改、基准测试和仿真（MOBS）研讨会。*

[49] Stephan Van Schaik、Alyssa Milburn、Sebastian Öster- lund、Pietro Frigo、Giorgi Maisuradze、Kaveh Razavi、Herbert Bos 和 Cristiano Giuffrida。RIDL：恶意飞行数据加载。In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88-105.IEEE, 2019.

[50] wg. wrk. https://github.com/wg/wrk.Accessed: 2020-12-15.

[51] 杨子烨、詹姆斯-R-哈里斯、本杰明-沃克、丹尼尔-维尔坎普、刘长鹏、张存银、曹刚、乔纳森-斯特恩、维沙尔-维尔马、卢斯-E-保罗。SPDK：构建高性能存储应用的开发工具包。In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154-161.IEEE, 2017.

[52] Bennet Yee、David Sehr、Gregory Dardyk、J Bradley Chen、Robert Muth、Tavis Ormandy、Shiki Okasaka、Neha Narula 和 Nicholas Fullagar。本地客户端：可移植、不受信任的 x86 本地代码沙盒。*2009 年第 30 届 IEEE 安全与隐私研讨会*，第 79-93 页。电气与电子工程师学会，2009 年。

[53] Kai Yu, Chengfei Zhang, and Yunxiang Zhao. 基于 unikernel 的 Web 服务器。In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 280-282.IEEE, 2017.

## 艺术品附录

### 摘要

我们的成果包括 UB 的源代码和用于评估的应用程序。读者可以按照说明修改 Linux 内核以支持 UB，编译 UB 以在其上运行，并在其上评估应用程序。

### 范围

我们评估的所有应用程序的 IOPS 都可以再现。特别是图 5、图 7、图 8、图 9 和图 10。重现 I/O 基准是最方便的情况。因此，建议从图 5 开始。

整个实验可能会很耗时，因此人们可能会减少重复次数以节省时间。

### 内容

zz_lkm 是 UB 的内核部分，负责配置进程和执行 BTC。zz_daemon 位于用户空间，负责与内核模块通信，并调用 zz_disassem 进行实际翻译。

### 托管

源代码托管在 https://github.com/glarer/UserspaceBypass，以及 readme 文件。

### 要求

I/O 基准实验只需要一台服务器机器。由于 Redis、Nginx 和原始套接字实验涉及网络，因此需要另一台客户机连接到服务器。

IOPS 与 CPU 性能密切相关。因此，不同 CPU 重现的 IOPS 值可能不同，但我们总能看到性能的提升。

IOPS 也会受到网络性能的影响。如果使用的网卡功能不够强大，IOPS 可能会因 I/O 大小较大而下降，性能增益也会下降。