**用户空间绕过：加速系统调用密集型应用程序**

**摘要**

内核模式和用户模式之间的上下文切换通常会导致显着的开销，从而降低具有频繁系统调用（或系统调用）的应用程序的速度，例如那些具有高 I/O 需求的应用程序。 Linux 内核页表隔离 (KPTI) 等安全机制进一步放大了开销。为了加速此类应用程序，人们在从 I/O 路径中删除系统调用方面付出了很多努力，主要是通过将驱动程序和应用程序组合在同一空间中或批处理系统调用。尽管如此，此类解决方案要求开发人员重构其应用程序甚至更新硬件，这阻碍了其广泛采用。

在本文中，我们提出了另一种方法，即用户空间旁路（UB），通过透明地将用户空间指令移动到内核中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，并实现完全的二进制兼容性。具体来说，为了避免频繁的系统调用造成的开销，内核识别连续系统调用之间的短用户空间执行路径，并将路径中的指令转换为具有基于软件的故障隔离（SFI）保证的代码块。根据我们的评估，当应用程序以a执行时，I/O微基准可以加速30.3 – 88.3％，Redis每秒GET请求数（RPS）可以提高4.4 – 10.8％，对于1B – 4KiB数据大小。开启 KPTI 的虚拟化设置。当 KPTI 关闭时，性能提升将会降低。

1. **引言**

用户空间应用程序广泛使用系统调用 (syscall) 来访问托管操作系统 (OS) 提供的资源，并广泛用于 I/O 操作。然而，当 Linux 内核页表隔离 (KPTI) [47] 等机制打开时，系统调用可能会产生显着的性能开销 [43]。可以说，系统调用是追求高每秒 I/O 请求 (IOPS) 的应用程序的主要性能瓶颈之一，例如那些请求超过 100 万 IOPS 的应用程序 [7]。

系统调用重构方法。在最近的文献中，主要有两个工作流通过改变 I/O 路径上的系统调用处理方式来实现更高的 IOPS，我们称之为系统调用重构方法： 1）第一个方法流集成驱动程序和数据处理逻辑通过将数据处理逻辑移至内核 [26, 36, 53] 或将负责 I/O 的驱动程序移至用户空间（内核旁路）[21, 51]，从而在同一地址空间中实现相同的功能。这样，处理逻辑可以直接与I/O设备对话，避免用户模式和内核模式之间切换带来的开销[51]。 2）第二个方法流批处理系统调用，并允许用户空间进程对多个 I/O 请求进行排队，并仅通过一个系统调用将它们一起发出[43]。然而，这些解决方案要求开发人员更改他们的代码，这通常是一项艰巨的任务。

我们的方法。在本文中，我们提出了用户空间旁路（简称UB），它减少了与系统调用相关的I/O引入的开销，同时实现了二进制兼容性（即不需要更改或重建应用程序代码）。 UB 的动机是观察到具有高 IOPS 的应用程序在两个连续的系统调用之间不会执行许多指令（请参见第 3.1 节）。因此，我们可以根据预定义的安全要求透明地检测系统调用之间的指令（即将指令转换为经过净化的代码块），并让内核执行这些块而不返回用户空间。这样就可以避免连续系统调用带来的开销。图 1 说明了这一想法。

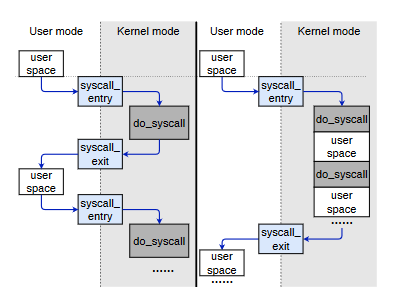


图1 不带 UB 和带 UB 调用系统调用

然而，还有一些挑战需要解决。首先，只有那些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绕过。然而，如果没有开发人员提供明确的信息，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，因此需要仔细选择要优化的系统调用以抵消此类开销。其次，恶意应用程序可能会利用UB窃取内核数据，甚至执行特权指令。此外，有缺陷的应用程序可能会污染内核内存。

因此，UB通过对用户空间代码和数据执行全面的清理来确保内核安全至关重要。最后，为了实现二进制兼容性，提升的应用程序代码应该不知道它们是在内核模式还是用户模式下执行。应保证使用和不使用 UB 的相同执行结果，包括多线程应用程序的内存顺序和原子性。

我们通过采用动态二进制翻译（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，我们通过挂钩条目来分析系统调用，以了解哪些系统调用调用是频繁的（即“热”系统调用）。受即时（JIT）编译[17,22,48]的启发，我们可以在运行时的热系统调用之后获取用户空间指令。如果指令位于同一函数内，则它们将被转换为二进制转换缓存 (BTC)。接下来，我们迭代执行 BTC 并从退出指令扩展 BTC，直到遇到下一个系统调用调用。我们执行指令和地址清理来限制 BTC 的行为，并实现 BTC 上的内核控制流完整性（CFI）和数据完整性。 UB 不会重新排序指令或分割内存访问。这样一来，其他线程就可以与UB优化的线程并发安全地执行。

我们实现了 UB 的原型，并在 I/O 微基准测试和包括 Redis 和 Nginx 在内的实际应用程序中评估其性能增益。在我们的默认设置下（测试的应用程序运行在虚拟机 (VM) 中，并且 Linux KPTI 已打开），I/O 微基准线程可以加速 30.3% 到 88.3%。对于 Redis GET，对于 1B – 4KiB 数据大小，加速比范围为 4.4% 到 10.8%。 Nginx 可以加速 0.4% – 10.9%。 UB 可以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 性能增益的影响。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微基准测试的加速比从88.3%下降到最小 I/O 大小为 41.6%。因此，未来的处理器（预计将消除硬件中的 Meltdown 和 Spectre 漏洞）从 UB 中获得的好处要少得多。当应用程序在物理机中运行时，与 VM 相比，UB 在大多数设置中实现了更高的上限加速比，因为在这种情况下 IOPS 通常更高，这导致可以优化的系统调用更多。我们还在实验研究中将 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，与微基准中的 io\_uring、Redis 宏基准中的 F-Stack 以及原始套接字中的 eBPF 相比，UB 的优势较小。然而，UB 有一个独特的优势，即应用程序开发人员无需更改代码。最后，我们承认 UB 可能会在侧通道、未记录的 x86 指令和内核竞争下引入新的安全风险。据此，我们提出了一些防御思路。

我们的 UB 原型的代码发布于 [15]。我们将本文的贡献总结如下。

• 我们提出用户空间旁路（UB），它直接在内核模式下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，可以透明地将用户空间指令转换为内核安全、经过净化的 BTC。通过这种方法，现有应用程序无需修改即可执行，并享受性能提升。 • 我们实现了一个原型，并针对多个高 IOPS 应用程序对其进行了评估。结果证明了UB的有效性。

1. **背景**

在本节中，我们首先概述系统调用机制及其引入的开销。然后，我们描述了之前为减少此类开销所做的努力。

**2.1系统调用及其成本**

Syscall 提供了用户空间应用程序和内核服务之间的默认接口。可以利用软件中断（例如，已弃用的 int 0x80）和特殊指令（例如，AMD 创建的 syscall/sysret 和 Intel 创建的 sysenter/sysexit）将控制权从用户空间转移到内核空间，反之亦然系统调用。

先前的研究表明，系统调用调用可能会给各种应用程序和场景带来显着的开销[16,35,43]，包括直接成本和间接成本[43]。对于第一种情况，由于要在用户模式和内核模式之间切换，因此必须执行额外的过程来保存寄存器、更改保护域以及处理注册的异常。对于后一种情况，处理器结构的状态包括L1缓存数据和指令缓存翻译后备缓冲区（TLB）等可能会被系统调用污染，并且必须停止CPU的乱序执行（OOE）以保证顺序。因此，系统调用后，每周期用户模式指令数 (IPC) 将会减少。

一种广泛使用的技术称为内核页表隔离（KPTI）[47]，使系统调用变得更慢。为了抵御瞬时执行攻击，例如 Meltdown [29] 和 Spectre v3a [49]，操作系统内核使用两组页表用于用户空间和内核空间。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来。除了 KPTI 之外，虚拟化还可能增加上下文切换开销。例如，VM 内部的 TLB 未命中（间接开销的一部分）的开销可能会更大，因为与物理机内部相比，需要检查更多的页表条目。

下面我们总结了之前研究的观察结果以及我们对具体系统调用开销的测量。 • 根据 Mi 等人的测量，启用 KPTI 的无操作系统调用可能会消耗 431 个 CPU 周期。英特尔 Skylake 和 seL4 [35]。 • 根据我们的实验平台（Intel Skylake 和 Linux）的测量，内核序言和尾声（直接成本）对于无操作系统调用需要 197 条指令（992 个 CPU 周期），这表明系统调用开销问题在研究十年后仍然存在苏亚雷斯等人。 [43]。 • 同样在我们的平台上，pwrite 系统调用可以将以下用户空间指令的IPC 从2.9 降级至0.2（间接成本）。执行20000条指令后IPC慢慢回到2.1。图2显示了IPC随时间变化的趋势。

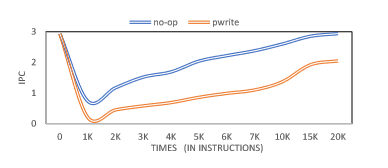


图2 在我们的平台（Intel Skylake 和 Linux）上测量的无操作系统调用和 pwrite 系统调用之后的 IPC

**2.2系统调用的性能优化**

研究社区正在积极致力于减少系统调用带来的开销。下面我们描述了相关工作，并与我们的方法进行了比较（也在表 1 中进行了总结）。

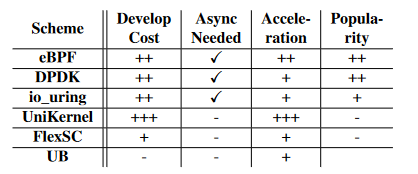


表1优化系统调用方案的比较

异步系统调用。系统调用引入了同步执行模型，因为系统调用完成后会恢复用户模式执行。 Brown 提出非阻塞 Linux 系统调用 [5]可以异步并行完成到用户空间执行流程。但是，这种方法并没有将系统调用调用与其执行完全解耦。到目前为止，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。由于局部性是一个主要的性能因素，因此还研究了批量执行系统调用。拉贾戈帕兰等人。建议将连续的系统调用分组为一个（系统调用的结果直接馈送到下一个）[38]。在两个系统调用之间不发生计算的假设下，这种方法是有效的。苏亚雷斯等人。建议对多个协同例程进行批处理系统调用，并要求开发人员将线程模型更改为 M-on-N（“M 个用户模式线程在 N 个内核可见线程上执行，其中 M » N”）[38]。因此，它仅在任务可以拆分为多个线程时才有效。现代内核提供本机队列，即 io\_uring [23]，来批量处理来自用户空间进程的 I/O 请求并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来让内核处理队列。

统一内核。为了减轻上下文切换的开销，Unikernel 解决方案在内核空间而不是用户空间中运行应用程序代码。示例包括可加载内核模块 (LKM) [42] 和库操作系统 [31, 40]。内核沙箱。为了减少系统调用引起的上下文切换的发生，内核沙箱允许应用程序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中。当内核达到这些点时，它将使用虚拟机来执行附加的代码。然而，eBPF对代码做了很多限制，内核在执行前会验证是否满足所有要求，期间合法的代码可能会因为验证误报而被拒绝。最近，德米特里等人。建议使用内核沙箱完全在内核中执行应用程序[27]，其中也可以减轻上下文切换开销。

内核绕过。观察到内核并不总是需要参与数据包处理等 I/O 任务，一些研究人员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(DPDK)。它接管用户空间中的 I/O 设备。具体来说，I/O 请求通过共享环形缓冲区而不是系统调用提交给设备。缓冲区在用户空间中维护，不涉及 I/O 的内核活动。

我们发现现有的方法都需要显着的开发工作。为了使用系统调用重构原语，必须遵循不同的编码范例。大多数内核旁路和系统调用批处理解决方案（例如 DPDK、RDMA、io\_uring）需要应用程序代码与队列对异步交互。尽管如此，开发人员仍然更喜欢以同步方式编写程序逻辑。重构遗留代码也是劳动密集型的。作为示例，我们将支持 DPDK 的非官方 Redis [2] 与其官方版本（版本 3.0.5）进行比较。我们发现前者包含 9,984 行额外代码（LoC）来支持 DPDK，占正式版本 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，不幸的是很难调试并且容易出现内存损坏等错误（没有内存隔离）。除了更改应用程序代码之外，内核绕过解决方案可能还需要特殊的用户空间驱动程序[24]。

1. **设计概述**

为了解决上述问题，我们提出了用户空间旁路（UB），这是一种用于系统调用优化的新原语。布法罗大学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动工作。与系统调用重构方法不同，系统调用重构方法需要开发人员更改遗留代码或调整为异步编程，UB 在执行时优化系统调用，同时不会影响应用程序的功能。 DG2：尽量减少对系统架构的更改。系统调用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 收集模型。 DG3：与系统调用重构方法相当的性能。 UB 旨在降低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程序**

我们专注于优化高 IOPS 的应用程序，例如 Redis 和 Nginx，这些应用程序也是系统调用密集型的。通过分析它们的代码和运行时行为，我们确定了以下两个指导 UB 设计的见解。

I/O 线程中的轻量级用户空间指令。我们发现，对于所检查的应用程序来说，I/O 事件之间的计算工作负载通常是轻量级的。此外，两个连续系统调用之间的指令数量通常很小。一种解释是，此类应用程序遵循流行的 I/O 模型，该模型将不同线程中的 I/O 密集型工作负载与 CPU 密集型工作负载分开。例如，Redis 服务器有一个主线程，它将接受的套接字分派给 I/O 线程 [10]，I/O 线程从/到内核进行 I/O，并让主线程完成 CPU 密集型计算。通过这样的设计，I/O 事件之间的指令只需处理缓冲区移动。我们还分析了 Redis 调用的系统调用（总共 3M），发现其中一半在下一个系统调用之前跟随了不到 400 个用户空间指令（当 IPC 为 2 时大约 200 个周期），这比执行系统调用本身更快（例如，431 个周期 [35]，如第 2.1 节所述）。

扩大了直接和间接成本。 2.1 节概述了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程序中可能会放大。如图 1 所示，进入和退出的频率随着系统调用调用的频率而线性上升。 TLB 未命中、OOE 停顿和缓存未命中造成的间接成本也是不可忽略的，特别是当系统调用处理较轻的任务时（无操作系统调用的 IPC 降至 0.74，pwrite 的 IPC 降至 0.21，如图 2 所示）。

**3.2 UB模型**

基于上述考虑，我们设计UB的动机是，它可以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 说明了我们的想法。尽管这个想法在高层看来很简单，但要使 UB 能够用于现实世界的成熟应用程序，还需要解决一些挑战。

• 与内核代码相比，应用程序代码的可信度较低。因此，在移至内核后，应进行必要的隔离以限制其能力。然而，识别不受信任的区域并使用正确的策略对其进行管理并非易事。 • 鉴于隔离会产生额外成本，转换用户空间指令的每个块并不总是有益的。但是，何时执行转换以及如何减少其开销尚不清楚。布法罗大学通过三个关键组成部分应对这些挑战。 1）“热”系统调用标识符，用于监视目标应用程序的执行，分析调用的系统调用，并确定何时需要提升用户空间指令； 2) 即时 (JIT) 翻译器，将用户空间指令转换为采用隔离策略的二进制翻译缓存 (BTC)； 3) 执行翻译后代码的内核 BTC 运行时。图 3 概述了 UB 的设计。

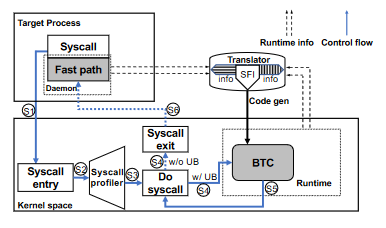


图3 UB 框架概述。每次线程调用系统调用（S1）时，热系统调用标识符都会钩住它（S2）并将其分派给 do\_syscall（S3），之后如果系统调用不热（S6），内核可能会返回到用户模式，或者将其发送到 BTC UB (S5) 的运行时

热系统调用标识符。该模块在内核模式下运行并挂钩每个系统调用。通过分析运行时统计数据，它可以识别哪些系统调用指令是热门指令，即那些很可能很快被另一个系统调用跟随的指令。两个连续热系统调用之间的用户空间指令将在下次应用程序运行时提升到内核并加速。为了避免由于运行时监控而引入大量开销，该模块间歇性运行。比特币翻译器。 BTC 翻译器将前面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（例如，间接控制流传输）转换为安全指令（例如，直接跳转），并进行仪器检查以限制内存访问和控制传输行为。翻译器在单独的、独立的用户空间进程中运行，以避免将其代码引入内核。翻译不会阻塞应用程序的执行，下次访问相同的代码路径时会执行翻译后的代码。除了优化一对热系统调用之间的用户空间指令之外，我们还考虑了一系列热系统调用的加速。我们将封闭的用户空间代码称为快速路径。 UB 的目标是链接此类用户空间代码并共同加速它们。通过观察跳跃目标逐步发现快速路径。详细信息将在第 5 节中讨论。

1. **热系统调用识别器**

用户空间绕过的标准。当用户空间指令区域的性能增益超过 BTC 翻译器的翻译和检测开销时，应该提升该区域。我们测量不同用户空间路径长度（即指令数量）的性能增益，并考虑具有短路径的区域。主要原因是，对于较长的路径，仪器成本会迅速增加，因为必须监视更多的指令。我们将 1,000 条指令（称为 Tpath）视为短路径长度 2 的阈值。通过实证研究，我们观察到此路径长度有明显的性能增益（超过 20%）（参见第 6.2 节）。模块设计。该模块旨在发现包含短用户空间路径的热系统调用。我们借助在线分析来实现无缝分析。具体来说，该模块挂钩系统调用条目并计算两个连续系统调用之间的指令数。当指令数小于 Tpath 时，这两个系统调用被归类为热系统调用的候选者。下面我们来描述一下详细步骤。

• 系统调用采样。监视每个系统调用调用会给应用程序执行带来很高的性能损失。因此，我们仅在线程频繁发出系统调用（例如 I/O 线程）时对系统调用进行采样并进行后续分析。根据我们对系统调用密集型应用程序（例如 Redis 和 Nginx）的测量，每秒至少发出 100K 系统调用（称为 Tsys）（每分钟 6M），并且我们选择分析不到 10% 的 Tsys 系统调用（最多每分钟 500K 系统调用）。因此，大多数系统调用不会被采样，也不会被干扰。

• 粗粒度分析。为了进一步减少分析开销，我们检查受监控的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少于 50K（Tsys 的一半），则模块将不会进行下一次细粒度的系统调用分析。在此情况下，IOPS 较低的线程将被跳过。

• 细粒度分析。对于频繁调用syscall的线程，该模块进一步分析哪些syscall指令被频繁调用。频繁出现的问题值得绕过用户空间，因为可以获得更多性能提升。我们监控每轮的 15K 系统调用（Tsys 的 15%），并维护一个表，记录每个调用的系统调用指令、其位置寄存器（RIP）以及 4 微秒内调用下一个系统调用的次数的计数器（大约为执行 Tpath 指令的时间）。当计数器大于 900（分析的 15K 系统调用的 6%）时，我们认为系统调用频繁。这些系统调用及其包含的用户空间指令将在下一阶段由 BTC 翻译器处理。

人们可能想知道该模块的性能是否对参数选择敏感。为了测试灵敏度，我们检查实验使用的两个应用程序 Redis 和 Nginx 的热系统调用是否可以在三台不同的机器上正确发现：一台配备 Core i5 10500 的 PC（2021 年），两台配备 Xeon 8175 和 8260 的服务器（2017 年和 2019 年）。所有热系统调用都可以被正确识别，这表明在大多数情况下可以跳过参数调整。

1. **BTC 运行时间和翻译器**

在本节中，我们将描述 BTC 翻译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 翻译器遵循动态二进制翻译（DBT）[19,22,48]的程序。一般来说，给定一个由二进制基本块组成并触发事件（例如，我们案例中的热系统调用）的路径，DBT 会对其进行反解，使用 SFI 规则手册对其进行翻译，并将其编译为 BTC 以供将来执行。由于 SFI，翻译后的代码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 中的检测指令可以访问这些变量，以进行策略执行和上下文切换。局部变量包括：1）保存的内核上下文，即被调用者保存的寄存器，2）保留寄存器的值，以及3）用于构建快速路径的间接跳转目的地信息。

在执行 BTC 之前，运行时会准备用户空间的返回状态，即通过恢复保存在系统调用条目上的用户空间上下文（例如，x86\_64 的 pt\_regs）。一个区块完成后，运行时会处理 BTC 的返回状态并采取进一步的操作。当跳转目标丢失时，例如，当遇到新路径时，BTC 的执行可能会在中间退出运行时。在这种情况下，运行时会记录本次跳转的信息，并立即返回到用户空间，即跳转目标。我们使 BTC 运行时可以访问用户空间内存，因此内存上的所有更改都会保留。对寄存器所做的更改将更新到用户空间上下文（即 x86\_64 的 pt\_regs），当内核返回到用户空间时，这些更改将被写入寄存器。因此，BTC 所做的用户空间状态更改也被保留并对其他线程可见，这确保了 UB 下的应用程序逻辑不被更改。

当遇到系统调用指令时，BTC 的执行也可能会退出。在这种情况下，两个连续系统调用之间的快速路径已在内核中完全执行，这表明用户空间绕过成功。 BTC 运行时通过根据系统调用表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。 do\_syscall 返回后，BTC 运行时检查下一个系统调用指令是否再次成为热门指令。如果答案是肯定的，运行时会尝试进行另一次用户空间绕过。这样do\_syscalls和用户空间bypass就可以链起来，类似于DBT的直接分支链。在理想情况下，整个线程可以在内核中执行。

快速路径发现。 UB 的性能很大程度上取决于快速路径的识别精度，我们利用增量式、JIT 式的方法来实现高精度。给定一个入口地址，即紧邻热系统调用的指令，BTC 翻译器首先通过迭代地从入口地址反汇编目标线程的代码段来发现快速路径的一部分。翻译器在每次迭代中都会跳过潜在不可达的路径。具体来说，翻译器仅遵循直接跳转并在调用指令处停止，这迫使翻译器在一次迭代中仅处理函数内的代码，并将其视为快速路径。当稍后确实进行间接跳转或调用时，目标信息将由 BTC 运行时收集并发送给翻译器，以在替换跳转指令后扩展快速路径（参见第 5.2.1 节）。这种方法类似于 QEMU [9] 采用的方法，但我们不会将二进制文件提升为其中间表示形式。

**5.2 BTC翻译器**

下面我们描述如何将安全策略融入到用户空间代码中。我们遵循SFI原则在内核上提供数据访问策略和控制流策略[44]，并且实现继承和扩展自Nacl[52]，Nacl[52]将浏览器中不受信任的x86本机代码沙箱化。值得注意的是，Nacl 假设源代码可用，因此可以在静态编译下强制执行 SFI 规则。相反，UB 对二进制文件执行 DBT。因此，SFI规则必须进行调整和扩展。

威胁模型。我们假设用户空间代码是不可信的，它可能包含任意代码和数据，副作用包括对内核内存的无中介访问、特权函数等。UB的目标是确保用户空间代码无法获得更多特权（并造成更多危害） ）在提升到内核后，即保护内核的控制流完整性。值得注意的是，这个目标不同于保证用户空间应用程序上的控制流完整性[1]（第 5.3 节中详细阐述）。我们在设计 UB 时采取保守的方法，避免在无法立即确定后果时（例如，翻译过程中跳转目标未知）提升快速路径。我们专注于 x86\_64 平台，但所提出的技术可以轻松推广到其他平台。下面我们描述与跳转、寄存器、指令和内存访问相关的实现，以确保该威胁模型下的安全。

**5.2.1 净化跳跃**

Nacl 的内部沙箱检查用调用和跳转表示的显式控制流，并不允许间接跳转和调用指令上的内存取消引用。跳跃的目标被限制在沙箱内。相比之下，整个内核内存空间对 UB 下提升的用户空间代码开放。因此，我们采取不同的方法来净化跳跃。

直接跳跃。为了防止 BTC 中的代码跳转到任意地址，只有在已知跳转目标的情况下才会进行转换。换句话说，仅处理目标已知的直接跳转。地址清理在第 5.2.3 节中描述。

间接跳跃。然而，用户空间快速路径可能包含间接跳转，我们阻止 BTC 处理此类路径，直到目标已知为止。特别是，当第一次遇到相关代码时，翻译器会插入检查，将目标与目标地址表（类似于跳转表[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。当在执行 BTC 块期间触发此类退出时，BTC 运行时将跳转指令地址（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。

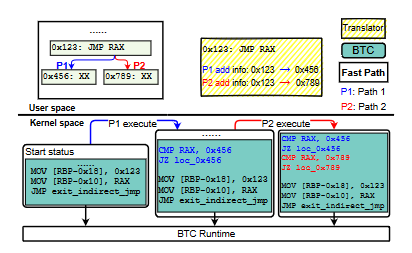


图4 跳跃净化条件下的翻译示例

我们在图 4 中展示了一个示例。间接跳转（跳转到 RAX，位于 0x123）最初被转换为写下跳转目标（将 RAX 保存到堆栈）并退出到 BTC 运行时（跳转到 exit\_indirect\_jmp）。当路径P1第一次执行时，BTC运行时学习目标0x456，并将信息发送到翻译器，翻译器通过添加目标表条目来更新BTC。之后路径P1就被添加到BTC中，下次就不会触发exit\_indirect\_jmp了。如果稍后到达P2，则可以获知另一个目的地0x789并且BTC将被更新，因此快速路径被进一步延伸。

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目标。由此产生的BTC最终可以覆盖整个快速路径。插入到 BTC 中的检查可以高效执行，因为：1）根据我们对系统调用密集型应用程序的实证分析和之前的研究[18]，间接控制流传输指令并不经常出现； 2)允许CPU在乱序执行下推测性跳转到目的地，而不需要等待目的地检查。

**5.2.2 寄存器重新映射**

为了保护内核寄存器和堆栈，BTC 转换器不允许 BTC 代码访问堆栈寄存器（即 RSP、RBP 和 RIP）。此外，一些寄存器是为 BTC 运行时保留的，也不能被 BTC 代码访问。因此我们开发了这个模块来管理寄存器。具体来说，BTC 翻译器使用 BTC 中的 M 个保留寄存器来服务对 N 个寄存器的潜在访问（N = M + 3，其中 3 个用于堆栈寄存器）。由于M < N，翻译器需要调度寄存器。 N 个寄存器的值存储在局部变量中，翻译器从 M 个保留寄存器中选择一个来暂时充当具有重命名的特殊寄存器。翻译器还插入代码以将 N 个寄存器与堆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。登记预约。翻译器保留 R12-R15（M = 4）供 BTC 运行时使用，因为它们在常见用户空间应用程序中使用频率最低（使用频率低于 1% [18]）。当它们出现在快速路径中时，就会发生重命名。我们还优化了常用特殊寄存器（即 RSP）的重命名机制，让翻译器修复保留寄存器以保存其值。这样做可以减少昂贵的寄存器同步的发生。

**5.2.3 指令清理**

特权指令（例如sysret）不允许出现在BTC中，以避免利用UB的恶意代码进行特权升级。在翻译过程中，如果包含任何特权指令，翻译器会避免提升到内核的快速路径。由于寄存器重新映射，一些指令必须重写。对于像 PUSH/POP 这样的堆栈操作指令，翻译器会用多条指令代替它们。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（即作用堆栈指针）寻址的内存中的 MOV 到弹出目标，然后用新的堆栈指针值（即加 8）更新保留寄存器。

**5.2.4 内存访问清理**

为了防止对内核内存的未经授权的访问，翻译器清理了所有内存访问指令。对于每条这样的指令，翻译器都会在指令之前插入地址检查指令，这样只允许访问用户空间地址，即地址以0开头。与SFI[44]的地址掩码类似，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%）。请注意，添加的检查不会阻止 BTC 访问未映射的内存区域并触发页面错误，我们按照下面描述的过程进行处理。

页面错误处理。我们修改页面错误处理程序来监视页面错误事件。对于次要错误和主要页面错误，页面错误处理程序在内核模式和用户空间模式下的行为相同。因此，由用户空间应用程序引起的故障的解决方式与没有 UB 时相同。当发生无效页面错误（即非法访问某些内存区域）时，BTC代码的执行将被中止。

Nacl 还借助 x86 CPU 提供的分段来隔离扩展程序和主机浏览器之间的内存空间。因此，扩展指令只能访问段内的内存，并且不允许修改段状态的指令。不过，虽然x86\_64仍然提供了分段，但它只是在地址上添加了段偏移量，但没有检查段边界，无法直接用于内存隔离。

**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 在内核上实现 SFI 策略[44]。

SP1：BTC 的内核控制流完整性 (CFI)。这个属性是有保证的，因为当 BTC 运行时将控制流移交给用户空间时，执行只会通过退出点终止。更重要的是，当运行时执行BTC时，线程无法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块的标签。通过替换目的地，间接控制流传输全部转换为直接传输。因此，BTC 可以防止恶意代码在提升后劫持内核控制流。我们想指出的是，UB 并未声称针对控制流劫持添加额外的保护，例如 ROP、JOP、COOP [4,6,8,37,41]，并且它们仍然可能发生在用户空间中。尽管攻击者可以在通过目标检查时构造小工具，但从 BTC 跳转到内核代码段是绝对不允许的，因为它可以被翻译器检测到并中止。

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），我们将 BTC 运行时设计为与调用约定兼容，调用者（内核）上下文在跳转到 BTC 之前保存在堆栈上，在返回内核指令之前恢复。上下文切换是轻量级的，因为它不会导致特权转移。对于内核内存，访问清理可确保清理后的指令无法访问任何内核内存，因此内核堆栈不会受到污染。尽管运行时局部变量必须可由 BTC 中的指令访问，但它们不能被恶意程序利用来触及内核堆栈。只有有意插入的指令才能触及堆栈基址指针引用的局部变量，该指针存储运行时信息，例如换出的寄存器（请参见第 5.2.2 节）。因为内核 CFI 是有保证的，所以执行永远不会跳转到这些指令。

SP3：BTC 中没有特权指令。第 5.2.3 节对此进行了解释。

SP4：死循环中断。我们还考虑针对系统资源可用性的攻击和错误。例如，用户空间应用程序可能由于错误或故意而陷入死循环。作为对策，翻译器在 BTC 运行时维护一个计数器，以跟踪已执行的指令数量。一旦计数器超过阈值，执行流程就可以退出到运行时，然后返回到用户空间，这可以避免内核被 BTC 代码阻塞。

**5.4 线程安全**

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之外，UB 无法控制其他线程。必须保留内存顺序和原子性以避免数据竞争。幸运的是，线程安全是由译者自动保证的，我们在下面解释一下。

记忆顺序。为了保持内存顺序，翻译器将所有用户空间内存视为易失性，并且仅在用户空间指令之间插入指令而不优化块（例如，重新排序指令或在寄存器中缓存内存修改）。然而，CPU 仍然可以根据其内存模型重新排序内存加载和存储。用户空间应用程序放置的原始内存栅栏全部被继承，并且翻译器不会插入额外的栅栏。

原子性。当使用多条指令模拟一条用户空间指令时，翻译器会采取特殊措施来保证原子性。当翻译一条指令时，翻译器更喜欢使用与原始指令具有相同操作码的指令。因此，原始指令的原子性被自动保留。例如，带有锁定前缀的指令被转换为仍然带有锁定的指令（例如，LOCK MOV）。如果仿真需要多条指令，则必须在单条指令中完成内存加载或存储。例如翻译PUSH RIP时，下一条指令的偏移地址必须被移到用户空间栈顶。从译者的角度来看，当 BTC 运行时到达该指令时，RIP 的值是静态已知的，并成为立即数。但是，x86\_64 没有直接将 64 位中间值移至内存的指令。结果，翻译器生成的指令首先将立即值移动到 64 位保留寄存器，然后将 64 位寄存器移动到用户空间堆栈的顶部。

1. **评估**

我们为Linux内核5.4.44实现了UB的原型。 BTC运行时被实现为具有416行C代码的内核模块，它挂钩系统调用尾声以进行系统调用识别并管理BTC运行时。翻译器在用户空间用 786 行 Python 代码实现（除了依赖的 Python 反汇编器 miasm 和 gcc 汇编器 as），通过 sys 文件与 BTC 运行时内核模块进行通信。通过在系统调用条目中仅添加 6 行代码来修改内核，以允许模块挂钩系统调用。

我们在 I/O 微基准测试和宏基准测试的两个实际应用程序（Redis 和 Nginx）中评估了我们的原型。还与DPDK、io\_uring、eBPF等相关技术进行了比较。为了评估这些应用程序，我们设置了虚拟化环境和裸机环境。裸机环境由一台客户端计算机和一台服务器计算机组成，它们在 40G 以太网 LAN 内连接。虚拟化环境运行在服务器上，并启用网卡直通。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。对于其他场景，我们在客户端计算机中运行客户端应用程序，在服务器计算机中运行服务器应用程序，因此流量会通过物理网络。为了显示虚拟化和 KPTI 的影响（如第 2.1 节所述），我们在四种设置中运行每个服务器应用程序：KPTI 开/关 × VM/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 或每秒请求数 (RPS) 值。对于6.1节到6.4节演示的结果，我们重点关注KPTI打开的VM设置，并简要描述在其他设置下结果如何变化。在表2中，我们列出了不同设置之间的加速比。

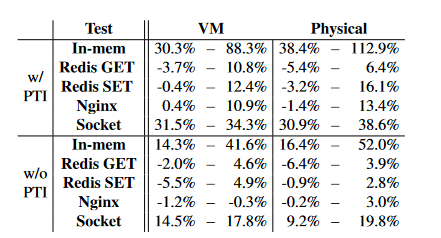


表2 不同设置的加速比范围。 “In-mem”表示内存中文件访问基准

**6.1 I/O 微基准测试**

我们首先考虑加速一个通过阻塞系统调用纯粹执行文件 I/O 请求的线程作为微基准，这近似于 UB 的最佳情况。该线程运行一个紧密循环，通过 READ 系统调用 839 M 次顺序将文件从内核读取到用户空间缓冲区。现实世界的应用程序可能会表现出不同的模式，例如在连续 I/O 请求之间执行更多指令，从而降低 UB 的加速比。为了进行比较，我们使用 io\_uring (liburing-2.2) 来执行相同的任务（即紧密循环 READ 系统调用）并比较 IOPS。

内存中文件访问。我们在 ramfs 中创建一个大文件以避免可能的磁盘瓶颈，以便更准确地评估 UB 如何加速系统调用。不可否认，这种设置使得微基准不太现实。我们逐渐增加每次读取的缓冲区大小，并评估不同缓冲区大小下UB的加速比。

图 5 显示了结果。对于启用 KPTI 的虚拟化环境，当 I/O 大小较小（64B）时，UB 将基于系统调用的 I/O 加速了 88.3%±0.75%4。对于较大的 I/O 大小，UB 和基线的 IOPS 都会下降，并且对于 4KiB I/O 大小，加速比下降至 30.3%±0.96%，因为调用的系统调用较少。关闭KPTI会增加IOPS，但UB的加速比下降到14.3%±1.83% – 41.6%±1.73%，因为系统调用开销减少了。物理机上的加速更高，特别是当 I/O 大小较小时（例如，KPTI 打开时 I/O 大小为 64B 时为 112.9%±1.78%），因为物理机上的 IOPS 更高，并且 UB 节省更多上下文切换开销。

对于io\_uring，我们首先检查从1到1024的不同队列深度（即可以批量处理多少个请求），发现深度达到128后IOPS稳定，如图6所示。因此，我们将深度设置为128与 UB 的比较。根据图 5，事实证明，对于大多数缓冲区大小，io\_uring 会产生更多的 IOPS。当运行在物理机上时，对于小尺寸（64 字节），UB 比 io\_uring 产生更高的 IOPS，尽管预计 io\_uring 应始终优于 UB。我们还没有找到很好的解释，但我们注意到，当 Linux 内核从 5.4.44（我们的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 在较新的 Linux 上可能会始终优于 UB。

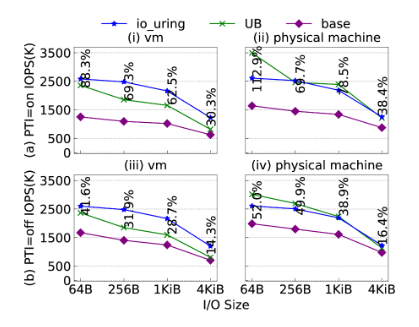


图5 针对不同缓冲区大小的 READ 系统调用、io\_uring 和 UB 系统调用的 IOPS。百分比数字是UB相对于基线的加速比。图 7、8、9 和 10 遵循这种风格。

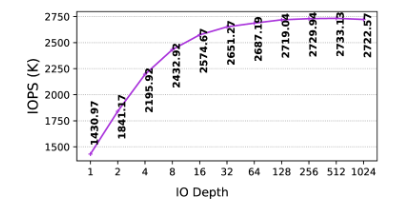


图6 不同io\_uring深度的IOPS

NVMe 上的文件访问。我们还测试以 1KiB 块大小读取 NVMe 磁盘中的文件，并在表 3 中显示比较（“w/o sum”）。我们只考虑物理机设置，因为当VM访问虚拟NVMe磁盘中的文件时，该文件会自动先验地缓存到内存中，其行为类似于内存中文件访问。 IOPS 可从 779±5K 增加到 852±3K，产生 9.4%±0.3% 的加速（KPTI 开启）。当KPTI关闭时，基线增加到810±22K，而UB小幅增加到858±10K，使得加速比变小。

我们还考虑了 I/O 线程进行轻量级计算的情况，例如解析数据包。当连续 I/O 请求之间的计算存在依赖关系时，请求无法批量处理。具体来说，我们设置 I/O 线程在从内核检索缓冲区后，将其视为 64 位整数数组来计算缓冲区的总和。

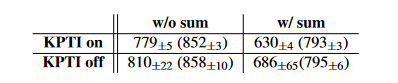


表3 在物理机上读取 NVMe 磁盘（1KiB 大小）上的文件（w/o sum）的 KIOPS，以及与整数求和一起读取（w/ sum）。 UB加速数显示在括号中

如表 3（“w/ sum”）所示，即使是轻量级计算也可能会减少大量 IOPS。基线IOPS下降了149K，而UB打开时只下降了59K，因为用户空间的这种轻量级计算可以完全移植到内核中执行，因此它们的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 作为一个宏观基准来测试 UB 如何处理实际工作负载。我们使用内置的 Redisbenchmark 工具 [39] 来评估 Redis 6.2.6 以生成工作负载。我们使用默认配置运行 Redis 服务器，并使用 2 个线程启动 Redisbenchmark。连接数保持默认值50。每轮客户端发出1M个请求。

默认情况下，Redis 在主线程中完成大部分工作，主线程不仅负责 I/O，还负责哈希等计算任务。对于正常的工作流程（也在[30]中进行了描述），主线程调用 EPOLL 来获取可读套接字列表。对于每个可读套接字，线程读取套接字，然后处理请求。因此，READ 后的用户空间路径很长（从 3k 到 20k），因为计算任务在那里发生。最后，Redis WRITE 对相应的套接字一一进行响应，中间有少量指令（大约 300 条）。

结果。图 7 显示了大小范围为 1B 到 16KiB 的 GET 和 SET 数据的带和不带 UB 的 RPS。在开启 KPTI 的 VM 中测试时，对于 GET，当数据大小小于或等于 4KiB 时，加速比范围为 4.4%±1.52% 至 10.8%±2.69%。当大小增加到 16KiB 时，该比率下降到 -3.7%±0.51%。关闭 KPTI 会将加速比降至 -2.0%±1.32% 和 4.6%±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。在物理机上执行会观察到不同的范围：KPTI 打开时为 -5.4%±1.17% 至 6.4%±2.01%，KPTI 关闭时为 -6.4%±3.02% 至 3.9%±1.67%。值得注意的是，Redis的RPS要小得多而我们的 I/O 微基准测试中的费用要高，因此系统调用的费用不是主要因素。结果，加速比要小得多。

对于SET，在KPTI开启的VM中，加速比范围为-0.4%±2.19%至12.4%±3.96%。当 KPTI 关闭并在物理机中运行时，也会观察到类似的趋势。值得注意的是，对于 SET 和 GET，Redis RPS 显着下降超过 1KiB 数据大小，Redis 基准测试的官方文档中也报告了类似的观察结果 [39]。

令人惊讶的是，我们发现虚拟机上的 RPS 通常高于物理机，尽管虚拟设置应该产生较低的 RPS。我们没有很好的解释为什么 Redis 会出现相反的情况。

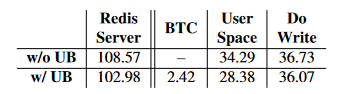


表4 Redis各部分花费的时间（VM+KPTI、SET）

分析性能增益。我们使用 RDTSCP 指令让 BTC 运行时分析 BTC 执行和用户空间执行。我们在有和没有 UB 的情况下运行 20M Redis SET 事务（大约 100 秒）。结果如表4所示。我们可以看到，通过提升 BTC 的快速路径，可以节省 5.91 秒的用户空间时间，而 BTC 仅花费 2.42 秒。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”列），通过调用较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。当不需要强大的内核内存安全性时，例如，当二进制文件被正式验证时，用户可以选择通过删除插入的检查内存边界的指令（即 SHL 和 SHR）来追求更高的性能增益。我们评估如果我们要求译者不插入这样的指令，我们能获得多少RPS增益。结果表明，RPS 只能提高 0.4%。

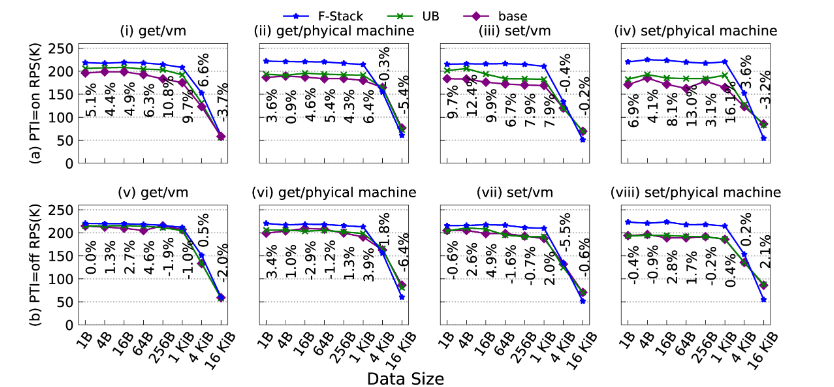


表4 Redis GET和SET在不同数据大小下的RPS

与DPDK的比较。我们比较了 UB 在 Redis 上和在 DPDK 上的加速比，因为有开源实现可以为 Redis 提供支持，例如 Redis-DPDK [2] 和 F-Stack Redis [45]。我们选择 F-Stack 是因为 Redis-DPDK 自 2017 年起已停止维护，并且无法在最新的 CPU 上运行。 F-stack 支持最新的 Redis 6.2.6 [46] 以及最新的 DPDK 20.11。比较结果也如图7所示。

事实证明，F-Stack 始终为小尺寸（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 基线更差。一种可能的解释是 F-Stack 没有从我们的多核设置中受益。当我们测量CPU使用率时，F-Stack始终为100%，但UB和基线可以达到124%，这意味着使用了多个核心。因此，当我们将核心数量限制为 1 时，F-Stack 的性能可能会始终优于 UB。

**6.3 NGINX**

除了Redis之外，我们还使用Nginx（版本1.20.0）作为另一个宏观基准，这是一种流行的高RPS静态Web服务器。

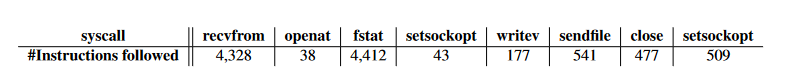


表5 Nginx 每个系统调用之后的指令数。那些后面有少于 1,000 条指令的指令很热门。两个setsockopt 调用是不同的

表 5 显示了每个系统调用所遵循的路径中的指令数。接下来是不到 1,000 条指令可以算是热指令。因此，8 个中的 6 个可以加速。我们在客户端机器上运行 HTTP 基准测试工具 wrk [50]（版本 4.1.0，具有 8 个线程和 1024 个连接），向 Nginx 服务器发出 12 秒的请求，以检查 Nginx 可以处理多少 RPS。

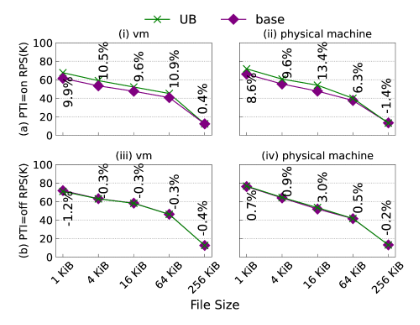


图8 Nginx 针对不同文件大小（字节）的 RPS。

结果。我们逐渐增加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 对于 1KB 到 64KB 文件可以加速 9.6%±1.81% 到 10.9%±0.22%，但对于 256KB 文件该比例下降到 0.4%±0.86%。对于物理机，1KB至64KB文件的加速比范围为6.3%±0.17%至13.4%±3.32%，但256KB文件则下降至-1.4%±0.28%。这些结果表明大文件的瓶颈从系统调用转移到 I/O。当关闭 KPTI 时，UB 不会产生明显的加速。

多个工作线程。我们评估多线程如何影响加速比。我们逐渐增加Nginx的工作线程数量并评估4KB文件大小的情况。图 9 显示了 RPS。可以看到，随着工作线程数量的增加，KPTI开启时加速比明显下降（虚拟机从8.6%±0.22%下降到7.1%±0.17%，物理机从4.7%±0.15%下降到2.0%±0.26%） ，因为工作线程从 2 个增加到 8 个。当工作线程越多时，用于线程同步的周期就越多，因此每个线程可以处理的请求就越少，从而减少了 UB 节省的系统调用开销。

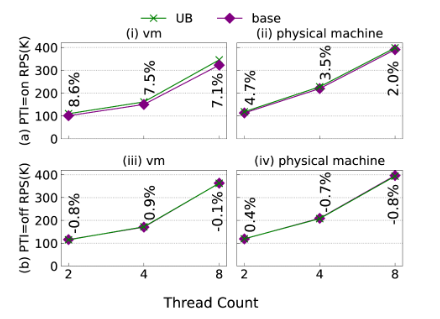


图9 Nginx 针对不同线程数的 RPS

**6.4 原始套接字与 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。我们表明，在 UB 的帮助下，开发人员可以使用原始套接字完全在用户空间中简单地编写处理逻辑，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，将 UDP 数据包发送到服务器，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，这会使服务器饱和。处理任务包括对数据包的数量进行计数以及通过将数据包视为整数数组来对数据包进行求和。

结果。图 10 显示了 3 种数据包大小（128B、512B 和 1472B）的结果。对于启用 KPTI 的 VM，对于小数据包，eBPF 的性能优于原始套接字高达 368.4%±8.92%。对于 MTU 大小的数据包（即 1472B），eBPF 仍然多出 236.7%±4.15% 的 PPS。 UB 将 raw socket 加速了 31.5%±0.25% 34.3%±0.72%，比 eBPF 小很多。对于不同的数据包大小，原始套接字的 PPS 相似。然而，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 绕过了协议栈处理，其瓶颈可能是数据移动，其时间消耗与数据包大小有关。当 KPTI 关闭时，对于各种数据包大小，UB 的加速比下降至 14.5%±0.45% – 17.8%±0.44%。在物理机上，UB的加速比具有更大的范围（KPTI 打开时为 30.9%±0.87% – 38.6%±0.56%，KPTI 关闭时为 9.2%±0.14% – 19.8%±0.31%）。

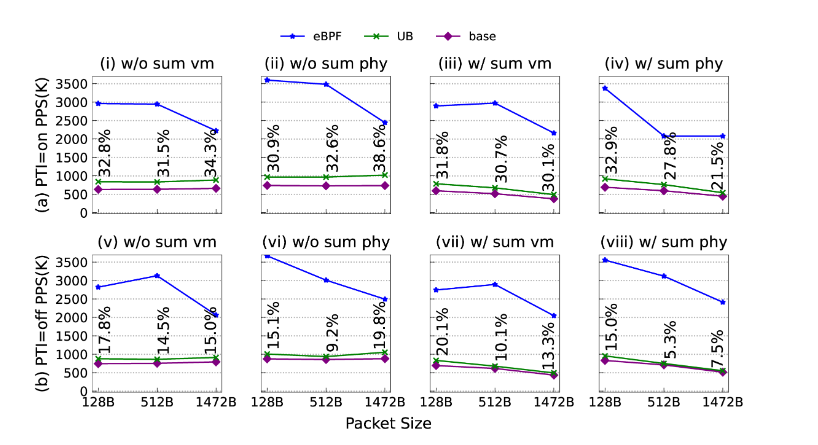


图10 服务器处理不同数据包大小的传入 UDP 数据包的 PPS

计算。我们还考虑添加轻量级计算工作负载，即数据包求和，如 NVMe 文件访问实验（第 6.1 节）。在VM中，当数据包大小增加时，原始套接字的PPS会出现更大的下降，但UB仍然可以以类似的比率加速原始套接字（KPTI开启时为30.1%±0.20% 31.8%±0.50%，以及10.1%±0.18% – 20.1%） KPTI 关闭时为 ±0.15%）。 eBPF 能够在不进行数据包求和的情况下保持类似的 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 开启的情况下看到 PPS 大幅下降。

分析执行性能。我们分别分析了使用 RDTSCP 进行数据包求和的情况下 BTC 和 eBPF 的执行时间，就像我们在 Redis 上的实验（第 6.2 节）。在开启 KPTI 的 VM 中，处理 33.85M 个 128B 的传入数据包，BTC 花费了 5.86 秒。相比之下，eBPF 的成本为 9 秒。我们可以看到，BTC 的执行性能优于 eBPF VM。然而，根据之前的结果，UB仍然无法实现与eBPF类似的PPS。根据我们的分析，原因是eBPF运行在softirq中，因此数据包可以被调度到不同的内核中。相比之下，原始套接字协议栈具有用于并发访问的内核锁。特别是，我们为套接字读取添加了更多线程，但根本没有看到 PPS 增加。我们还尝试通过将 NIC 的 IRQ 限制为单核并在打开 KPTI 的 VM 中重复求和实验来评估 eBPF 在没有多线程的情况下如何工作。 UB加速套接字对于三种数据包大小分别达到1M、0.96M和0.93M PPS，而eBPF分别达到0.96M、0.93M和0.91M PPS。因此，我们相信如果内核针对并发访问优化其协议栈，原始套接字的PPS可以得到显着提高。一种可能的方法是构建更好的 UB 运行时，以便可以通过系统调用公开更深入的内核跟踪点，我们将其留作未来的工作。

1. **讨论**

**7.1 UB与 eBPF**

除了UB和eBPF的性能比较之外，这里我们还比较了它们的限制和安全保证。由于eBPF主要是为了数据包处理和内核跟踪而开发的，因此它对应用程序代码有很多限制。例如，eBPF 不是图灵完备，因为不允许无限循环 [33]。由于其对代码的广泛限制，eBPF验证器很容易产生误报，即合法代码被视为非法[14]。 UB不会给开发者添加任何限制，并且透明地翻译用户空间代码。

在性能方面，UB仅加速系统调用之后的路径，但eBPF可以附加到内核内部的许多跟踪点，这使得它更加灵活并且能够克服内核瓶颈。我们相信，如果内核通过系统调用公开更多跟踪点，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VM的隔离，而UB依赖于SFI翻译器的策略。针对 eBPF 的攻击也可能对 UB 有效，如第 7.2 节所述。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 UB 更容易，因为 eBPF 有官方规范，并且使用了精简的指令集。

**7.2 安全风险**

虽然我们遵循SFI原则来设计UB，但可能会引入新的安全风险。首先，UB 可能容易受到侧通道攻击，侧通道攻击根据微架构状态变化推断秘密。例如，Spectre 攻击已证明 eBPF 可以被利用来窃取内核内存，因为 eBPF VM 将用户空间代码编译为内核代码 [25]。 UB的BTC也可能被利用进行类似的攻击。为了减轻这种风险，应考虑防御推测攻击，例如，由编译器放置推测阻止指令[25]。其次，我们的 BTC 翻译器可能无法清理特权未记录的 X86 指令。为了减轻引入的风险，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，之前的研究表明内核竞争可能导致检查时间到使用时间（TOCTOU）攻击[28]。由于 BTC 运行时不强制执行快速路径的检查点和使用点之间的原子性，因此恶意用户空间代码可以利用内核竞争。缓解措施可以依赖主动检测内核竞争的现有防御措施 [20]。

**7.3 其他限制**

诚然，当开发人员采取正确的措施将诸如 DPDK 之类的内核绕过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上下文切换开销的减少，还来自简化且更高效的用户空间驱动程序。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仅减少了上下文切换开销。 UB 的主要优点是它不需要开发人员对应用程序进行任何更改（参见表 1）。因此，我们相信，当开发人员愿意重构代码或设计考虑内核绕过的新应用程序时，内核绕过将受到青睐。

UB 的目的不是取代异步 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 可以帮助开发人员将 I/O 与不同线程中的计算解耦，从而更好地利用多核。 UB 并没有为同步 I/O 任务提供比异步任务更多的 IOPS，但它可以与异步 I/O 联合使用。在某些情况下，异步任务的I/O线程仍然密集地调用系统调用来提交I/O，UB可以加速这些任务。

1. **相关工作**

第 2 部分调查了有关系统调用优化的相关工作。下面我们介绍其他相关工作。

动态二进制翻译（DBT）。 DBT 是一种强大的调试和检测方法 [3,19,22,48]。凯迪亚等人。在内核中提出了一种快速 DBT 来检测内核代码 [22]。我们的翻译器在间接分支处理方面与他们的翻译器有一些相似之处，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，其运行时的某些功能需要回滚。相反，我们的运行时永远不会回滚。

基于软件的故障隔离 (SFI)。在内核中强制执行 SFI 并不是一个全新的想法。 XFI 最初被提出用 SFI 来隔离内核模块，后来 LXFI 添加了内核 API 检查来限制通过内核 API 传播的故障 [13, 32]。 UB 以不同的方式使用 SFI 作为快速路径。

加速进程间通信 (IPC)。最近提出了一些利用硬件辅助来加速 IPC 的方案。与加速系统调用类似，它们也尝试最小化上下文切换开销。顾等人。建议借助英特尔处理器的最新创新，即 MPK [16] 来加速 IPC。米等人。借用了为虚拟化设计的硬件功能来加速IPC[35]。杜等人。建议在不涉及内核的情况下向 CPU 添加新的上下文切换功能 [12]。他们在 RISC-V FPGA 处理器上实现了原型。

1. **结论**

系统调用带来的开销对于高 IOPS 应用程序来说很突出，但现有的方法还没有完全解决这个问题，因为它们需要代码重构的努力。为了保持二进制兼容性，我们建议直接在内核中执行用户空间指令的用户空间旁路（UB）。 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。代码块受到限制以避免引入额外的危害，因此它们可以直接在内核中执行。借助 UB，当应用程序在具有 KPTI 的 VM 中执行时，对于 GET 下的 1B – 4KiB 数据大小，I/O 微基准测试可以加速 30.3 – 88.3%，Redis 等实际应用程序可以加速 4.4 – 10.8%。

**致谢**

我们感谢我们的牧羊人 Dan Tsafrir 提出的非常有价值的建议。复旦大学作者得到国家重点研发计划（批准号：2022YFB3102901）和上海市自然科学基金（批准号：23ZR1407100）的资助。

**参考文献**

1. Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC), 13(1):1–40, 2009.
2. ansyun. DPDK-Redis. https://github.com/ansyu n/dpdk-redis. Accessed: 2021-05-05.
3. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In USENIX annual technical conference, FREENIX Track, volume 41, page 46. Califor-nia, USA, 2005.
4. Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pages 30–40, 2011.
5. Zach Brown. Asynchronous system calls. In Proceedings of the Ottawa Linux Symposium (OLS), pages 8185, 2007.
6. Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. Black Hat, 8, 2008.
7. Jeff Caruso. 1 million IOPS demonstrated. https://www.networkworld.com/article/2 244085/1-million-iops-demonstrated.html. Accessed: 2021-12-01.
8. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS ’10, page 559–572, New York, NY, USA, 2010. Association for Computing Machinery.
9. Vitaly Chipounov and George Candea. Dynamically translating x86 to LLVM using QEMU. Technical report, EPFL, 2010.
10. Alibaba Cloud. Improving Redis performance through multi-thread processing. https://alibaba-cloud.m edium.com/improving-redis-performance-thr ough-multi-thread-processing-ca4d8353523f. Accessed: 2020-11-30.
11. DPDK. Data Plane Development Kit. https://www. dpdk.org/. Accessed: 2021-05-01.
12. Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In Proceedings of the 46th International Symposium on Computer Architecture, pages 671–684, 2019.
13. Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 75–88, 2006.
14. Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 1069–1084, 2019.
15. GlareR. Code repository of this project. https://gith ub.com/GlareR/UserspaceBypass. Accessed: 202209-25.
16. Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In 2020 USENIX Annual Technical Conference (USENIXATC 20), pages 401–417, 2020.
17. Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, ChienMin Wang, and Yeh-Ching Chung. HQEMU: a multithreaded and retargetable dynamic binary translator on multicores. In Proceedings of the Tenth International Symposium on Code Generation and Optimization, pages 104–113, 2012.
18. Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. An analysis of x86-64 instruction set for optimization of system softwares. Planning perspectives, page 152, 2011.
19. Andrew Jeffery. Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU. University of Adelaide Honors Thesis, 2009.
20. Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In 2019 IEEE Symposium on Security and Privacy (SP), pages 754–768. IEEE, 2019.
21. EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 489–502, 2014.
22. Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In Proceedings of the TwentyFourth ACM Symposium on Operating Systems Principles, pages 101–115, 2013.
23. Kernel.dk. Efficient IO with io\_uring. https://kern el.dk/io\_uring.pdf. Accessed: 2021-12-01.
24. Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), 2016.
25. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1–19. IEEE, 2019.
26. Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–15, 2020.
27. Dmitry Kuznetsov and Adam Morrison. Privbox: Faster system calls through sandboxed privileged execution. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022.
28. Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In 30th USENIX Security Symposium (USENIX Security 21), pages 2363–2380, 2021.
29. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), pages 973–990, 2018.
30. Zhiyun Luo. How does Redis process requests? (translated). https://www.luozhiyun.com/archives/6 74. Accessed: 2022-09-25.
31. Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. ACM SIGARCH Computer Architecture News, 41(1):461–472, 2013.
32. Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 115–128, 2011.
33. Andrea Mayer, Pierpaolo Loreti, Lorenzo Bracciale, Paolo Lungaroni, Stefano Salsano, and Clarence Filsfils. Performance monitoring with H^2: Hybrid kernel/eBPF data plane for SRv6 based hybrid SDN. Computer Networks, 185:107705, 2021.
34. Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In USENIX winter, volume 46, 1993.
35. Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–15, 2019.
36. Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pages 59–73, 2019.
37. M. Prandini and M. Ramilli. Return-oriented programming. IEEE Security and Privacy, 10(6):84–87, 2012.
38. Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, and Richard D Schlichting. Cassyopia: Compiler assisted system optimization. In HotOS, volume 3, pages 1–5, 2003.
39. Redis. Redis Benchmark. https://redis.io/docs/ reference/optimization/benchmarks/. Accessed: 2022-09-25.
40. Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: a library OS with software componentisation for practical isolation. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 546–558, 2021.
41. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In 2015 IEEE Symposium on Security and Privacy, pages 745–762. IEEE, 2015.
42. Amol Shukla, Lily Li, Anand Subramanian, Paul AS Ward, and Tim Brecht. Evaluating the performance of user-space and kernel-space web servers. In CASCON, volume 4, pages 189–201, 2004.
43. Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI’10, page 33–46, USA, 2010. USENIX Association.
44. Gang Tan. Principles and implementation techniques of software-based fault isolation. Now Publishers, 2017.
45. Tencent. F-Stack. https://github.com/F-Stack/f -stack. Accessed: 2022-09-25.
46. Tencent. F-Stack Redis. https://github.com/F -Stack/f-stack/tree/dev/app/redis-6.2.6. Accessed: 2022-09-25.
47. The kernel development community. Page table isolation (PTI). https://www.kernel.org/doc/html/la test/x86/pti.html. Accessed: 2021-12-01.
48. Nigel Topham and Daniel Jones. High speed CPU simulation using JIT binary translation. In Workshop on Modeling, Benchmarking and Simulation (MOBS), 2007.
49. Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue inflight data load. 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. Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. 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, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In 2009 30th IEEE Symposium on Security and Privacy, pages 79–93. IEEE, 2009.
53. Kai Yu, Chengfei Zhang, and Yunxiang Zhao. Web service appliance based on unikernel. In 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pages 280–282. IEEE, 2017.