Userspace Bypass: Accelerating Syscall-intensive Applications

Zhe Zhou, Yanxiang Bi, Junpeng Wan, and Yangfan Zhou, *Fudan University;*
Zhou Li, *University of California, Irvine*

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

This paper is included in the Proceedings of the
17th USENIX Symposium on Operating Systems
Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the
17th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Userspace Bypass: Accelerating Syscall-intensive Applications

Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou
Fudan University
{zhouzhe, 19210240167, 19210240003, zyf}@fudan.edu.cn}

Zhou Li
University of California, Irvine
zhou.li@uci.edu

Abstract

Context switching between kernel mode and user mode often causes prominent overhead, which slows down applications with frequent system calls (or syscalls), e.g., those with high I/O demand. The overhead is further amplified by security mechanisms like Linux kernel page-table isolation (KPTI). To accelerate such applications, many efforts have been put in removing syscalls from the I/O paths, mainly by combining drivers and applications in the same space or batching syscalls. Nonetheless, such solutions require developers to refactor their applications or even update hardware, which impedes their broad adoption.

In this paper, we propose another approach, userspace bypass (UB), to accelerate syscall-intensive applications, by transparently moving userspace instructions into kernel. Userspace bypass requires no modification to userspace binaries or code and achieves full binary compatibility. Specifically, to avoid overhead caused by frequent syscalls, kernel identifies the short userspace execution path between consecutive system calls, and converts the instructions in the path into code blocks with Software-Based Fault Isolation (SFI) guarantee. According to our evaluation, I/O micro-benchmark can be accelerated by 30.3 – 88.3%, Redis GET Requests Per Second (RPS) can be improved by 4.4 –10.8% for 1B – 4KiB data sizes, when the application is executed in a virtualized setting with KPTI turned on. The performance boost will be reduced when KPTI is turned off.

1 Introduction

System call (syscall) is widely used by a userspace application to access the resources provided by the hosting operating system (OS) and extensively used for I/O operations. However, syscall could incur prominent performance overhead [43] when mechanisms like Linux kernel page-table isolation (KPTI) [47] are turned on. Arguably, syscall is one of the major performance bottlenecks for applications pursuing high I/O requests Per Second (IOPS), e.g., those requesting over a million IOPS [7].

Syscall-refactoring approaches. In the recent literature, there are mainly two streams of work in achieving higher IOPS by changing how syscalls are processed from the I/O path, which we call syscall-refactoring approaches: 1) The first stream of approaches integrate drivers and data processing logic in the same address space by moving data processing logic into kernel [26, 36, 53] or moving drivers responsible for I/O into userspace (kernel bypass) [21, 51]. In this way, the processing logic can directly talk to I/O devices and avoid the overhead caused by the switching between user mode and kernel mode [51]. 2) The second stream of approaches batch syscalls and allow userspace processes to queue multiple I/O requests and issue them together with only one single syscall [43]. However, these solutions require developers to change their code, which is usually a non-trivial task.

Our approach. In this paper, we propose *userspace bypass* (or UB for short), which reduces the overhead introduced by syscall-related I/O and achieves *binary compatibility* (i.e., no application code needs to be changed or rebuilt) at the same time. UB is motivated by the observation that applications with high IOPS do not execute many instructions between two consecutive syscalls (see Section 3.1). As a result, we can transparently instrument the instructions between syscalls under pre-defined security requirements (i.e., translating the instructions into sanitized code blocks), and let kernel execute the blocks without returning to userspace. In this way, the overhead caused by consecutive syscalls can be avoided. Figure 1 illustrates this idea.

Yet, a few challenges should be addressed. First, only those instructions that will potentially be executed between *frequently* invoked consecutive syscalls deserve userspace bypass. However, without explicit information provided by the developer, it is difficult to find such syscall sequence. As elevating the instructions to kernel also introduces overhead, the syscalls to be optimized need to be carefully chosen to offset such overhead. Second, malicious applications may exploit UB to steal kernel data and even execute privileged instructions. In addition, buggy applications may pollute kernel memory. Hence, it is critical for UB to guarantee kernel

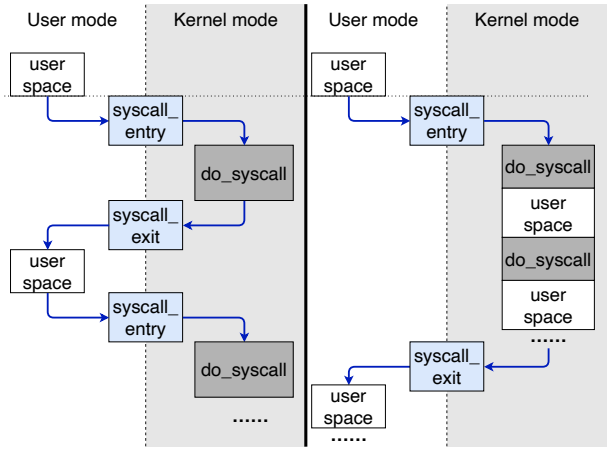

Figure 1: Invoking system calls without and with UB.

safety by performing comprehensive sanitization on userspace code and data. Finally, to achieve binary compatibility, the elevated application code should be oblivious about whether they are executed in kernel mode or user mode. The identical execution outcomes with and without UB should be guaranteed, including memory order and atomicity for multi-thread applications.

We address these challenges by adapting Dynamic Binary Translation (DBT) [52] and Software-Based Fault Isolation (SFI) [44] techniques. First, we profile syscalls by hooking their entries, to learn which syscall invocations are frequent (i.e., "hot" syscalls). Inspired by Just-In-Time (JIT) compilation [17, 22, 48], we can obtain the userspace instructions following the hot syscalls in the runtime. The instructions, if within the same function, will be translated into Binary Translation Cache (BTC). Next, we iteratively execute the BTC and extend the BTC from the exit instruction until we meet the next syscall invocation. We perform instruction and address sanitization to restrict the behaviors of BTC, and achieve kernel control-flow integrity (CFI) and data integrity on the BTC. UB does not re-order instructions or split memory access. As a result, other threads can execute concurrently and safely with the thread optimized by UB.

We implement a prototype of UB and evaluate its performance gain in I/O micro-benchmark and real-world applications including Redis and Nginx. Under our default setting (the tested application runs in a virtual machine (VM) and the Linux KPTI is turned on), I/O micro-benchmark threads can be accelerated by 30.3% to 88.3%. For Redis `GET`, the acceleration ratio ranges from 4.4% to 10.8% for 1B – 4KiB data sizes. Nginx can be accelerated by 0.4% – 10.9%. UB can accelerate raw socket-based packet filters by 31.5% – 34.3%. We also evaluate the impacts of KPTI and virtualization on UB's performance gain. Since turning off KPTI reduces the syscall overhead, UB is less effective. For example, the acceleration ratio for I/O micro-benchmark drops from 88.3% to 41.6% for the smallest I/O size. Hence, future processors, which are expected to eliminate Meltdown and Spectre vulnerability in hardware, will benefit much less from UB. When the applications run in the physical machine, UB achieves higher upper-bound acceleration ratios in most settings compared to VM, because IOPS is usually higher in this case, which results in more syscalls that can be optimized.

We also compare UB with other systems that optimize syscalls, including io_uring [23], F-Stack DPDK [45] and eBPF [34] in our experimental study. The results show that UB is less advantageous, comparing with io_uring in the micro-benchmark, F-Stack for the Redis macro-benchmark, and eBPF for raw sockets. However, UB has a unique advantage that no code change is required for the application developers.

Finally, we acknowledge UB might introduce new security risks under side-channels, undocumented x86 instructions, and kernel races. We accordingly suggest a few defense ideas.

The code of our UB prototype is published at [15]. We summarize the contributions of this paper as follows.

- We propose userspace bypass (UB), which directly executes the instructions between syscalls in kernel mode, to accelerate syscalls.
- We provide a concrete design that transparently translates userspace instructions to kernel-safe, sanitized BTC. With this method, existing applications can be executed without modification and enjoy the performance gain.
- We implement a prototype and evaluate it against several high IOPS apps. The results prove the effectiveness of UB.

2 Background

In this section, we first overview the syscall mechanisms and their introduced overhead. Then, we describe the prior efforts in reducing such overhead.

2.1 Syscalls and Their Costs

Syscall presents the default interface between userspace applications and kernel services. Software interrupt (e.g., `int 0x80`, which has been deprecated) and special instructions (e.g., `syscall`/`sysret` created by AMD and `sysenter`/`sysexit` created by Intel) can be leveraged to transfer the control from user space to kernel space and vice versa after syscall.

Previous studies have shown that syscall invocation can introduce prominent overhead on various applications and scenarios [16, 35, 43], including *direct costs* and *indirect costs* [43]. For the first case, because of switching between user mode and kernel mode, extra procedures have to be executed to save registers, change protection domains, and handle the registered exceptions. For the latter case, the state of processor structure, including L1 cache data and instruction

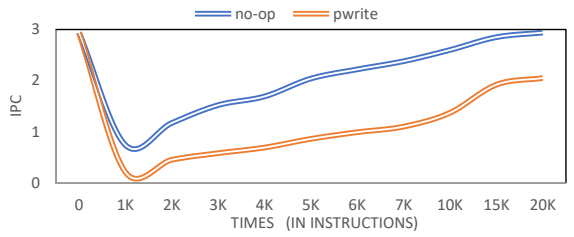

Figure 2: IPC after no-op syscall and `pwrite` syscall, measured on our platform (Intel Skylake and Linux).

Scheme	Develop Cost	Async Needed	Accele-ration	Popula-rity
eBPF	++	✓	++	++
DPDK	++	✓	+	++
io_uring	++	✓	+	+
UniKernel	+++	-	+++	-
FlexSC	+	-	+	-
UB	-	-	+	

Table 1: Comparison of schemes for optimizing syscalls.

caches, translation look-aside buffers (TLB), etc., can be *polluted* by syscalls, and the Out of Order Execution (OOE) of CPU has to be stalled for the order guarantee. As a result, the user-mode instructions per cycle (IPC) would be decreased after syscall.

A widely-used technique called kernel page-table isolation (KPTI) [47] makes syscalls even slower. To defeat transient execution attacks, e.g., Meltdown [29] and Spectre v3a [49], OS kernel uses two sets of page tables for user space and kernel space. As a consequence, CPU should switch to kernel page table upon entering syscall, and switch back when returning to userspace. Besides KPTI, virtualization may also increase the context-switching overhead. For example, the overhead of TLB miss (part of indirect overhead) inside VM can be larger, as more page table entries have to be examined than inside physical machines.

Below we summarize the observations from previous studies and our measurement about the concrete syscall overhead.

- A no-op system call with KPTI enabled can cost 431 CPU cycles, as measured by Mi et al. on Intel Skylake and seL4 [35].
- As measured in our experiment platform (Intel Skylake and Linux), the kernel prologue and epilogue (direct costs) take 197 instructions (992 CPU cycles) for a no-op syscall, suggesting the issue of syscall overhead persists a decade after the study of Soares et al. [43].
- Also on our platform, a `pwrite` syscall can degrade the IPC of the following userspace instructions from 2.9 to 0.2 (indirect costs). The IPC slowly goes back to 2.1 after executing *20,000* instructions. Figure 2 shows the trend of IPC by time elapsed.

2.2 Performance Optimization on Syscalls

The research community is actively working on mitigating the overhead resulting from syscalls. Below we describe the related work with a comparison to our approach (also summarized in Table 1).

Asynchronous syscalls. Syscall introduces a synchronous execution model, as the user-mode execution is resumed after a syscall is finished. Brown proposed non-blocking Linux syscalls [5] that can be completed asynchronously parallel to the userspace execution flow. But, this approach does not completely decouple the syscall invocation from its execution. So far, most of the syscall implementations on Linux are still synchronized.

Syscall batching. As locality is a major performance factor, executing syscalls in a batch has also been investigated. Rajagopalan et al. proposed to group consecutive syscalls into one (the result of a syscall is directly fed to the next) [38]. This approach is effective under the assumption that no computation happens between two syscalls. Soares et al. proposed to batch syscalls of multiple co-routines and asked the developers to change the thread model to M-on-N ("M user-mode threads executing on N kernel-visible threads, with M » N") [38]. Thus, it only works when the task can be split into many threads. Modern kernel provides native queues, i.e., io_uring [23], to batch I/O requests from userspace processes and reduce the occurrences of syscalls. In particular, userspace code can issue multiple requests to the queue and invoke one syscall to let kernel process the queue[1].

Unikernel. To mitigate the overhead of context switching, the Unikernel solutions run application code in kernel space instead of user space. Examples include Loadable Kernel Module (LKM) [42] and library OS [31, 40].

In-kernel sandbox. To reduce the occurrences of context switching caused by syscalls, in-kernel sandbox allows application code to run in privileged mode. For instance, eBPF [34] allows developers to attach code into kernel trace points. When kernel reaches these points, it will use a VM to execute the attached code. However, eBPF places many restrictions on the code, and kernel verifies if all the requirements are met before execution, during which legal codes may be rejected because of false positives in verification. Recently, Dmitry et al. propose to use in-kernel sandbox to execute applications entirely in kernel [27], in which context switching overhead can also be mitigated.

Kernel bypass. Observing that kernel does not always have to be involved in I/O tasks like packet handling, some researchers proposed the kernel bypass approaches. One prominent example is the Data Plane Development Kit

[1] io_uring also supports a kernel polling mode if the application has root privilege, where no syscall is required.

(DPDK) [11], which takes over I/O devices in userspace. Specifically, I/O requests are submitted to devices via a shared ring buffer, instead of syscalls. The buffer is maintained in userspace, involving no kernel activity for I/O.

We find that existing approaches all require noticeable development efforts. Different coding paradigms have to be followed in order to use the syscall-refactoring primitives. Most kernel bypass and syscall batching solutions (e.g., DPDK, RDMA, io_uring) require application code to interact with a queue pair asynchronously. Nonetheless, developers still prefer to write program logic in the synchronous style. Refactoring the legacy code is also labor-intensive. As an example, we compare the unofficial Redis with DPDK support [2] to its official version (version 3.0.5). We find that the former includes 9,984 extra lines of code (LoC) to support DPDK, which accounts for *10%* of the LoC of the official version. Another example is Unikernel: It requires developers to write kernel-mode code, which is unfortunately difficult to debug and prone to errors like memory corruption (there is no memory isolation). In addition to changing the application code, special userspace drivers may be required for kernel-bypass solutions [24].

3 Design Overview

To address the aforementioned issues, we propose userspace bypass (UB), a new primitive for syscall optimization. UB aims to fulfill the following three design goals (DG).

DG1: Minimizing the manual efforts of developers. Different from syscall-refactoring approaches, which require developers to change their legacy code or adjust to asynchronous programming, UB optimizes the syscalls at the *execution time*, which meanwhile does not impact application's functionality.

DG2: Minimizing changes to system architecture. Syscall-refactoring approaches may change the current system architecture, e.g., mapping and binding devices to userspace. In contrast, UB keeps the current system architecture unchanged, including device driver and I/O harvesting models.

DG3: Comparable performance to syscall-refactoring approaches. UB aims to reduce the direct and indirect costs of syscalls, and achieve similar performance boost compared to syscall-refactoring approaches.

3.1 Syscall-intensive Applications

We focus on optimizing applications of high IOPS, e.g., Redis and Nginx, which are also syscall-intensive. By analyzing their code and runtime behaviors, we identify the following two insights that guide the design of UB.

Lightweight userspace instructions in I/O threads. We find that the computation workloads between I/O events are usually lightweight for the examined applications. Moreover, the number of instructions between two consecutive syscalls is usually small. One explanation is that such applications follow a popular I/O model that separates I/O-intensive workload from CPU-intensive workload in different threads. For example, Redis server has a main thread that dispatches accepted sockets to I/O threads [10], which conduct I/O from/to kernel and let the main thread complete the CPU intensive computation. With such a design, the instructions between I/O events simply handle buffer movement. We also profiled syscalls invoked by Redis (in total 3M), and found half of them are followed by less than 400 userspace instructions (around 200 cycles when IPC is 2) before the next syscall, which is faster than executing a syscall itself (e.g., 431 cycles [35] as described in Section 2.1).

Amplified direct and indirect costs. Section 2.1 overviews the direct and indirect costs of syscall in general, and those costs can be amplified in syscall-intensive applications. As shown in Figure 1, the frequency of entry and exit rises linearly following the frequency of syscall invocation. The indirect costs due to TLB misses, OOE stalls and cache misses are also non-negligible, especially when the syscall handles lighter tasks (IPC drops to 0.74 for no-op syscall, and 0.21 for `pwrite`, as shown in Figure 2).

3.2 UB Modules

Based on the above considerations, we are motivated to design UB in a way that it can detect the occurrences of syscalls, and elevate the userspace instructions between consecutive syscalls to kernel through binary transformation. Figure 1 illustrates our idea. Although the idea seems simple at the high level, a few challenges should be addressed to enable UB for real-world, full-fledged applications.

- The application code is less trustworthy compared to the kernel code. Hence, necessary *isolation* should be performed to confine its capability after being moved to kernel. However, identifying the untrusted regions and governing them with the right policies are non-trivial.
- Given that isolation would incur extra costs, it is not always beneficial to transform every chunk of userspace instructions. But, when to perform transformation and how to reduce its overhead are unknown.

UB addresses these challenges with three key components. 1) A "hot" syscall identifier that monitors the execution of the target application, profiles the invoked syscalls, and determines when userspace instructions need to be elevated; 2) a *Just-in-Time (JIT)* translator that converts the userspace instructions into *Binary Translation Cache (BTC)* that is instrumented with isolation policies; 3) a *kernel BTC runtime* that executes the translated code. Figure 3 overviews the design of UB.

Note that the components in UB are not fundamentally new concepts. BTC is a standard component for Dynamic Binary Translation (DBT) [3, 22]. The JIT translator follows the guideline of Software-Based Fault Isolation (SFI) [44] in

code instrumentation and isolation policies. Yet, we find that the existing systems cannot be directly used in our problem setting. Below we briefly discuss the main modules in UB.

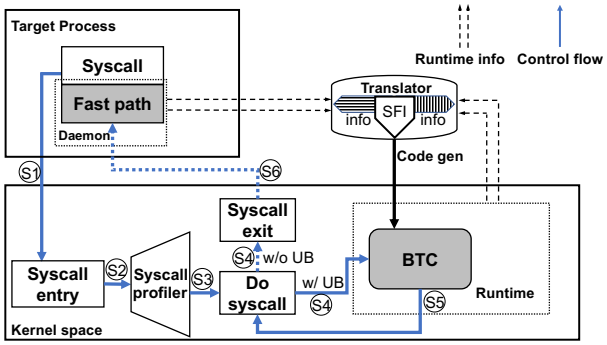

Figure 3: Overview of the UB framework. Every time a thread calls syscall (S1), hot syscall identifier hooks it (S2) and dispatches it to `do_syscall` (S3), after which kernel may return to user mode if the syscall is not hot (S6), or send it to BTC runtime for UB (S5).

Hot syscall identifier. This module runs in kernel mode and hooks each syscall. By analyzing the runtime statistics, it can identify which syscall instructions are *hot*, i.e., ones with high chances to be followed by another syscall shortly. The userspace instructions between two consecutive hot syscalls will be elevated to kernel and accelerated next time when the application runs. To avoid introducing large overhead due to runtime monitoring, this module runs intermittently.

BTC translator. The BTC translator converts the userspace instructions marked by the previous modules to BTC and has it executed by the kernel BTC runtime. Under the SFI guidelines, it converts dangerous instructions (e.g., indirect control-flow transfer) to the safe ones (e.g., direct jumps), and instrument checks to constrain memory access and control-transfer behaviors. The translator runs in a separate, independent userspace process to avoid introducing its code to kernel. The translation does not block the application execution, and the translated code is executed *next time* when the same code path is visited.

In addition to optimizing userspace instructions between a pair of hot syscalls, we also consider the acceleration on a *sequence* of hot syscalls. We call the enclosed userspace code *fast path*. UB aims to chain such userspace code and accelerate them altogether. The fast path is discovered incrementally by watching the jump targets. The details will be discussed in Section 5.

4 Hot Syscall Identifier

Criteria of userspace bypassing. A region of userspace instructions should be elevated when its performance gain outweighs the translation and instrumentation overhead by BTC translator. We measure the performance gain against different userspace path length (i.e., the number of instructions), and consider the regions with *short* path. The major reason is that the instrumentation costs increase rapidly for longer paths, because more instructions have to be monitored. We consider 1,000 instructions (termed T_{path}) as the threshold for the short path length[2]. Through an empirical study, we observe obvious performance gain (over 20%) with this path length (see Section 6.2).

Module design. This module aims to discover hot syscalls that enclose a short userspace path. We resort to online analysis to achieve seamless profiling. Specifically, this module hooks syscall entry and counts the number of instructions between two consecutive syscalls. The two syscalls are classified as candidates of hot syscalls when the instruction number is less than T_{path}. Below we describe the detailed steps.

- **Syscall sampling.** Monitoring every syscall invocation will introduce high performance penalty to the application execution. Hence, we sample syscalls and conduct the follow-up analysis only when a thread is issuing syscalls frequently (e.g., I/O threads). According to our measurement on syscall-intensive applications (e.g., Redis and Nginx), at least 100K syscalls (termed T_{sys}) are issued per second (6M per minute), and we choose to profile less than 10% of T_{sys} syscalls (up to 500K syscalls every minute). Therefore, most syscalls are not sampled and not interfered.
- **Coarse-grained profiling.** To further reduce the profiling overhead, we check whether a monitored thread invokes syscalls at high frequency. If the thread invokes less than 50K syscalls per second (half of T_{sys}), the module will not conduct the next fine-grained syscall profiling. In this, the threads with low IOPS will be skipped.
- **Fine-grained profiling.** For a thread invoking syscalls frequently, this module further analyzes which syscall instructions are invoked frequently. The frequent ones deserve userspace bypassing as more performance boost can be gained. We monitor 15K syscalls (15% of T_{sys}) of each round, and maintain a table recording, for each invoked syscall instruction, its location register (`RIP`) and a counter of how many times the next syscall is invoked within 4 microseconds (approximately the time of executing T_{path} instructions). We consider a syscall frequent when the counter is larger than 900 (6% of the profiled 15K syscalls). These syscalls and their enclosed userspace instructions will be handled by BTC translator in the next stage.

One might wonder if the performance of this module is sensitive to the parameter selection. To test the sensitivity, we

[2] Soares et al. consider the invocation of a syscall frequent if it is invoked once every 2,000 or less instructions [43]. We use a more conservative number to accommodate different platforms.

check if hot syscalls of Redis and Nginx, two applications used by our experiments, can be correctly discovered on three different machines: a PC with Core i5 10500 (year 2021), two servers with Xeon 8175 and 8260 (year 2017 and 2019). All hot syscalls can be correctly identified, suggesting parameter tuning could be skipped in most cases.

5 BTC Runtime and Translator

In this section, we describe how the BTC translator converts userspace instructions into kernel BTC and meets the security requirements. Our BTC translator follows the procedure of Dynamic Binary Translation (DBT) [19, 22, 48]. In general, given a path consisting of basic blocks in binary and triggering an event (e.g., hot syscall in our case), DBT dissembles it, translates it with a SFI rulebook, and compiles it to BTC for the future execution. Due to SFI, the malicious or unwanted behaviors of the translated code can be contained, and safely run by the BTC runtime.

5.1 BTC Runtime

The translated code block is executed by a BTC runtime in kernel. The BTC runtime holds local variables in kernel stack, which can be accessed by the instrumentation instructions within the BTC for policy enforcement and context switching. The local variables include: 1) the saved kernel context, i.e., callee-saved registers, 2) the values of reserved registers, and 3) the indirect jump destination information which is used to build the fast path.

Before executing the BTC, the runtime prepares the return status for userspace, i.e., through restoring the userspace context saved on syscall entry (e.g., `pt_regs` for x86_64). After a block is finished, the runtime processes the return status of the BTC and takes further actions. The execution of a BTC might exit the runtime in the middle when the jump target is missing, e.g., when a new path is encountered. In this case, the runtime records the information about this jump and immediately returns to userspace, i.e., the jump target. We make the userspace memory accessible to the BTC runtime, so all changes on memory are kept. Changes made to registers are updated to userspace context (i.e., `pt_regs` for x86_64), which will be written to registers when kernel returns to userspace. Therefore, userspace state changes made by the BTC are also preserved and visible to other threads, which ensures the application logic is not changed under UB.

The execution of the BTC might also exit when a syscall instruction is encountered. In this case, a fast path between two consecutive syscalls has been completely executed in kernel, which indicates a successful userspace bypass. The BTC runtime emulates the syscall trap, by looking up the syscall number against the syscall table and dispatching syscall parameters to the corresponding `do_syscall` function (i.e., executing the syscall). After `do_syscall` returns, the BTC runtime checks if the next syscall instruction is again hot. If the answer is yes, the runtime tries to conduct another userspace bypass. In this way, `do_syscalls` and userspace bypass can be chained, which is similar to direct branch chaining of DBT. In an ideal case, *a whole thread can be executed in kernel*.

Fast path discovery. The performance of UB highly depends on the identification accuracy of fast path, and we leverage an incremental, JIT-style approach to achieve high accuracy. Given an entry address, i.e., the instruction next to a hot syscall, the BTC translator first discovers a part of the fast path, by dissembling the code segment of the target thread from the entry address iteratively. The potentially unreachable paths are skipped by the translator in each iteration. Specifically, the translator only follows *direct jumps* and stops at the call instructions, which forces the translator to handle code only within a function at one iteration and consider it fast path. When an indirect jump or call is indeed made later, the target information will be collected by the BTC runtime and sent to the translator to extend the fast path after replacing the jump instructions (see Section 5.2.1). Such an approach is similar to the one adopted by QEMU [9], but we do not lift the binary to its intermediate representation.

5.2 BTC Translator

Below we describe how the security policies are instrumented into the userspace code. We follow the SFI principles to provide *data-access policies* and *control-flow policies* [44] on *kernel*, and the implementations are inherited and extended from Nacl [52], which sandboxes the untrusted x86 native code in browser. Noticeably, Nacl assumes source code is available so SFI rules can be enforced under static compilation. In contrast, UB performs DBT on the binaries. As such, the SFI rules have to be adjusted and extended.

Threat model. We assume the userspace code is untrusted, which could contain *arbitrary* code and data, and the side-effects include unmediated access to kernel memory, privileged functions, etc. The goal of UB is to ensure the userspace code cannot gain more privilege (and do more harm) after it is elevated to kernel, i.e., protecting kernel's control-flow integrity. Noticeably, this goal is different from guaranteeing control-flow integrity [1] on the userspace application (elaborated in Section 5.3). We take a *conservative* approach in designing UB and avoid elevating a fast path when the consequences can not be immediately determined (e.g., the jump targets are unknown during translation). We focus on x86_64 platform but the proposed techniques could be easily generalized to other platforms. Below we describe the implementations related to jump, register, instruction, and memory access that ensure security under this threat model.

5.2.1 Jump Sanitation

The inner sandbox of Nacl checks the *explicit* control flow expressed with calls and jumps, and disallows memory dereferencing on indirect jump and call instructions. The targets of jumps are confined within the sandbox. In contrast, the entire kernel memory space is open to elevated userspace code under UB. Therefore, we take different approaches to sanitize jumps.

Direct jump. To prevent the code in BTC from jumping to an arbitrary address, the translation only happens when the jump target is known. In other words, only direct jumps whose targets are known are processed. The address sanitization is described in Section 5.2.3.

Indirect jump. Yet, userspace fast path may contain indirect jumps, and we deter BTC from processing such path till the targets are known. In particular, the translator inserts checks that compare the targets against a *target address table* (similar to jumptable [22]) when encountering the associated code at first. If the target address is not in the table, the control flow will exit BTC runtime. When such an exit is triggered during executing a BTC block, the BTC runtime sends the jump instruction address (i.e., `RIP` of the address) and the target address to the BTC translator, and extends the fast path, as described in Section 5.1.

We show an example in Figure 4. The indirect jump (jump to `RAX`, located at `0x123`) is initially translated to writing down the jump target (saving `RAX` to stack) and exiting to BTC runtime (jump to `exit_indirect_jmp`). When the path `P1` is firstly executed, the BTC runtime learns a target `0x456`, and the information is sent to the translator, which updates the BTC by adding a target table entry. After that, the path `P1` is added to the BTC, and it will not trigger `exit_indirect_jmp` for the next time. If `P2` is reached later, another destination `0x789` can be learnt and the BTC will be updated, so the fast path is further extended.

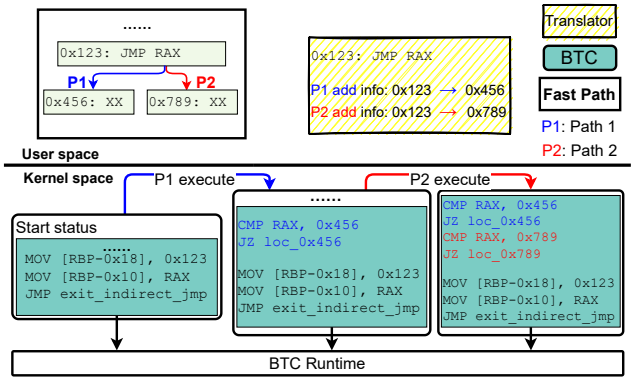

Figure 4: An example of translation under jump sanitation.

As the application runs longer, more indirect jump targets can be learned. The resulting BTC can eventually cover the entire fast path. The checks inserted into the BTC can perform efficiently because: 1) indirect control-flow transfer instructions do not appear frequently, based on our empirical analysis on the syscall-intensive applications and previous studies [18]; and 2) CPU is allowed to speculatively jump to the destination under out-of-order execution without waiting for the destination check.

5.2.2 Register Remapping

To protect kernel registers and stack, the BTC translator disallows the BTC code to access stack registers (i.e., `RSP`, `RBP`, and `RIP`). Besides, some registers are reserved for BTC runtime and cannot be accessed by the BTC code as well. Hence we develop this module to manage the registers.

Specifically, the BTC translator uses the M reserved registers in BTC to serve the potential access to N registers ($N = M + 3$, 3 are for stack registers). As $M < N$, the translator needs to schedule registers. The N registers have their values stored in local variables, and the translator chooses one from the M reserved registers to temporally act as a special register with *renaming*. The translator also inserts code to synchronize the N registers to local variables on stack. As a result, the behaviour of the BTC code is the same as the fast path in the user space.

Register reservation. The translator reserves `R12`- `R15` ($M = 4$) for BTC runtime use, as they are the least frequently used in common userspace applications (less than 1% usage frequency [18]). When they appear in the fast path, renaming will occur. We also optimize our renaming mechanism for frequently-used special registers (i.e., `RSP`), by letting the translator fix the reserved registers to hold their values. Doing so reduces the occurrences of the costly register synchronization.

5.2.3 Instruction Sanitization

Privileged instructions (e.g., `sysret`) are not allowed to appear in the BTC, to avoid privilege escalation by the malicious code that exploits UB. During translation, the translator avoids elevating a fast path to the kernel if it contains any privileged instruction.

Due to register remapping, some instructions have to be rewritten. For stack operation instructions like `PUSH`/`POP`, the translator substitutes them with multiple instructions. Take `POP` as an example. The translator first adds an instruction to `MOV` the operand to the popped target from the memory addressed by the reserved register (i.e., the acting stack pointer), and then updates the reserved register with the new stack pointer value, i.e., plus 8.

5.2.4 Memory Access Sanitization

To prevent unauthorized access to the kernel memory, the translator sanitizes all memory access instructions. For ev-

ery such instruction, the translator inserts address checking instructions before the instruction, such that only userspace addresses are allowed to be accessed, i.e., the addresses start with 0. Similar to address masking of SFI [44], the translator shifts left the address by one bit and then shifts it right by one bit, to fulfill the address requirement. Two extra instructions (i.e., `SHL` and `SHR`) are introduced to this end, but our evaluation suggested the extra overhead is negligible (0.4%). Note that the added checks do not prevent BTC from accessing *unmapped* memory region and triggering page fault, and we handle it with the procedure described below.

Page fault handling. We modify the page fault handler to monitor the page fault events. For minor fault and major page fault, the page fault handler behaves the same for kernel mode and userspace mode. Therefore, faults caused by the userspace applications are resolved in the same way as without UB. When invalid page fault (i.e., illegally accessing some memory regions) happens, the execution of BTC code is aborted.

Nacl also isolates the memory space between the extensions and the host browser, with the help of the segmentation provided by x86 CPUs. As such, extensions' instructions can only access memory within a segment and instructions to modify segment states are not allowed. However, although x86_64 still provides segmentation, it only adds a segment offset to the address but does not check segment boundary, which cannot be directly used for memory isolation.

5.3 Security Guarantees

The translated BTC has the following security properties (termed SP), and they jointly make UB fulfill SFI policies [44] on kernel.

SP1: Kernel control-flow integrity (CFI) for BTC. This property is guaranteed because when the BTC runtime hands control flow over to the userspace, the execution will only terminate through the exit point. More importantly, when the runtime executes the BTC, the thread cannot jump to a location unknown to the translator. For direct control-flow transfer, the destination can only be a label of a known basic block that has been translated. Indirect control-flow transfers are all translated to direct transfers by replacing the destinations. Therefore, the BTC prevents malicious code from hijacking the kernel control-flow after it is elevated.

We want to point out that UB does not claim to add extra protection against control-flow hijacking, e.g., ROP, JOP, COOP [4, 6, 8, 37, 41], and they can still occur in userspace. Though the attacker can construct gadgets when the destination checks are passed, jumping to the kernel code segment from BTC is never allowed, as it can be detected and aborted by the translator.

SP2: Kernel data (memory and register) integrity. For kernel context (or registers), we design BTC runtime to be compatible with the calling conventions, and the caller (kernel) context is saved on the stack before jumping to the BTC, which is recovered before returning to kernel instructions. The context switching is lightweight, as it does not cause privilege transfer.

For kernel memory, access sanitization ensures that no kernel memory can be accessed by the sanitized instructions, hence the kernel stack will not be tainted. Though runtime local variables must be accessible by the instructions in BTC, they cannot be exploited by malicious programs to touch the kernel stack. Only intentionally inserted instructions can touch the local variables referred by the stack base pointer, which stores runtime information like swapped-out registers (see Section 5.2.2). Because kernel CFI is guaranteed, execution would never jump to these instructions.

SP3: No privileged instructions in BTC. It is explained in Section 5.2.3.

SP4: Dead loop break. We also consider the attacks and bugs against the *availability* of the system resources. For instance, userspace applications may fall into a dead loop because of bugs or intentionally. As a countermeasure, the translator maintains a counter in BTC runtime to keep track of the number of instructions already executed. Once the counter exceeds a threshold, the execution flow can exit to runtime and in turn return to userspace, which avoids the kernel being blocked by the BTC code.

5.4 Thread Safety

Special attention should be paid to multi-thread userspace applications, because UB has no control over other threads except the one elevated to kernel. Memory order and atomicity have to be preserved to avoid data race. Fortunately, thread safety is automatically guaranteed by the translator and we explain it below.

Memory order. To preserve memory order, the translator regards all userspace memory as *volatile*, and only inserts instructions between userspace instructions without optimizing the block (e.g., reordering instructions or caching memory modification in registers). Yet CPUs can still reorder memory loads and stores according to their memory model. The original memory fences placed by the userspace applications are all inherited, and the translator does not insert extra fences.

Atomicity. The translator takes special measures to guarantee atomicity when using multiple instructions to emulate one userspace instruction. When translating an instruction, the translator prefers to use one instruction that has the same opcode as the original one. Hence, the atomicity of the original instruction is automatically preserved. For example, instructions with a lock prefix are translated to ones still with lock (e.g., `LOCK MOV`). If more than one instruction is needed for emulation, memory load or store must be completed in a single instruction. For example, when translating `PUSH RIP`, the offset address of the next instruction must be moved to the

userspace stack top. From the view of the translator, when BTC runtime reaches the instruction, the value of `RIP` is statically known and becomes an immediate number. However, x86_64 does not have an instruction to directly move a 64-bit intermediate value to memory. As a result, the translator generates instructions that first move the immediate value to a 64-bit reserved register and then move the 64-bit register to the top of the userspace stack.

6 Evaluation

We implement the prototype of UB for Linux kernel 5.4.44. The BTC runtime is implemented as a kernel module with 416 lines of C code, which hooks syscall epilogue to conduct syscall identification and manage BTC runtime. The translator is implemented with 786 lines of Python code at userspace (except the dependant Python disassembler `miasm` and gcc assembler `as`), which communicates with the BTC runtime kernel module via sys file. The kernel is modified by adding only 6 lines of codes to the syscall entry to allow the module to hook syscalls.

We evaluated our prototype in an I/O micro-benchmark and two real-world applications (Redis and Nginx) for macro-benchmarks. It is also compared to related technologies including DPDK, io_uring, and eBPF. To evaluate these applications, we set up a virtualized environment and a bare-metal environment. The bare-metal environment consists of a client machine and a server machine[3], which are connected within the 40G Ethernet LAN. The virtualized environment runs on the server, with NIC pass-through being enabled. For the micro-benchmark I/O experiment, we run the tests directly on server, as it does not require network. For other scenarios, we run the client application in the client machine and the server application in the server machine, so the traffic goes through the physical network. To show the effects of virtualization and KPTI, which impact the syscall performance as explained in Section 2.1, we run each server application in four settings: KPTI on/off × VM/physical machine. When KPTI is on, Linux turns on PCID to mitigate performance degradation. All the following tests are conducted 10 rounds, and the average IOPS or Requests Per Second (RPS) values are shown. For the results demonstrated in Section 6.1 to Section 6.4, we focus on the setting of VM with KPTI on and briefly describe how the results are changed under other settings. In Table 2, we list the acceleration ratios among different settings.

[3]The server machine has an Intel Xeon 8175 CPU (24 cores), 192GB memory, Samsung 980 pro NVME SSD, and Mellanox Connectx-3 NIC. It runs Ubuntu 20.04 with 5.4.44 kernel. When set up for VM, it uses QEMU-KVM 1:4.2-3, and assigns 24 cores to the VM. The client machine has an Intel Xeon 8260 CPU, 128GB memory and Mellanox Connectx-5 NIC.

	Test	VM	Physical
w/ PTI	**In-mem**	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%
	Socket	31.5% – 34.3%	30.9% – 38.6%
w/o PTI	**In-mem**	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%
	Socket	14.5% – 17.8%	9.2% – 19.8%

Table 2: Ranges of acceleration ratios for different settings. "In-mem" means the in-memory file access benchmark.

6.1 I/O Micro-benchmark

We first consider accelerating a thread that purely performs file I/O requests via blocking syscalls as the micro-benchmark, which approximates the best-case scenario for UB. The thread runs a tight loop that sequentially reads files from kernel to userspace buffer via `READ` syscall 8.39 M times. The real-world applications may exhibit different patterns like executing more instructions between consecutive I/O requests, reducing the acceleration ratios by UB. For comparison, we employ io_uring (liburing-2.2) for the same task (i.e., tight-loop `READ` syscall) and compare the IOPS.

In-memory file access. We create a large file in ramfs to avoid possible disk bottleneck, in order to assess how UB accelerates syscalls more accurately. Admittedly, this setting makes the micro-benchmark less realistic. We gradually increase the size of the buffer for each read and evaluate the acceleration ratios of UB under different buffer sizes.

Figure 5 shows the results. For the virtualized environment with KPTI on, UB accelerates syscall-based I/O by $88.3\%_{\pm 0.75\%}$[4], when the I/O size is small (64B). For larger I/O size, IOPS drops for both UB and baseline, and the acceleration ratio drops to $30.3\%_{\pm 0.96\%}$ for the 4KiB I/O size, because fewer syscalls are invoked. Turning off KPTI increases the IOPS, but the acceleration ratio of UB drops to $14.3\%_{\pm 1.83\%} - 41.6\%_{\pm 1.73\%}$, because the syscall overhead is reduced. The acceleration on physical machine is higher especially when the I/O size is small (e.g., $112.9\%_{\pm 1.78\%}$ when the I/O size is 64B when KPTI is on), as the IOPS on physical machine is higher and UB saves more context switching overhead.

For io_uring, we first examine the different queue depths (i.e., how many requests can be batched) from 1 to 1024, and found IOPS is stable after the depth reaches 128, as shown in Figure 6. Hence, we set the depth to 128 for its comparison with UB. It turns out io_uring yields more IOPS for most buffer sizes, according to Figure 5. When running in physical

[4]We report the acceleration ratio together with the standard deviation.

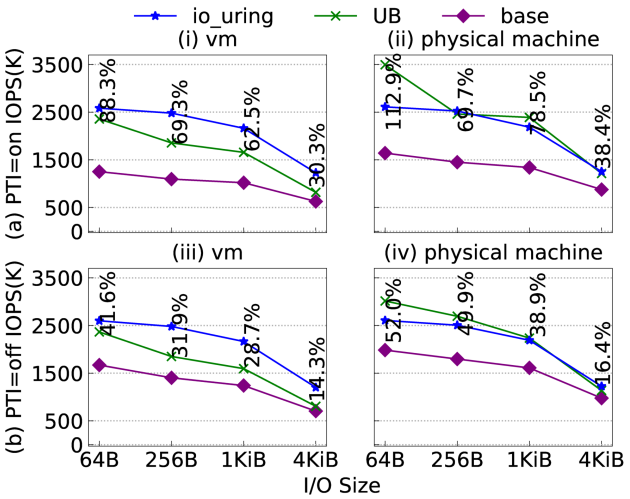

Figure 5: IOPS of READ syscalls, io_uring and UB syscalls against different buffer sizes. The percentage number is the acceleration ratio of UB against the baseline. Figure 7, 8, 9 and 10 follow this style.

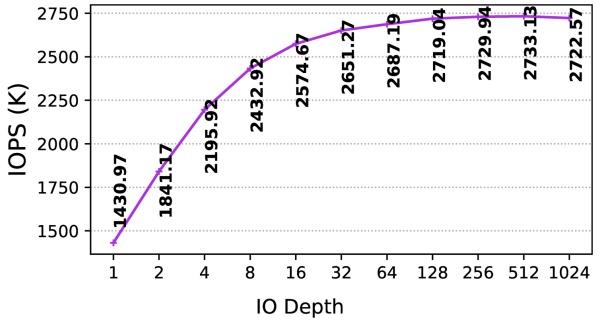

Figure 6: IOPS of different io_uring depth.

machine, for the small size (64 Byte), UB yields higher IOPS than io_uring, though it is expected io_uring should always outperform UB. We have not found a good explanation, but we notice that IOPS of io_uring increases by 13% when upgrading the Linux kernel from 5.4.44 (the version used by our testing environment) to 5.15. Hence, it is possible that io_uring will outperform UB consistently on newer Linux.

File access on NVMe. We also test reading a file in NVMe disk by 1KiB block size, and show the comparison in Table 3 ("w/o sum"). We only consider the physical machine setting, because when VM accesses a file in a virtual NVMe disk, the file will be automatically cached into memory a priori, which behaves similarly to in-memory file access. The IOPS can be increased from $779_{\pm 5}$K to $852_{\pm 3}$K, yielding $9.4\%_{\pm 0.3\%}$ acceleration (KPTI on). When KPTI is off, the baseline increases to $810_{\pm 22}$K while UB increases slightly to $858_{\pm 10}$K, making the acceleration ratio smaller.

We also consider the situation that an I/O thread conducts lightweight calculation, like parsing packets. When the computations between consecutive I/O requests have dependency, the requests cannot be batched. Specifically, we set the I/O thread to calculate the sum of the buffer by treating it as a 64-bit integer array, after retrieving the buffer from kernel.

	w/o sum	**w/ sum**
KPTI on	$779_{\pm 5}$ ($852_{\pm 3}$)	$630_{\pm 4}$ ($793_{\pm 3}$)
KPTI off	$810_{\pm 22}$ ($858_{\pm 10}$)	$686_{\pm 65}$($795_{\pm 6}$)

Table 3: KIOPS of reading file on NVMe disk (1KiB size) on physical machine (w/o sum), and reading together with integer summation (w/ sum). The UB accelerated number is shown in the bracket.

As shown in Table 3 ("w/ sum"), even the lightweight computation could reduce considerable amount of IOPS. The baseline IOPS drops by 149K, while it only drops by 59K when UB is on, as such lightweight calculations in userspace can be entirely ported into kernel for execution, so their IPC is less affected by syscalls.

6.2 Redis

We choose a popular key-value store engine Redis as one macro-benchmark to test how UB handles real-world workloads. We evaluate Redis 6.2.6 with the built-in Redis-benchmark tool [39] to generate workload. We run the Redis server with its default configuration and launch the Redis-benchmark with 2 threads. The connection number is kept at the default value 50. In each round, the client issues 1M requests.

By default, Redis completes most of its work within the main thread, which is responsible for not only I/O but also computation tasks like hashing. For a normal workflow, which is also described in [30], the main thread invokes EPOLL to get a list of readable sockets. For each readable socket, the thread READs the socket and then processes the request. As a result, the userspace paths following READs are long (from 3k to 20k), as the computation tasks happen there. At last, Redis WRITEs responses to corresponding sockets one by one, with a small number of instructions in between (around 300).

Results. Figure 7 shows RPS with and without UB for GET and SET data of sizes ranging from 1B to 16KiB. When tested in VM with KPTI on, for GET, the acceleration ratio ranges from $4.4\%_{\pm 1.52\%}$ to $10.8\%_{\pm 2.69\%}$, when the data size is less or equal than 4KiB. The ratio drops to $-3.7\%_{\pm 0.51\%}$, when the size rises to 16KiB. Turning KPTI off drops the acceleration ratio to between $-2.0\%_{\pm 1.32\%}$ and $4.6\%_{\pm 1.96\%}$. The negative acceleration ratio suggests the overhead brought by UB outweighs the syscall overhead saved by itself. Executing on physical machine observes a different range: $-5.4\%_{\pm 1.17\%}$ to $6.4\%_{\pm 2.01\%}$ for KPTI on and $-6.4\%_{\pm 3.02\%}$ to $3.9\%_{\pm 1.67\%}$ for KPTI off. Noticeably, the RPS of Redis is much smaller

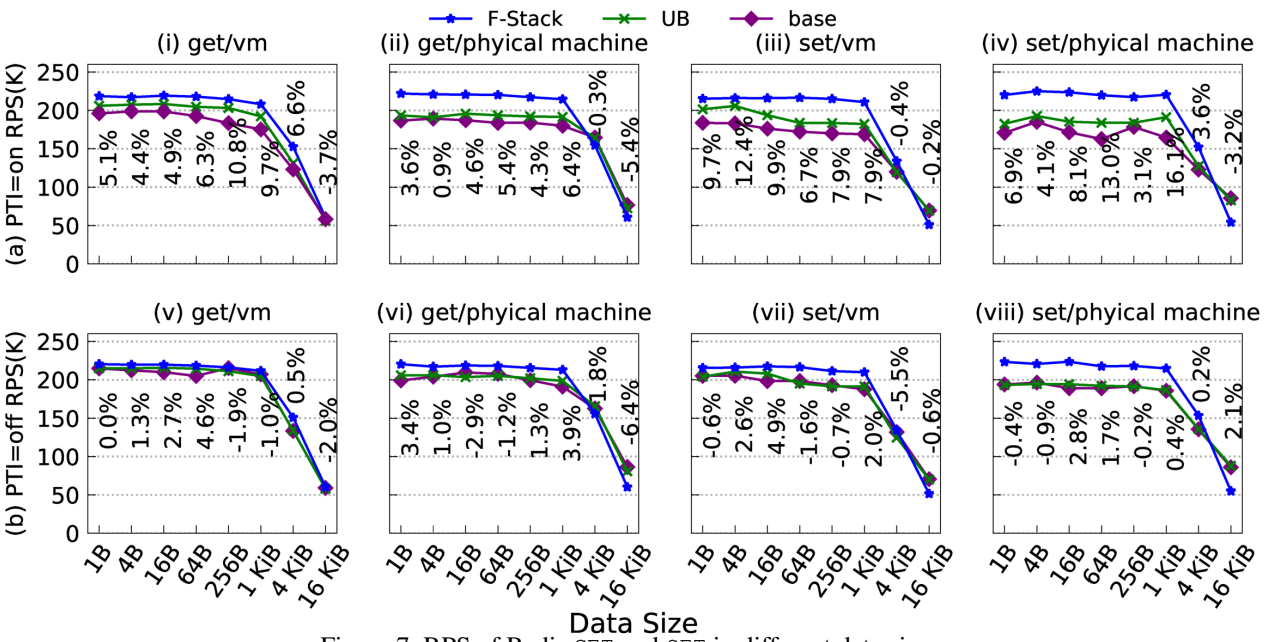

Figure 7: RPS of Redis GET and SET in different data sizes.

than that in our I/O micro-benchmark, so the expense from syscall is not the dominant factor. As a result, the acceleration ratio is much smaller.

Regarding SET, the acceleration ratio ranges from $-0.4\%_{\pm 2.19\%}$ to $12.4\%_{\pm 3.96\%}$ in VM with KPTI on. Similar trend is observed when KPTI is turned off and running in physical machine. Noticeably, Redis RPS drops significantly over 1KiB data size for both SET and GET, and similar observation was reported in the official documentation of the Redis benchmark [39].

Surprisingly, we found the RPS on VM is often higher than physical machine, though the virtual setup is supposed to yield lower RPS. We do not have a good explanation for why the opposite happened for Redis.

	Redis Server	BTC	User Space	Do Write
w/o UB	108.57	–	34.29	36.73
w/ UB	102.98	2.42	28.38	36.07

Table 4: Time spent on each part of Redis (VM+KPTI, SET).

Profiling performance gain. We let the BTC runtime profile BTC execution and userspace execution using the RDTSCP instruction. We run 20M Redis SET transactions (about 100 seconds) with and without UB. Table 4 shows the results. As we can see, by elevating the fast path to BTC, 5.91s userspace time can be saved, while the BTC only costs 2.42s. The difference (3.49s) can be attributed to the userspace IPC increase (indirect overhead). 5.59s are saved in total (the "Redis Server" column) and 2.1s (i.e., 5.59s - 3.49s) are saved directly by invoking fewer syscalls.

Overhead of memory checks. When strong kernel memory safety is unnecessary, e.g., when the binary is formally verified, a user may choose to chase higher performance gain by removing the instructions inserted to check memory boundary (i.e., SHL and SHR). We evaluate how much RPS gain can we get if we ask the translator not to insert such instructions. The results show that only 0.4% more RPS can be gained.

Comparison with DPDK. We compare the acceleration ratio of UB on Redis with that on DPDK as there are open-source implementations to empower Redis, like Redis-DPDK [2] and F-Stack Redis [45]. We chose F-Stack as the maintenance of Redis-DPDK has stopped since 2017 and it cannot run on the latest CPUs. F-stack supports the recent Redis 6.2.6 [46] as well as the recent DPDK 20.11. The comparison result is also shown in Figure 7.

It turns out F-Stack provides higher acceleration ratios for small size consistently (no larger than 4KiB). Interestingly, we found for 16KiB, F-Stack performs worse than UB and Redis baseline. One potential explanation is that F-Stack does not benefit from our multi-core setting. When we measure the CPU usage, it is always 100% for F-Stack, but UB and baseline can go up to 124%, which means multiple cores are used. Hence, F-Stack might outperform UB consistently when we restrict the core number to 1.

6.3 Nginx

In addition to Redis, we use Nginx (Version 1.20.0), a popular static web server with high RPS, as another macro-benchmark. Table 5 shows the number of instructions in the path followed by each syscall. These followed by less than 1,000 instruc-

syscall	recvfrom	openat	fstat	setsockopt	writev	sendfile	close	setsockopt
#Instructions followed	4,328	38	4,412	43	177	541	477	509

Table 5: Number of instructions following each syscall of Nginx. Those followed by less than 1,000 instructions are hot. The two setsockopt calls are different.

tions can be regarded as hot. Therefore, 6 out of the 8 can be accelerated. We run wrk [50] (Version 4.1.0, with 8 threads and 1024 connections), an HTTP benchmark tool, on the client machine to issue requests to the Nginx server for 12s to examine how much RPS Nginx can handle.

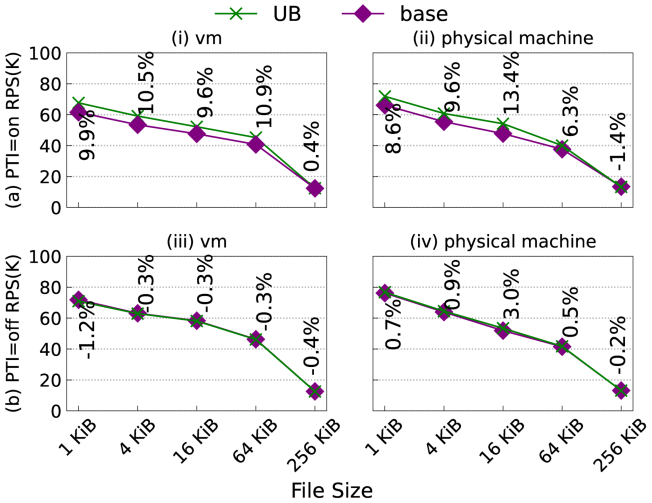

Figure 8: RPS of Nginx against different file sizes (Bytes).

Results. We gradually increase the file size requested by wrk and Figure 8 shows the RPS before and after UB acceleration. When being tested in VM with KPTI on, Nginx can be accelerated by $9.6\%_{\pm 1.81\%}$ to $10.9\%_{\pm 0.22\%}$ for 1KB to 64KB files, but the ratio drops to $0.4\%_{\pm 0.86\%}$ for 256KB file. For physical machine, the acceleration ratio ranges from $6.3\%_{\pm 0.17\%}$ to $13.4\%_{\pm 3.32\%}$ for 1KB to 64KB files, but also drops to $-1.4\%_{\pm 0.28\%}$ for 256KB file. These results show the bottleneck shifts from syscall to I/O for large files. When turning off KPTI, UB does not yield noticeable acceleration.

Multiple worker threads. We evaluate how multi-threading affects the acceleration ratio. We gradually increase the number of worker threads of Nginx and evaluate the case of 4KB file size. Figure 9 shows the RPS. As we can see, with more worker threads, the acceleration ratio drops noticeably when KPTI is on (from $8.6\%_{\pm 0.22\%}$ to $7.1\%_{\pm 0.17\%}$ for VM and $4.7\%_{\pm 0.15\%}$ to $2.0\%_{\pm 0.26\%}$ for physical machine), as the worker threads are increased from 2 to 8. When there are more worker threads, more cycles are used for thread synchronization, so fewer requests can be served per thread, reducing the syscall overhead saved by UB.

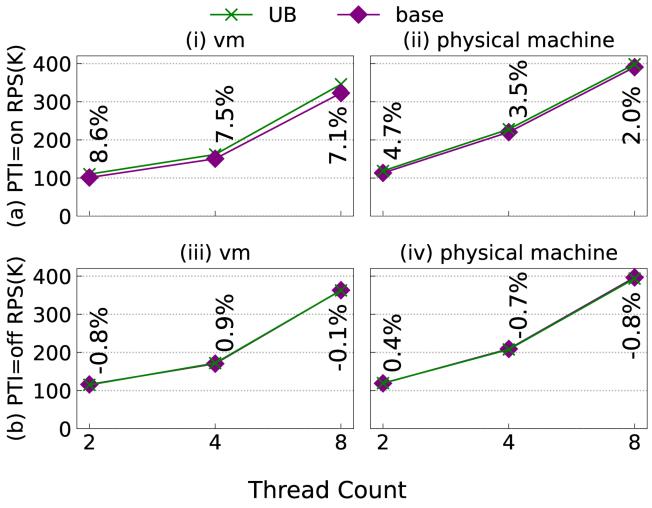

Figure 9: RPS of Nginx against different # of threads.

6.4 Raw Socket vs. eBPF

To avoid the syscall overhead, eBPF is another popular solution as described in Section 2.2. We show that, with the help of UB, developers can simply write the processing logic entirely in userspace with *raw socket*, and compare Packets Per Second (PPS) with eBPF.

We run a program on the client machine to send UDP packets to the server, and the server handles the incoming packets by either raw socket or XDP (eBPF library for packet processing) for 12s in each round. The client runs 15 threads, which can saturate the server. The processing tasks include counting the number of packets and summing packets by treating a packet as an integer array.

Results. Figure 10 shows the results by 3 packet sizes (128B, 512B, and 1472B). For VM with KPTI on, eBPF outperforms raw socket for small packets by up to $368.4\%_{\pm 8.92\%}$. For packets of MTU size (i.e., 1472B), eBPF still has $236.7\%_{\pm 4.15\%}$ more PPS. UB accelerates raw socket by $31.5\%_{\pm 0.25\%}$ – $34.3\%_{\pm 0.72\%}$, which are much smaller than eBPF. The PPSs for raw socket are similar across different packet sizes. However, eBPF is very sensitive to packet size, and we believe it is because the bottleneck of raw socket is protocol stack processing, which is bypassed by eBPF whose bottleneck may be the data movement, whose time consumption is related to packet size. When KPTI is off, the acceleration ratio of UB drops to $14.5\%_{\pm 0.45\%}$ – $17.8\%_{\pm 0.44\%}$ for various packet sizes. On physical machine, the acceleration ratios of UB

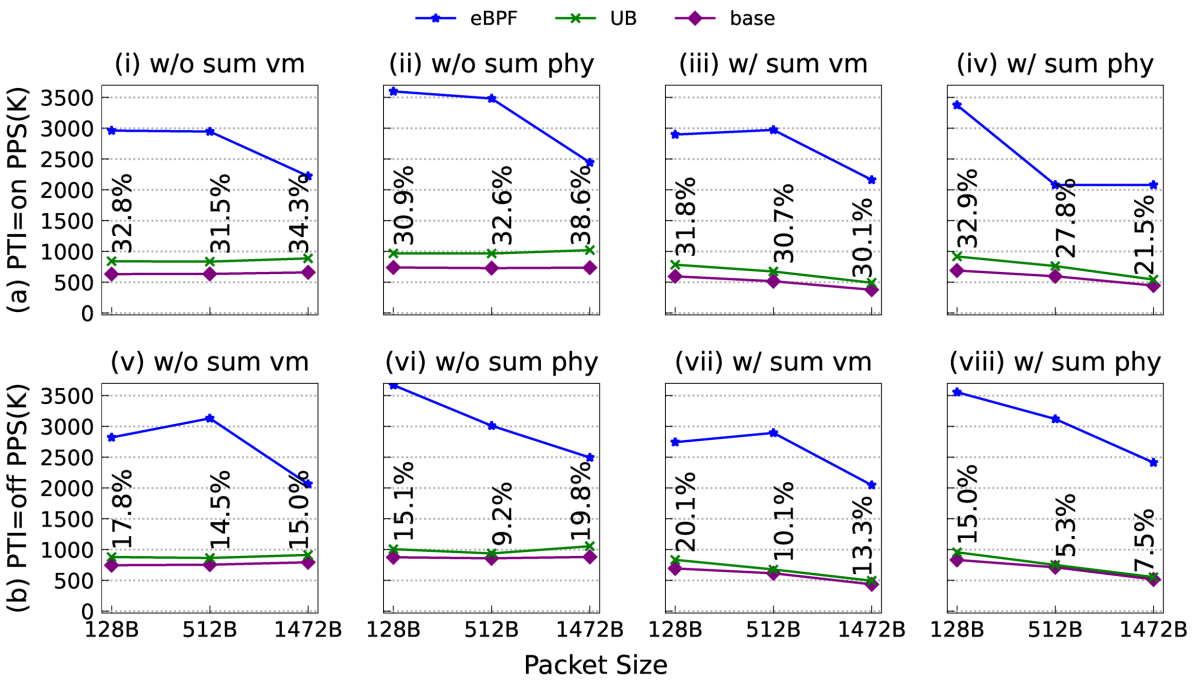

Figure 10: PPS of server handling incoming UDP packets in different packet sizes.

have larger ranges ($30.9\%_{\pm 0.87\%} - 38.6\%_{\pm 0.56\%}$ for KPTI on and $9.2\%_{\pm 0.14\%} - 19.8\%_{\pm 0.31\%}$ for KPTI off).

Computation. We also consider adding lightweight computation workload, i.e., packet summing, like the experiments for NVMe file access (Section 6.1). In VM, the PPS of raw socket sees greater drop when the packet size increases, but UB can still accelerates raw socket in similar ratios ($30.1\%_{\pm 0.20\%} - 31.8\%_{\pm 0.50\%}$ for KPTI on and $10.1\%_{\pm 0.18\%} - 20.1\%_{\pm 0.15\%}$ for KPTI off). eBPF is able to keep the similar PPS without packet summing. On the physical machine, similar trend is observed for raw socket, UB, and eBPF, except that eBPF sees considerable drop of PPS for 512B packet size and KPTI on.

Profiling execution performance. We profile the execution time of BTC and eBPF respectively for the case of packet summing with `RDTSCP`, like our experiments on Redis (Section 6.2). In VM with KPTI on, for handling 33.85M incoming packets of 128B, BTC spent 5.86s. In contrast, eBPF costs 9s. As we can see, the execution performance of BTC is better than eBPF VM. However, UB still cannot achieve similar PPS to eBPF based on the previous results. According to our analysis, the reason is that eBPF runs in softirq, so the packets can be dispatched into different cores. In contrast, the raw socket protocol stack has in-kernel locks for concurrent access. In particular, we added more threads for socket read, but did not see PPS increase at all. We also tried to assess how eBPF works without multi-threading, by restricting the IRQ of the NIC to a single core and repeating the sum experiment in VM with KPTI on. UB-accelerated socket reaches 1M, 0.96M and 0.93M PPS for the three packet sizes, while eBPF reaches 0.96M, 0.93M and 0.91M PPS respectively. Therefore, we believe the PPS of raw socket can be significantly improved if kernel optimizes its protocol stack for concurrent access. One potential approach is to build a better UB runtime so more deeper kernel trace points can be exposed via syscall, and we leave this as a future work.

7 Discussion

7.1 UB vs. eBPF

In addition to the comparison on the performance between UB and eBPF, here we compare their restrictions and security guarantees. As eBPF is developed mostly for packet processing and kernel tracing, it has a number of restrictions on the application code. For example, eBPF is not Turing Complete, as infinite loops are not allowed [33]. Due to its extensive restrictions on code, the eBPF verifier is prone to produce false positives, i.e., legal code regarded as illegal [14]. UB does not add any restrictions to developers and translates the userspace code transparently.

Regarding performance, UB only accelerates the paths following syscalls, but eBPF can be attached to many tracing points inside kernel, which makes it more flexible and capable of overcoming kernel bottlenecks. We believe UB could realize similar performance as eBPF, if kernel exposes more tracing points via syscalls.

In terms of security, eBPF relies on the isolation from in-kernel VM, while UB relies on the policies of SFI translator. Attacks targeting eBPF might be effective against UB as well, as described in Section 7.2. Formally verifying the implementation of eBPF and UB could mitigate these issues, but verifying eBPF is likely easier than UB, because eBPF has an official specification and it uses a reduced set of instructions.

7.2 Security Risks

Though we follow the SFI principles to design UB, new security risks could be introduced. First, UB might be vulnerable under side-channel attacks, which infer the secrets according to micro-architectural state changes. For instance, the Spectre attack has demonstrated that eBPF can be exploited to steal kernel memory, as eBPF VM compiles userspace code into kernel code [25]. The BTC of UB may also be exploited for similar attacks. To mitigate such risk, defenses against speculation attacks should be considered, e.g., placing speculation blocking instructions by the compiler [25]. Second, our BTC translator might not be able to sanitize privileged undocumented X86 instructions. To mitigate the introduced risk, the translator could allow a whitelist of instructions. When instructions outside the whitelist are encountered, UB should give up elevating their fast path. Third, previous research showed kernel races can lead to time-of-check to time-of-use (TOCTOU) attacks [28]. Since the BTC runtime does not enforce atomicity between the checking point and the use point for the fast path, the malicious userspace code can exploit kernel races. The mitigation can rely on the existing defenses that detect kernel races actively [20].

7.3 Other Limitations

Admittedly, kernel-bypass frameworks like DPDK could achieve better performance than UB, when the developers take the right measures to integrate them into the userspace applications. The better performance not only comes from the reduction of context switching overhead, but also the simplified and more efficient userspace drivers. For example, userspace drivers could avoid unnecessary buffer copying, interrupt, etc. In contrast, UB only reduces the context switching overhead. The key advantage of UB is that it does not require any change on the applications by the developers (see Table 1). Therefore, we believe kernel bypass would be favored when the developers are willing to refactor their code or design a new application with kernel bypass in mind.

UB does not aim to replace asynchronous I/O. Admittedly, when an application is both computation-intensive and I/O intensive, asynchronous I/O helps the developers decouple I/O from computation in different threads, making better use of multi-cores. UB does not give synchronous I/O tasks more IOPS than asynchronous tasks, but it can be used jointly with asynchronous I/O. In some cases, the I/O threads of asynchronous tasks still intensively invoke syscalls to submit I/O and UB can accelerate these tasks.

8 Related Work

Section 2 has surveyed related works about syscall optimization. Below we describe other related works.

Dynmaic Binary Translation (DBT). DBT is a powerful method for debugging and instrumentation [3, 19, 22, 48]. Kedia et al. proposed a fast DBT in kernel to instrument kernel code [22]. Our translator has some similarities with theirs in indirect branch processing, but our translator differs largely in memory protection and register renaming. Besides, some functionalities of their runtime require rollback. In contrast, our runtime never rolls back.

Software-Based Fault Isolation (SFI). Enforcing SFI in kernel is not an entirely new idea. XFI was firstly proposed to isolate kernel modules with SFI, and later LXFI added kernel API check to restrict the fault propagated via kernel APIs [13, 32]. UB uses SFI in a different way for the fast path.

Accelerating Inter-Process Communication (IPC). Some schemes were proposed recently to exploit hardware assistance to accelerate IPC. Similar to accelerating system calls, they also try to minimize context switching overhead. Gu et al. proposes to accelerate IPC with the help of recent innovation in Intel processors, i.e., MPK [16]. Mi et al. borrows a hardware function designed for virtualization to accelerate IPC [35]. Du et al. proposes to add new features to CPU for context switching without involving kernel [12]. They implemented the prototype on RISC-V FPGA processors.

9 Conclusion

The overhead brought by syscalls is prominent to high-IOPS applications, but the existing approaches have not completely addressed this issue, because they require efforts in code refactoring. To preserve binary compatibility, we propose userspace bypass (UB) that executes userspace instructions directly in kernel. UB employs a JIT translator that translates userspace instructions between syscalls into sanitized code blocks. The code blocks are constrained to avoid introducing extra harm, therefore they can be executed directly in kernel. With UB, I/O micro-benchmark can be accelerated by 30.3 – 88.3% and real-world applications like Redis can be accelerated by 4.4 – 10.8% for 1B – 4KiB data sizes under `GET`, when the applications are executed in VM with KPTI on.

Acknowledgement

We thank our shepherd Dan Tsafrir for his highly valuable suggestions. The Fudan authors are sponsored by National Key R&D Program of China (Grant No. 2022YFB3102901) and Natural Science Foundation of Shanghai (No. 23ZR1407100).

References

[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/ansyun/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 81–85, 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/2244085/1-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.medium.com/improving-redis-performance-through-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://github.com/GlareR/UserspaceBypass. Accessed: 2022-09-25.

[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, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded 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 Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 101–115, 2013.

[23] Kernel.dk. Efficient IO with io_uring. https://kernel.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/674. 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/latest/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 in-flight 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.

A Artifact Appendix

Abstract

Our artifact includes the source code of UB, and the apps we used for evaluation. The readers can follow the instructions to modify the Linux kernel to support UB, compile UB to run on it, and evaluate the apps on it.

Scope

The IOPS of all the apps we evaluated can be reproduced. Specifically, Figure 5, Figure 7, 8, 9 and 10. Reproducing the I/O benchmark is the most convenient case. Therefore, it is recommended to start from Figure 5.

The whole experiment can be time-consuming, so people may take fewer repeat rounds to save time.

Content

The artifact includes the implementation of UB, which consists of the three files to be modified over Linux Kernel (zz_lkm, zz_daemon, and zz_disassem). zz_lkm is the kernel part of UB, which profiles processes and executes the BTC. zz_daemon sits at userspace to communicate with the kernel module and invoke zz_disassem to do the actual translation.

Hosting

The source codes are hosted at https://github.com/glarer/UserspaceBypass, as well as the readme file.

Requirement

The I/O benchmark experiment requires only a server machine. Because Redis, Nginx, and raw socket experiments involve network, another client machine is required to be connected to the server.

The IOPS is highly related to CPU performance. Therefore, the reproduced IOPS values may be different by different CPUs, but we can always see the performance gain.

The IOPS can also be disturbed by network performance. If the NIC used is not sufficiently powerful, the IOPS may drop for large I/O size, as well as the performance gain.