Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication

Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc20/presentation/gu

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication

Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University

Abstract

This paper presents UnderBridge, a redesign of traditional microkernel OSes to harmonize the tension between messaging performance and isolation. UnderBridge moves the OS components of a microkernel between user space and kernel space at runtime while enforcing consistent isolation. It retrofits Intel Memory Protection Key for Userspace (PKU) in kernel space to achieve such isolation efficiently and design a fast IPC mechanism across those OS components. Thanks to PKU's extremely low overhead, the inter-process communication (IPC) roundtrip cost in UnderBridge can be as low as 109 cycles. We have designed and implemented a new microkernel called ChCore based on UnderBridge and have also ported UnderBridge to three mainstream microkernels, i.e., seL4, Google Zircon, and Fiasco.OC. Evaluations show that UnderBridge speeds up the IPC by 3.0× compared with the state-of-the-art (e.g., SkyBridge) and improves the performance of IPC-intensive applications by up to 13.1× for the above three microkernels.

1 Introduction

The microkernel OS design has been studied for decades [3, 27, 35, 44, 49, 52, 70]. Microkernels minimize code running in supervisor mode by moving OS components, such as file systems and the network stack, as well as device drivers, into isolated user processes, which achieves good extensibility, security, and fault isolation. Other than the success of microkernels in safety-critical scenarios [1, 40, 66], there is a resurgent interest in designing microkernels for more general-purpose applications, such as Google's next-generation kernel Zircon [3].

However, a cost coming with microkernel is its commonly lower performance compared with its monolithic counterparts, which forces a tradeoff between performance and isolation in many cases. One key factor of such cost is the communication (IPC) overhead between OS components, which is considered as the Achilles' Heel of microkernels [30, 33, 51, 53, 62, 76]. Hence, there has been a long line of research work to improve the IPC performance for microkernels [19, 30, 36, 44, 50, 52, 62, 79, 82]. Through a combination of various optimizations such as in-register parameter passing and scheduling avoidance, the performance of highly optimized IPC has reached less than 1500 cycles per roundtrip [13]. The state-of-the-art SkyBridge IPC design, which retrofits Intel *vmfunc* to optimize IPCs, has further reduced the IPC cost to around 400 cycles per roundtrip [62]. However, such cost is still considerable compared with the cost of invoking kernel components in monolithic kernels (e.g., calling function pointers takes around 24 cycles).

There is always a tension between isolation and performance for OS kernel designs. In this paper, we present a new design named UnderBridge, which redesigns the runtime structure of microkernels to harmonize performance and isolation. The key idea is building isolated domains in supervisor mode while providing efficient cross-domain interactions, and enabling user-space system servers[1] of a microkernel OS to run in those domains. A traditional microkernel OS usually consists of a core kernel in supervisor mode and several system servers in different user processes. With UnderBridge, a system server can also run in an isolated kernel space domain besides a user process. The system servers that run in kernel can interact with each other as well as the core kernel efficiently without traditional expensive IPCs, and applications can invoke them with only two privilege switches, similar to a monolithic OS. Although the number of isolated domains is limited and may be smaller than the number of system servers, UnderBridge supports server migration. The microkernel can dynamically decide to run a server either in a user process or a kernel domain based on how performance-critical it is. However, it is challenging to efficiently provide mutually-isolated domains together with fast cross-domain interactions in kernel space.

Protection Keys for Userspace (PKU [7], Section-2.7, Volume-3), also named as Intel memory protection keys (MPK), has been introduced in recent Intel processors and investigated by researchers to achieve intra-process isolation in user space [29, 39, 64, 77]. As the name "Userspace" indicates, PKU is a mechanism that appears to be only effec-

[1] We name the microkernel OS components implementing system functionalities as system servers. File system and drivers are typical examples.

tive in user space. After a detailed investigation, we observed that *no matter in kernel space (Ring-0) or user space (Ring-3), PKU-capable CPUs transparently enforce permission checks on memory accesses only if the User/Kernel (U/K) bit of the corresponding page table entry is User (means user-accessible)*. Hence, PKU, as a lightweight hardware feature, also offers an opportunity to achieve efficient intra-kernel isolation if all the page table entries for kernel memory are marked with U bit instead of K bit. However, marking kernel memory as user-accessible is dangerous since unprivileged applications may directly access kernel memory. Fortunately, today's OS kernels are usually equipped with kernel-page-table-isolation (KPTI) when preferring stronger security guarantees, including defending against Meltdown-like attacks [20, 55] and protecting kernel-address-space-layout-randomization [8, 42]. User processes and the kernel use different page tables with KPTI, so marking kernel memory as user-accessible in a separate page table does not risk allowing applications to access kernel space.

Hence, UnderBridge allocates an individual page table for the kernel and builds isolated *execution domains* atop MPK memory domains in kernel space. Unlike software fault isolation (SFI), guaranteeing memory isolation with MPK hardware incurs nearly *zero* runtime overhead. Meanwhile, a new instruction, *wrpkru*, can help switching domains by writing a specific register, PKRU (protection key rights register for user pages), which only takes 28 cycles. Thus, domain switches can be quick. With UnderBridge, we design and implement a prototype microkernel named ChCore, which comprises a core kernel and different system servers similar to existing microkernels. ChCore still preserves the IPC interfaces with fast domain switch for the servers but embraces better performance by significantly reducing the IPC costs.

However, we find merely using MPK in kernel fails to achieve the same isolation guarantee as traditional microkernels. On the one hand, since MPK only checks read/write permissions when accessing memory but applies no restrictions on instruction fetching, any server can execute all the code in the same virtual address space. Thus, an IPC gate (a code piece), which is used in UnderBridge to establish the connection between two specific servers, can be abused by any server to issue an illegal IPC. To address the problem, UnderBridge authenticates the caller of an IPC gate through checking its (unique) memory-access permission and ensures the authentication is non-bypassable by validating a secret token at both sides of the IPC gate.

On the other hand, system servers running in kernel space can execute privileged instructions. Although the servers (initially running in user space) should not contain any privileged instruction, such instructions may arise inadvertently on x86, e.g., being composed of the bytes of adjacent instructions. Thus, a compromised server may use return-oriented programming (ROP) [21, 69] to execute them. Traditional microkernels confine system servers in user processes, which, inherently, prevent them from executing privileged instructions and exclude them from the system's trusted computing base (TCB). To do not bloat the TCB, we also prevent the in-kernel system servers from executing any privileged instruction. We leverage hardware virtualization and run ChCore in non-root mode and deploy a tiny secure monitor in root mode. For most privileged instructions that are *not* in the critical path, we configure them to trigger *VMExit* and enforce permission checks in the monitor. For others, we carefully handle them using binary scanning and rewriting.

We have implemented ChCore on a real server with Intel Xeon Gold 6138 CPUs and conducted evaluations to show the efficiency of UnderBridge. To demonstrate the generality of UnderBridge, we have also ported it to three popular microkernels, i.e., seL4 [14], Google Zircon [4], and Fiasco.OC [2]. In the micro-benchmark, UnderBridge achieves 3.0× speedup compared with SkyBridge. In IPC-intensive application benchmarks, UnderBridge also shows better performance than SkyBridge (up to 65%) and improves the performance of the above three microkernels by 2.5×∼13.1×.

In summary, this paper makes the following contributions:

- A new IPC design called UnderBridge that retrofits Intel MPK/PKU in kernel to achieve ultra-low overhead interactions across system servers.
- A microkernel prototype ChCore which uses UnderBridge to move system servers back to kernel space while keeping the same isolation properties as traditional microkernels.
- A detailed evaluation of UnderBridge in ChCore and three widely-used microkernels, which demonstrates the efficiency of the design.

2 Motivation

2.1 Invoking Servers with IPCs is Costly

To obviate the severe consequences incurred by crashes or security breaches [9, 22, 24], the microkernel architecture places most kernel functionalities into different user-space system servers and only keeps crucial functionalities in the privileged kernel, as depicted in Figure-1. Therefore, a fault in a single system server can be caught before it propagates to the whole system.

However, compared with a monolithic OS like Linux, a system service invocation usually becomes more expensive

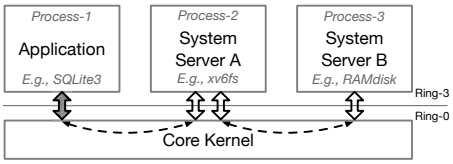
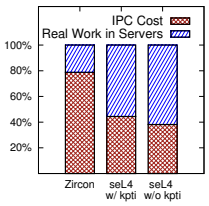


Figure 1: A simplified microkernel architecture. Even without KPTI, calling a server with IPC requires two user-kernel roundtrips with two process switches. A vertical arrow represents one roundtrip, and a dotted line means two process switches.

in such an OS architecture. Figure-1 shows a service invocation procedure that involves two system servers and thus leads to two IPCs. In this case, a microkernel requires *four* roundtrips between user and kernel in total, while Linux only requires *one* (i.e., the leftmost arrow between the application and the OS). It is because Linux invokes different kernel components (like system servers in microkernel) directly through function calls.



(a)

Parts (cycles)	**w/o KPTI**	**w/ KPTI**
Privilege Switch	158	690
Process Switch	295	Included above
Others	277	320
Total	**730** *	**1010**

*The result conforms to the officially reported data (722∼736) [13].

(b)

Figure 2: (a) Invoking servers with IPCs is expensive. (b) A breakdown of seL4 fast-path IPC.

To measure the performance cost of server invocations with IPC, we run SQLite3 [15] on Zircon and seL4 (§ 6 gives the detailed setup). We measure (i) the total time spent on invoking system servers (i.e., an FS server and a RAMdisk server) and (ii) the effective time used in system servers for handling the requests. The difference between the two time durations is considered as the IPC cost. As presented in Figure 2(a), the IPC cost is as high as 79% in Zircon. Even in seL4, which uses highly-optimized IPCs, 44% of the time is spent on IPC when KPTI enabled (38% without KPTI).

Moreover, a system functionality may involve even more IPCs to invoke multiple system servers. For instance, launching an application requires 8 IPC roundtrips (among *Shell, Loader, FS, and Driver*) on Zircon. In contrast, it only needs one or two system calls on Linux (e.g., fork + exec). Therefore, invoking system servers with IPCs is time-consuming in microkernels, which motivates our work in this paper.

2.2 IPC Overhead Analysis

To further understand the overhead of IPC, we break down and measure the cost of each step in the IPC procedure in seL4, which is known to be an efficient implementation of microkernel IPC. Here we use the ideal configuration by referring to [13] and measure a one-way IPC (not a roundtrip) without transferring data.

We find that the direct cost of the IPC consists of three main parts as shown in Figure-2(b). The first part is the privilege switch. A user-space caller starts an IPC by using a *syscall* instruction to trap into the kernel. The kernel needs to save the caller's context, which will be restored when resuming the caller. To invoke the target user-space callee, the kernel transfers the control flow with a *sysret* instruction after restoring the callee's context. The second part is the process switch. The major cost in this part is the *CR3* modification instruction (270 cycles). Since the caller and callee are isolated in different user-space processes (different address spaces), the kernel has to change the address space from the caller to the callee. With KPTI enabled, the kernel further needs to change the address spaces twice during the privilege switch, which inflates the overhead. The third part is other logics in IPC, such as permissions and fast-path conditions checks.

Besides its inherent cost, an IPC will inevitably cause pollution to the CPU internal structures such as the pipeline, instruction, data caches, and translation look-aside buffers (TLB), which has been evaluated in prior work [18, 32, 62, 71]. According to [71], the pollution caused by frequent privilege switches can degrade the performance by up to 65% for SPEC CPU programs.

In summary, the switches of privilege and address space in IPC bring considerable overhead, which motivates our lightweight IPC design to remove these switches.

2.3 Using Intel MPK in Kernel

Background: Intel memory protection keys (MPK) [7] is a new hardware feature to restrict memory accesses. MPK assigns a four-bit domain ID (aka, protection key) to each page in a virtual address space by filling the ID in previously unused bits of page table entries. Thus, MPK can partition user pages within a virtual address space into at most 16 memory domains. To determine the access permissions (read-only, read-write, none) on each memory domain, it introduces a per-core register, *PKRU*, and a new instruction, *wrpkru*, to modify the *PKRU* register in only 28 cycles. It is worth mentioning that MPK checks on memory accesses incur nearly *zero* runtime overhead [39, 77]. Nevertheless, MPK does not enforce permission checks on execution permission. One executable memory page is always executable to any domain, even if *PKRU* forbids the domain from reading the page.

Observation: After a detailed investigation, we observe that no matter in Ring-0 or Ring-3, MPK enforces permission checks on any user-accessible memory page. To enable MPK checks in Ring-0, the *User/Kernel (U/K)* bits of all the corresponding page table entries in a four-level page table have to be set as *User* (i.e., 1). If there exists one entry that contains the *Kernel* bit at any level, MPK will not check the access on the corresponding memory pages. Furthermore, the Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) should also be disabled for accessing or executing these pages (tagged with *User*) in Ring-0.

2.4 Building Isolated Domains

There are many approaches to build lightweight and isolated domains. SFI (Software Fault Isolation), which has been actively studied over 20 years [34, 46, 60, 68, 80, 87], is one of the most mature candidates. However, although being a general solution to achieve memory/fault isolation, SFI incurs non-negligible runtime overhead due to excessive code instrumentations. For example, two representative studies

show that SFI introduces around 15% overhead for SPEC CPU programs on average (Table-2 in [68]), even with the help of the latest boundary-checking hardware MPX (Figure-3 in [46]). Some other approaches [48, 88] uses x86 segmentation for memory isolation, which avoids software checks on memory accesses and is suitable for sandbox execution, but it is not widely used anymore [68].

Instruction	Cost (cycles)
Indirect Call + Return	24
syscall + sysret	150
Write *CR3* (no TLB flush)	226
vmfunc (switch EPT)	146
wrpkru	28

Table 1: Cost comparison of selected instructions.

Leveraging advanced hardware features to build isolated domains can achieve better runtime performance. For example, prior work [39, 46, 56, 57, 62, 65, 77] utilizes (extended) page tables to provide isolated domains in user space and exploits instructions like *vmfunc* and *wrpkru* for fast domain switches. We list the costs of these frequently-used instructions in Table-1. The cost of *wrpkru* is the closest to *indirect call*, which is used to invoke kernel components (with function pointers) in monolithic kernels. Therefore, considering that MPK is applicable to the kernel and has good performance property, we propose that MPK can be leveraged to implement an efficient fine-grained isolation mechanism in the kernel.

3 UnderBridge

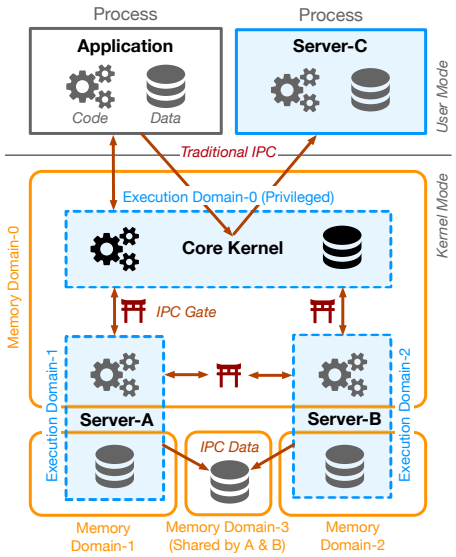


Figure 3: The overview of ChCore based on UnderBridge.

The goal of UnderBridge is to optimize the synchronous IPC[2] while simultaneously maintaining strong isolation. An intuitive design is adopting the MPK-based intra-process isolation [39, 77] to run system servers within an application address space. However, this design has three major problems. First, it requires to map a server into multiple applications' page tables, which makes updating the server's memory mappings especially expensive (i.e., update all the page tables). Second, its cost to setting up the IPC is non-negligible due to intensive page table modifications. Third, it restricts the applications' ability to use the whole address space and the MPK hardware freely.

Instead, UnderBridge boosts IPC performance by putting the system servers, which are user-space processes in traditional microkernels, back into kernel space. Figure-3 shows the system overview. The core kernel resembles the traditional microkernel, which provides crucial functionalities such as managing memory protection, capability enforcement, scheduling, and establishing IPC connections. With UnderBridge, system servers can run in kernel space (e.g., Server-A/B) but are confined in isolated environments (called *execution domain*). UnderBridge makes the core kernel and in-kernel system servers share the same (kernel) address space while leveraging Intel MPK to guarantee memory isolation.

To use MPK in kernel space, UnderBridge tags all the kernel memory pages with "User" bits in the kernel page table, as introduced in § 2.3. However, marking kernel memory as user-accessible enables unprivileged user-space applications to access kernel memory directly. UnderBridge prevents this by allocating a separate page table to each application. An application's page table does not contain the kernel space memory except for a small trampoline region (tagged with "Kernel"), which is used for privilege switch (e.g., *syscall*).

Building IPC connections with in-kernel system servers takes the following steps. First, a system server proactively registers a function address (IPC function) in the core kernel before serving requests. The core kernel will check whether the address is legal, i.e., both executable and belonging to the server. Second, another server can ask the core kernel to establish an IPC connection with the registered server. Third, the core kernel generates an *IPC Gate*, which helps to accomplish the IPC function invocation.

If an application needs to invoke the in-kernel server, it also needs to establish the connection first. Later, it invokes the system call for IPC and traps into the core kernel. Then, the kernel will help to invoke the requested server via the corresponding IPC gate (between the kernel and the server).

3.1 Execution Domains

As shown in Figure-3, UnderBridge constructs isolated *execution domains* over MPK *memory domains* and confines each in-kernel system server in an individual execution domain. Specifically, UnderBridge builds 16 execution domains in kernel space and assigns a unique domain ID (0∼15) to each of them. Execution domain 0 is specialized

[2]Synchronous IPC is commonly used in microkernels, especially when calling system servers. After issuing a synchronous IPC, the caller blocks until the callee returns.

for running the (trustworthy) core kernel and can access all the memory. Every other execution domain (1∼15) has a private memory domain with a specific ID and can only access its private memory domain by default. A system server, exclusively running in one execution domain, stores its data, stack, and heap regions in its private memory domain, which cannot be accessed by other servers. Nevertheless, its code region resides in memory domain 0 that can only be read/written by the core kernel. In this way, the server cannot read-/write its own code but can still execute it (i.e., execute-only memory) as MPK memory domains do not affect instruction fetching. UnderBridge ensures an execution domain can only access allowed memory domains by configuring its *PKRU* register. It also forbids an execution domain (a server) from modifying this register by itself (details in § 4.2).

Shared memory between two servers is allocated by allowing them to access a free memory domain together (e.g., Memory Domain-3). Shared memory between a server and the core kernel is achieved by letting the core kernel directly access the server's private memory domain. Shared memory between an application and a server is achieved by mapping some private memory of the server in the application's page table, which does not require a free memory domain.

3.2 IPC Gates

Even though system servers reside in the same kernel address space, UnderBridge still preserves the well-defined IPC interfaces for them. When connecting two system servers (in two execution domains), the core kernel generates an IPC gate for them, which resides in memory domain 0. Specifically, it first allocates a piece of memory for the gate and loads the gate code, which is small, as shown in Figure-4, into the memory. Then, it fills specific values (e.g., per-gate *SECRET_TOKEN*) into the gate. After that, it gives the gate address to the two system servers connected by this gate.

Later, during an IPC invocation, the gate transfers the control flow from a caller to a callee. To be more specific, it saves the caller's execution state, switches to the callee's domain by setting the *PRKU* register, and restores the callee's execution state. UnderBridge allows the caller and callee to define their calling conventions (the gates only save/restore necessary state by default), which is flexible and efficient. Transferring messages by registers and shared memory are both supported.

Since the system server in UnderBridge only executes when being called through an IPC gate, we adopt the mechanism of decoupling the execution context, which contains the execution state (e.g., register values), with the scheduling context, which contains the scheduling information (e.g., time slice used in the scheduler) [47, 58, 72] and mark those servers as passive. When an application (T1) invokes the system server (T2) through an IPC, T2 inherits T1's scheduling context and then starts executing. When T2 invokes another server (T3), T3 also inherits the scheduling context, which is originally from T1. Thus, the scheduling overhead is avoided in the IPC gates.

Besides intra-server, IPC gates also exist between the core kernel and the servers, which enables the core kernel to interact with the servers. Specifically, those in-kernel system servers invoke the core kernel's service through dedicated IPC gates connected with the core kernel instead of using *syscall* instructions. To handle exceptions/interrupts during the execution of in-kernel servers, UnderBridge provides similar gates at the beginning of each handler to switch the execution domain to the core kernel (execution domain 0).

3.3 Server Migration

As MPK provides 16 memory domains in total, UnderBridge can only support at most 16 execution domains concurrently, including the reserved one for the core kernel. Nevertheless, more system servers can be required, considering the number of different device drivers. When the number of concurrent system servers exceeds 15, one solution is time-multiplexing [64] but will bring non-negligible overhead if frequently stopping and restarting servers.

Instead, UnderBridge enables *server migration*, which dynamically moves servers between user and kernel space. Each server is compiled as a position-independent executable, and the core kernel assigns disjoint virtual memory regions for different system servers. Whenever runs in an execution domain in the kernel or a user process, a server always uses the same virtual addresses. So, when migrating a server between user and kernel, UnderBridge does not need to do relocations because all the memory references in the server are always valid (no changes), which significantly simplifies the migration procedure. Moreover, the system call layer of the LibC used by servers is also modified, and thus all the *syscall* instructions are organized in one memory page, namely the *syscall page*. When migrating a server from user space to kernel space, the core kernel overrides this page with another prepared page which contains IPC gates connected to the core kernel. Therefore, a server can seamlessly perform system calls no matter in user or kernel space.

Specifically, there are four steps to migrate a server from kernel space to user space. First, the core kernel will wait for the server to enter a quiescent state by blocking new IPC requests to the server temporarily and waiting for the finishes of on-going ones. Second, it will modify the *syscall page* of the server and making the server perform system calls through *syscall* instruction instead of IPC gates later. Third, it will free the execution domain (ID) of the server by setting the domain ID of the server's page table entries to 0 and flush TLBs. If the server installs shared memory with another in-kernel server (S), the domain ID of the shared memory will also be freed because the shared memory can use the domain ID of S in the kernel page table. Last, the core kernel will activate the in-user server and allow clients to issue IPCs to it. Migrating a server from user space to kernel space

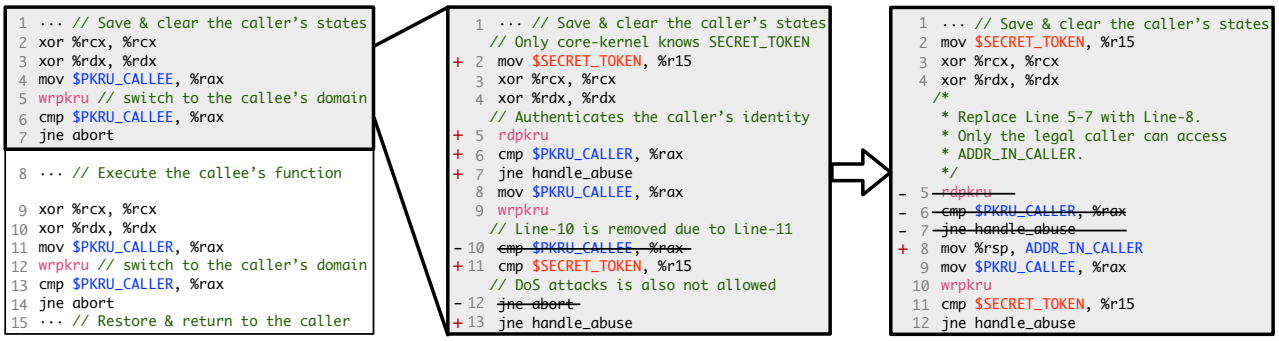


Figure 4: (a) A basic IPC gate for switching execution domains: Line 1-7 is from the caller to the callee and Line 9-15 is just a reverse process. (b) A security-enhanced IPC gate (from the caller to the callee only) that solves the arbitrary IPC problem. (c) An optimized IPC gate based on the secure one.

works similarly. The mappings of a system server in the kernel page table (used when running in the kernel) will not be removed, and the system server's page table (used when running in user) always exists. And the core kernel will keep corresponding mappings in the two page tables the same.

With server migration, UnderBridge can run frequently-used servers (according to either online or offline profiling) in kernel space while accommodating other servers in user processes. Besides that, we think the number of frequently-used servers may usually be small according to a preliminary survey on some popular applications including Memcached, MySQL, GCC, and a ROS-based (robot) application. We run those applications on Linux and find the most required system calls are only related to the File System, Network, Synchronization, and Memory-Management. For a microkernel, these system calls are usually implemented in only several system servers or directly in the core kernel, which indicates the server migration may rarely happen.

4 Enforcing Isolation in UnderBridge

Threat Model and Assumptions. UnderBridge aims to achieve the same security guarantee as existing microkernels and inherits the same trust model. Specifically, the (trusted) core kernel is assumed to be bug-free and correctly implemented because it has relatively small codebase (e.g., 8,500 LoC in our implementation) and is amenable to formal verification [44]. We do not trust applications or the operating system servers, which may have vulnerabilities or even be maliciously crafted and can be fully compromised by attackers. Physical attacks and hardware bugs are out of the scope of this paper.

Two Security Challenges. Although UnderBridge achieves memory isolation by utilizing MPK hardware, there are still two security threats.

The first threat is the arbitrary IPC problem. The core kernel generates IPC gates in memory domain 0 to ensure any server cannot modify the gates. However, an IPC gate can still be invoked by any (in-kernel) system server as the MPK does not enforce permission check on *execution permission*. Although recent work on MPK-based intra-process isolation [39, 77] does not consider such gate abusing problem, we cannot neglect it because it violates the enforcement of the IPC capability in microkernels. Therefore, UnderBridge ensures only a legal caller (allowed by the core kernel) can successfully use an IPC gate by adding mandatory authentications in the gate. § 4.1 explains the secure design of IPC gates.

The second threat is that untrusted in-kernel system servers run in supervisor mode (Ring-0). Thus, a compromised server can execute any privileged instructions theoretically, which threatens the whole system. For example, it could install a new page table and freely access all memory. One possible defense solution is to enforce control-flow integrity (CFI), which ensures that servers cannot execute any illegal control flow leading to executing privileged instructions. However, CFI instrumentations inevitably bring obvious runtime overhead. Instead, we choose hardware virtualization technology and run ChCore in non-root mode. A tiny secure monitor in root mode audits the execution of most privileged instructions, as summarized in Table 2, by simply trapping them through *VMExits*. In the meanwhile, we use the binary rewriting technique to avoid the expensive *VMExits* on the critical paths. § 4.2 gives more details.

4.1 Unauthorized IPCs Prevention

As shown in Figure-4(a), the responsibility of an IPC gate is saving/restoring the (necessary) execution context of the caller/callee and switching the execution domain from the caller to the callee. Line 2-4 prepares the argument registers for *wrpkru*, which requires *eax* to store the target permission, and both *ecx* and *edx* to be zero. The *wrpkru* instruction in Line 5 sets *PKRU_CALLEE* (the callee's permission) to the *PKRU* register, which specifies the memory-access permission, so the execution domain changes to the callee's domain after this instruction finishes. Line 6-7 prevents a compromised thread from directly jumping (e.g., ROP) to Line 5 with some carefully chosen value in *eax*. Existing work [39, 77] on MPK also designs similar gates for intra-

process isolation.

However, such a design faces the gate abusing problem. Since MPK has no restriction on execution permission, a domain can use the gates belonging to other domains to issue IPCs. UnderBridge solves this problem by authenticating the caller's identity in the IPC gates. As the core kernel is responsible for generating IPC gates when two execution domains establish the IPC connection, it knows which domain is the legal caller of the gate. Besides, since each domain has unique memory-access permission, UnderBridge regards the permission as the domain's identity and checks the identity with *rdpkru* in Line 5-7 of Figure-4(b). Moreover, UnderBridge must ensure that the identity check cannot be bypassed. Otherwise, a compromised thread can jump to Line 8/9 without going through the check. To this end, each gate adds two cheap instructions (Line 2 and Line 11 in Figure-4(b)) to guarantee that a successful IPC invocation must go through the check. When setting up an IPC gate, the core kernel randomly produces a 64-bit secret token with *rdrand* instruction and inserts it to the gate. Note that any server cannot read the token value since the IPC gates belong to domain 0 (core kernel). Thus, any caller who wants to pass the check at Line 11 must execute Line 2 first, which ensures the identity check at Line 5-7 is non-bypassable for successful invocations. Similarly, Line 8 is also non-bypassable for successful IPCs; thus, Line 10 is no more required.

Figure-4(c) further gives a more efficient design, which eliminates the overhead of identity check. Although *rdpkru* only takes about 9 cycles, it as well as the extra comparison (Line 6-7) are still on the critical path of IPC. To remove the overhead, UnderBridge authenticates IPC callers by *reusing* the stack-pointer saving instruction (Line 8), which is initially located in the procedure of state saving (Line 1). In this way, any illegal caller will trigger a fault when accessing *ADDR_IN_CALLER* (Line 8) and get caught.

4.2 Privilege Deprivation

In traditional microkernels, system servers only have Ring-3 privilege. To achieve the same security/isolation guarantee, UnderBridge should restrict the servers' behavior when running them in execution domains (Ring-0). However, a compromised server may find and execute privileged instructions at unaligned instruction boundaries with ROP to attack the whole system. Based on an in-depth analysis of privileged instructions (briefly summarized in Table-2), UnderBridge combines virtualization hardware support, binary rewriting technique, and some specific solutions to de-privilege the execution domains for servers with negligible overhead.

According to our survey on four microkernels, system registers such as *IDTR* are only configured at boot time; debug/profile registers are only accessed at debugging/profiling time; most model-specific registers (*MSRs*) are also not accessed in the critical paths. Thus, UnderBridge runs the microkernel in non-root mode and configures the privileged instructions operating on those registers to trigger *VMExits*. And, the secure monitor in root mode checks whether they are executed by the core kernel according to the memory-access permission in *PKRU*. Nevertheless, *rdmsr/wrmsr* may also be used to operate *FS/GS* in the critical path. UnderBridge can configure these two specific registers not to trap or replace them with some newer non-privileged instructions (e.g., *wrfsbase*).

Similarly, since control registers *CR0* and *CR4* are set at boot time only, UnderBridge also traps the setting instructions. Nevertheless, accessing *CR2* and reading *CR0*/*CR4* cannot trigger VMExits. UnderBridge clears *CR2*, where CPU saves the page fault address in the fault handler to prevent information leakage, and hides the real *CR0*/*CR4* value with a shadow value by leveraging virtualization hardware functionality. As for *CR3*, it points to page tables and needs modifications when switching address spaces, which frequently appears in the critical path. So, triggering *VMExits* on *CR3* modifications is expensive. Instead, we use the following lightweight solution (*a special method*). When loading system servers, UnderBridge leverages binary scanning and rewriting to guarantee the servers contain no *CR3* modification instructions, including at unaligned instruction boundaries. While in the core kernel, this privileged instruction must exist to switch address spaces. UnderBridge prevents a compromised system server from executing this instruction in the core kernel through defenses in depth. First (a simple defense), the instruction location is unknown to the servers. Second, to achieve higher security, the core kernel can write this instruction right before executing it and immediately remove it after the execution. Thus, a system server cannot execute it even if knowing its location. Furthermore, on different cores, UnderBridge makes the page table mapping for this instruction (page) different. So, when one core writes this instruction, other cores still cannot execute it.

Instructions that invalidate cache/TLB may be used by compromised servers for conducting performance attacks. While trapping cache invalidation instructions via *VMExits* does not affect overall performance since they are rarely executed, flushing TLB frequently appears on the critical path. So, we use binary rewriting to ensure there are no TLB flush instructions in system servers instead of trapping these instructions. The core kernel must contain these instructions, but they cannot be abused by faulting servers because we make sure they are followed by instructions that access the core kernel memory. Cacheline flush instructions (non-privileged instructions, e.g., *clflush* and *clflushopt*) are also considered because the system servers share the same address space with the core kernel. Nevertheless, these instructions obey MPK memory (read) checks and thus, cannot be utilized by a server to flush others' memory areas.

For other privileged instructions and I/O related operations, we take similar solutions as listed in Table-2. One thing to notice is that a compromised server may disable interrupts

Categories	Related Instructions or Registers	Usages in Zircon/Fiasco.OC/seL4/ChCore or Brief Explanations	Solutions
Load/Store System Registers	IDTR, GDTR, LDTR TR, XCR0 ...	Although seL4 uses "LTR" instruction when switching processes, it can be removed by setting different TSSs at boot time as do in other microkernels. Others are required at boot time only.	VMExit
Debug/Profile Registers	Debug registers RDPMC	Required for debugging and profiling, which are not performance-oriented.	VMExit
Model Specific Registers	RDMSR/WRMSR	Usually, they are mostly used at boot time or debug time and can trigger VMExits for selected registers according to configuration bitmaps.	Selected VMExit

Read/Write Control Registers	mov CRn, reg
	mov reg, CRn
	CLTS (modify CR0)
		- In the four microkernels, CR0/CR4 is written at boot time only and CR8 is not used.
		- No VMExits: accessing CR2, reading CR0 and CR4.
		- CR3 is used for switching address space. So, we handle it with a special method.
			- VMExit
			- Clear/hide
			- Special method

Cache/TLB States	- INVD/WBINVD - INVLPG/INVPCID - clflush instructions	- Whole cache eviction. WBINVD is rarely used, and INVD is even not used. - TLB clear is needed after updating page tables. So, triggering VMExits is costly here. - (executable in Ring-3) Evicting a single cache line. MPK checks take effects.	- VMExit - Binary rewritng - None
I/O Related Operations	- Port I/O - MMIO - DMA	- Port I/O is not performance-oriented and can trigger VMExits with I/O bitmaps. - MMIO operations go through MPK checks. - The core kernel initializes DMA devices at boot time and takes the control plane.	- VMExit - None - None
Other Privilege Instructions	- SMSW, RSM, HLT ... - SWAPGS, SYSRET... - CLI, POPFQ...	- Either related to other modes like legacy and SMM mode or rarely used. - Cannot break the system states, otherwise leading to the execution of fault handlers. - Can be used by compromised servers to disable interrupts.	- VMExit - None - Check in VMM
PKRU Register (Ring-0/3)	- xsave set instructions - WRPKRU	- For restoring extended processor states, which may include PKRU state. - For changing the value of PRKU register as used in IPC gates.	- No restoring - Binary rewriting

Table 2: Deprive the execution domains for system servers of the ability to execute privileged instructions.

through instructions like *cli* to monopolize the CPU. Fortunately, it cannot disable the host timer interrupts, which unconditionally trigger *VMExits*. Thus, the secure monitor can easily detect such malicious behaviors by checking *PKRU* and *interrupt state*.

Last but not least, we must forbid system servers from changing the *PKRU* register, i.e., changing the memory-access permission, by themselves. There are two kinds of instructions that can modify *PKRU*. For the first kind (*xrstor/xrstors*), the core kernel configures them not to manage *PKRU* by setting a control bit (bit-9) in *XRC0*. For the second kind (*wrpkru*), the core kernel ensures it does not exist outside the IPC gates by rewriting the binary code (similar to [77]). The *wrpkru* in the IPC gates cannot be abused as specified in § 4.1.

We omit the detailed policies of the binary rewriting in this paper as it is a mature technique [17, 25, 77]. Nevertheless, it is worth noting that using binary rewriting to directly eliminating all privilege instructions is undecidable because some privilege instructions only take one byte (e.g., hlt). Our hybrid approach is both effective and efficient.

4.3 Security Analysis

By introducing in-kernel servers, our system has one major difference from existing microkernels, which may lead to a larger attack surface. The in-kernel servers run in the kernel mode, which means a compromised server is able to execute any privileged instruction. We will analyze the attacks caused by the difference and illustrate how to defend against them.

Restricting privileged in-kernel servers: System servers are not trusted in our threat model. Although they can run in the kernel mode, they are highly restricted when trying to attack the core kernel, other servers, or the applications.

We assume that an attacker has fully compromised an in-kernel server and can execute arbitrary instructions. Since the server runs in another execution domain (no access to the memory domain 0), it cannot directly access the memory of the core kernel. As long as it tries to read or write any disallowed memory, a CPU exception will immediately be triggered and handled by the core kernel.

There are four ways to bypass the memory isolation mechanism enforced by MPK: the first one is to run the disabling instructions, e.g., by setting *CR4.PKE* to 0 or setting *CR0.WP* to 0; the second is to change the *PKRU* register to gain access permission of other memory domains illegally; the third is to change the page table base address by setting the *CR3* register; the fourth is to modify the page table directly.

ChCore can defend against all these attacks. Before loading a server, ChCore uses binary scanning/rewriting to eliminate the undesired privileged instructions. At runtime, the secure monitor will prevent a server from executing other privileged instructions. Compared with running on traditional microkernels, servers, including maliciously crafted ones, have no more attack means on microkernels with UnderBridge. First, when the malicious server executes the disabling instructions, it will trigger *VMExits*, and the monitor will locate the compromised server. Second, as described in the last paragraph of § 4.2, the two ways of modifying the *PKRU* register are prevented. Third, the malicious server has no way to modify *CR3* since the binary rewriting guarantees no *CR3* modification instructions exist in any server's address space. Fourth, the malicious server cannot modify the page table because the kernel page table resides in memory domain 0. Meanwhile, it cannot modify or add instructions which require to change the page table first. The isolation between

in-kernel servers is the same as the isolation between an in-kernel server and the core kernel. Since an in-kernel server does not share address space with user applications, it cannot access applications' memory either.

Defending side-channel attacks: Since all the in-kernel servers share one address space, it is easier for a malicious one to issue Spectre [45] and Meltdown [54] attacks compared with the case where all servers have their own address spaces. Although these attacks are caused by CPU bugs (out-of-scope), ChCore can mitigate them with existing software defenses like using address randomization makes a compromised server hard to locate the sensitive memory area. Considering the secret tokens leakage on buggy CPUs, extra checks can be added in the IPC gates, e.g., Line-10 in Figure-4(b), to prevent malicious PKRU modification and ensure the memory isolation. Besides that, most known hardware vulnerabilities have been fixed by major CPU vendors in their latest products [5, 6], which is orthogonal to ChCore.

5 Implementation

Based on our UnderBridge design, we have implemented a prototype microkernel ChCore, which contains about 8500 lines of C code (LOC). ChCore runs in guest-mode, i.e., non-root mode on x86_64, and a small secure monitor (around 300 LOC) runs in hypervisor-mode, i.e., root mode on x86_64. We have implemented the tiny secure monitor in a minimal virtualization-layer, RootKernel (1,500 LOC) of SkyBridge [62]. Note that RootKernel is not a hypervisor and works only for running one OS in the guest-mode and thus avoids most overhead caused by virtualization. Therefore, although our system requires hardware virtualization, it still can be deployed in bare-metal machines with RootKernel and achieve close-to-native performance. Considering the above, our system increases the trusted computing base (TCB) by 1,800 LOC in total when running on bare-metal machines, which is acceptable. Besides, we also integrate the tiny secure monitor in a commercial hypervisor, KVM, which makes deploying our system in cloud feasible. Even if nested virtualization is required, our secure monitor can still work because it simply utilizes the hardware-provided capability to trap sensitive instructions. Because the instructions to trap are deliberately selected and do not exist on critical paths, they will not degrade the overall performance. Nevertheless, our current implementation requires cloud providers to patch their hypervisors. In such a case, our system increases TCB by 300 LOC. Alternatively, we may leverage eBPF [61, 81] to deploy our secure monitor without modifications to the commercial hypervisor (left as future work).

We also apply UnderBridge to three state-of-the-art microkernels, i.e., seL4 [3], Zircon, and Fiasco.OC, to demonstrate the generality of the design. The porting effort is about 1000∼1500 LOC for each of them. Since UnderBridge uses different page tables for applications and kernel, it also enables and leverages the PCID hardware feature for avoiding unnecessary TLB flushing. As native Fiasco.OC does not support to use this feature, we also add a simple extension to assign different PCIDs to an application and the kernel.

[3] We do not retain formal correctness guarantees of seL4.

6 Performance Evaluation

Basic Setup. We conduct all the experiments on a Dell PowerEdge R640 server, which is equipped with a 20-core Intel Xeon Gold 6138 2.0GHz CPU and 128GB memory. Both Turbo Boost and Hyper-threading are disabled. ChCore runs on Linux/KVM-4.19 and QEMU-4.1.

Systems for Comparison. We evaluate the native IPC performance of three popular microkernels (Zircon, seL4, and Fiasco.OC) on bare metal. Also, we evaluate SkyBridge [62], which is the state-of-the-art optimization for IPC in microkernels by using *vmfunc*. SkyBridge deploys a small hypervisor called RootKernel and runs microkernels in non-root mode. When evaluating UnderBridge, we deploy the secure monitor of UnderBridge in RootKernel and run microkernels with UnderBridge on it. As Zircon has no kernel-page-table-isolation (KPTI) support, we simulate the overhead of page table switching by writing *CR3* twice when evaluating UnderBridge on it, this is because UnderBridge requires to make the kernel and applications use different page tables.

6.1 IPC Performance Analysis

Cross-server IPC. Firstly, we analyze the IPC performance between two servers (we abbreviate "system server" as "server" in this section) in a micro-benchmark, which uses *rdtsc* instruction to measure the *round-trip* latency of invoking an empty function in server B from server A.

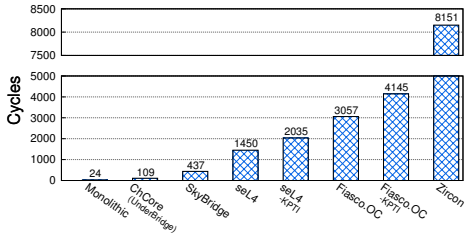


Figure 5: Round-trip latency of cross-server IPC.

Figure-5 gives the absolute cost of cross-server IPCs in different designs. Since invocations between servers (components) in monolithic kernels are usually achieved by using (indirect) *call*/*ret* instructions, the round-trip latency is only 24 cycles. The latency of an IPC round-trip in ChCore is 109 cycles, which is dominated by two *wrpkru* instructions (56 cycles in total). Note that the IPC is achieved by UnderBridge and thus only involves the lightweight *wrpkru* and the procedure of saving necessary registers. SkyBridge requires two much heavier *vmfunc* instruction (292 cycles in total) and therefore has larger latency.

The round-trip latency of a native IPC in the other three microkernels are much more noticeable. Among them, seL4

shows the best performance as it will directly switch the caller thread to callee thread when executing fast-path IPCs. Although Fiasco.OC applies a similar strategy, it has a more complex IPC capability handling procedure. Zircon does not support direct-switch and thus has the worst performance due to the high scheduling cost.

Compared with the native IPC in the three microkernels, our UnderBridge design is more than 12.3× faster. The performance improvement mainly comes from two parts. First, UnderBridge avoids the time-consuming privilege switches in traditional IPC designs, as measured in § 2.2. Second, UnderBridge avoids the complex validation and invocation of the IPC capability on the critical path. An IPC gate is only generated after the corresponding capabilities having been checked, so UnderBridge needs not to check the capability at runtime. Also, the gate only requires several lightweight instructions for the domain switching (details in § 4.1).

Another benefit of UnderBridge is that the in-kernel servers can invoke system calls faster via the IPC gates instead of using *syscall* instructions.

Application-to-Server IPC. We further analyze the IPC (round-trip) performance between a user application and a server in this part. Commonly, a system call involves multiple servers in microkernels (or multiple kernel components in monolithic kernels), which means an application-to-server IPC may involve several cross-server IPCs. To simulate such cases, we design a micro-benchmark that includes one application and several servers. Each server will do nothing but routing IPCs to another server. We vary the number of cross-server IPCs in the benchmark.

Approaches	Cross-server IPCs		
	0	1	2
SkyBridge	437	931	1390
ChCore (UnderBridge)	723	856	981
seL4	1450	2932	4266

Table 3: Round-trip latency (cycles) of one application-to-server IPC and different number of cross-server IPCs.

Table-3 compares the performance of SkyBridge (applied on Fiasco.OC) and UnderBridge (applied on ChCore) in this micro-benchmark. Results are similar when we apply them to other microkernels and thus omitted. If an application invokes a server *without* causing any cross-server IPCs, SkyBridge (437 cycles) shows better performance than UnderBridge (723 cycles). It is because UnderBridge has to switch the privilege level and the address space for transferring the control flow from user space to kernel space, while SkyBridge applies more lightweight EPT switching via *vmfunc*. Nevertheless, UnderBridge still outperforms the best native IPC (1450 cycles in seL4), because of reducing one privilege switch and minimizing the software logic of an IPC.

As the number of cross-server IPCs increases, the latency of SkyBridge increases in proportion. It is because that IPC from application to the server and between servers are symmetric and cost the same cycles. In contrast, the latency of UnderBridge grows much slower because its cross-server IPC is much more lightweight. As shown in Table-3, the performance of UnderBridge becomes better than SkyBridge when involving one cross-server IPC, and is better than that by 42% when involving two. The performance speedup is expected to grow along with the increasing number of the cross-server IPCs and finally close to 3.0× as in Figure-5.

6.2 Application Benchmarks

We further evaluate the performance of UnderBridge with two real-world applications: a database application and an HTTP server application. In the following experiments, shared memory is used to transfer data during IPCs.

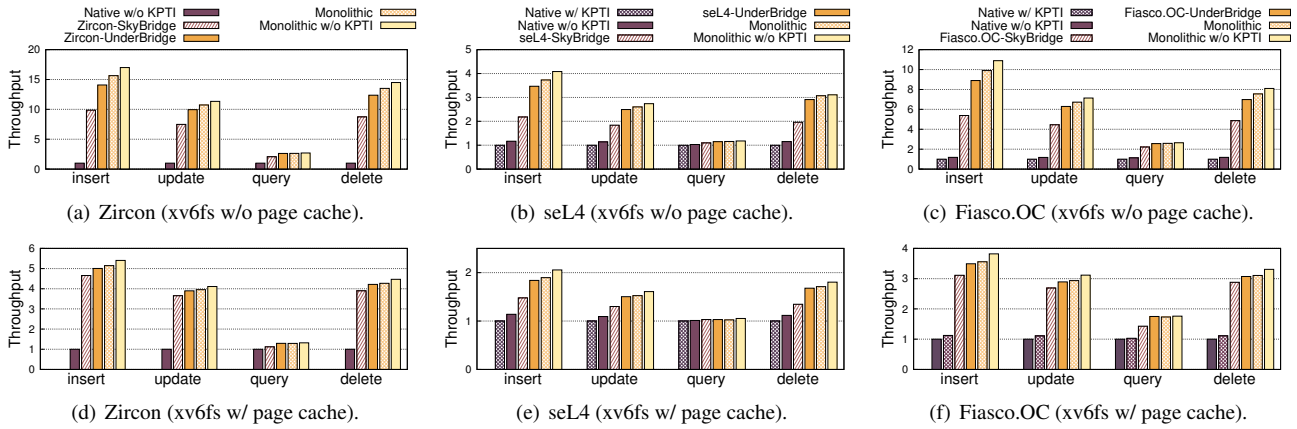
Database Evaluation. To faithfully compare with SkyBridge, we use the benchmarks in [62]. Specifically, for serving a relational database, SQLite3 [15], we run two system servers: one is a file system named xv6fs [16, 23], and the other is a RAMdisk (memory-only). When SQLite3 operates on a file, it will first invoke the xv6fs server by an application-to-server IPC, and then the xv6fs will read or write the RAMdisk by cross-server IPCs.

Basic Operations. We first evaluate the performance of basic operations, including insert, update, query, and delete operations. Our evaluation includes three IPC approaches: the native IPC, UnderBridge, and SkyBridge. Specifically, for each microkernel, we not only evaluate the performance with its native IPC designs but also test the performance after applying UnderBridge and SkyBridge to it. We also emulate the performance of a monolithic kernel by replacing all the IPC gates in UnderBridge with function calls.

Figure-6(a), 6(b) and 6(c) show the normalized throughput of basic operations in the three microkernels, separately. The baseline is set as the performance of native IPC design in each microkernel.

UnderBridge achieves up to 13.1×, 9.0×, 1.6× and 11.3× speedup for each of these operations, individually. The improvement of query operations is relatively small since SQLite3 has an internal buffer for storing recent data and may handle the queries without issuing IPCs. Compared with SkyBridge, the performance improvement of UnderBridge (up to 65%) is because a single IPC from SQLite3 to xv6fs is likely to trigger multiple cross-server IPCs between xv6fs and RAMdisk. Even compared with the emulated performance of monolithic kernels, UnderBridge only introduces about 5.0% overhead.

The above-tested xv6fs (exactly the same one used in the SkyBridge paper [62]) contains no page cache, which, thus, emulates an IPC-intensive scenario. We further enable the page cache in xv6fs to show how UnderBridge performs with fewer cross-server IPCs between xv6fs and RAMdisk. As shown in Figure-6(d), 6(e) and 6(f), the performance improvement of UnderBridge is still obvious. UnderBridge shows up to 4.0×, 2.9×, 0.7× and 3.2× speedup



(a) Zircon (xv6fs w/o page cache). (b) seL4 (xv6fs w/o page cache). (c) Fiasco.OC (xv6fs w/o page cache).

(d) Zircon (xv6fs w/ page cache). (e) seL4 (xv6fs w/ page cache). (f) Fiasco.OC (xv6fs w/ page cache).

Figure 6: Normalized throughput of basic SQLite3 operations.

compared with the native IPC and achieves comparable performance with monolithic kernels. Nevertheless, since fewer cross-server IPCs are involved, the maximum improvement of UnderBridge compared with SkyBridge drops from 65% to 25%.

Since UnderBridge runs in non-root mode, we also count the number of *VMExit*, which is known as the cost of virtualization. Thanks to the careful design of RootKernel and our secure monitor, there are *almost zero VMExits* during the experiments. For example, at most one VMExit (due to timer) happens during the query test.

YCSB Benchmark. We also evaluate SQLite3 against YCSB workloads. Figure-7(a) and 7(b) show the normalized throughput of YCSB-A (50% update and 50% query) with the page cache disabled and enabled in xv6fs, separately. We use the same baseline as the basic operation evaluation. Even for seL4, which is the most efficient among the three microkernels, UnderBridge improves the application's throughput from 35% to 105%. UnderBridge also brings a better performance (from 7% to 35%) than SkyBridge. Besides, it is only slightly slower (3.3% on average) than the monolithic kernel. Other YCSB workloads give similar results.

Furthermore, Figure-7(c) gives a detailed analysis of the experiments with page cache *enabled* in xv6fs. First, the ratio of IPCs from SQLite3 to xv6fs (application-to-server) and xv6fs to RAMdisk (cross-server) is about 1:2. Thus, UnderBridge outperforms SkyBridge, according to Table-3. Second, it helps to reduce the ratio of time spent on IPCs to around 11% while the other three microkernels spend at least 30% of the time on IPCs. Third, it makes the application and the servers execute faster (about 10%) owing to less indirect costs such as cache/TLB pollution.

Server Migration. We also evaluate the performance of server migration, although it should rarely happen. We still run SQLite3, xv6fs, and RAMdisk as above on Zircon and trigger the server migration. Taking RAMdisk (128 MB virtual memory range) as an example, migrating it from kernel to user takes about 84,361 cycles. Most of the cycles (83,517) are spent on modifying the kernel page table to free the domain ID (i.e., the third step for migrating a server specified in § 3.3). Migrating RAMdisk from user to kernel takes more cycles (90,189) mainly because more cycles are spent on waiting for finishes of on-going IPCs.

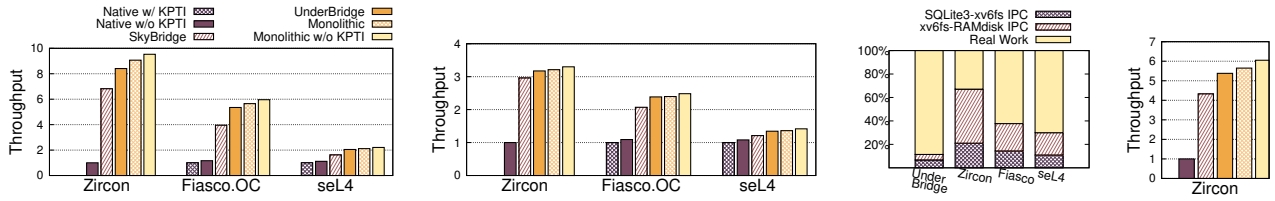
HTTP Server Evaluation. For running an HTTP server (a user-space application), we create three system servers: a socket server, a TCP/IP protocol stack server, and a loopback network device driver (not involving the real network device) atop lwIP [31] library. We measure the throughput of a simple HTTP server from the client-side, which receives requests from the network and sends back a static HTML page.

We perform this evaluation on Zircon. As shown in Figure-7(d), UnderBridge improves the throughput of the HTTP server by 4.4×. We also implement the same benchmark with SkyBridge. UnderBridge outperforms SkyBridge by about 24% because a network request also triggers multiple cross-server IPCs.

7 Related Work

Reconstructing monolithic kernels. The development of monolithic kernels follows the philosophy of modularization, but all the kernel components are not isolated from each other. With reliability and security attracting a fair amount of attention, we witness interest in reconstructing monolithic kernels to achieve better fault isolation and higher security [17, 25, 26, 38, 41, 59, 63, 65, 67, 74, 75, 84, 86]. Dautenhahn et al. [25] build one memory protection domain inside the BSD kernel and run a small trusted kernel in that domain to control memory mappings. Mondrix [84] implements a compartmentalized Linux kernel with eleven isolated modules based on customized security hardware [83]. Proskurin et al. [65] propose to use Intel EPT and *vmfunc* to isolate critical kernel data in different domains. Nooks [74] and LXFI [59] focus on improving the reliability of Linux by isolating kernel modules, especially device drivers.

Our work does share some similarities with prior work on intra-(monolithic)kernel isolation. Nevertheless, we fo-



(a) YCSB-A (xv6fs w/o page cache). (b) YCSB-A (xv6fs w/ page cache). (c) Breakdown analysis. (d) HTTP server.

Figure 7: (a), (b), and (d) share the same legend. (a) and (b) show the normalized throughput of YCSB-A with xv6fs's page cache disabled and enabled separately. (c) shows the time breakdown of YCSB-A benchmark. (d) demonstrates the normalized throughput of a HTTP server.

cus on accelerating IPCs for microkernel architectures while maintaining strong isolation (both ends). We need to do little modification/instrumentation on system servers of a microkernel. This is because system servers of a microkernel are designed to run in different user processes, and all the interactions are explicit IPCs, which is different from the subsystems in Linux (no clear boundaries and have complex shared memory references). We achieve intra-kernel isolation by retrofitting Intel MPK, which is lightweight and commercially-available, to build multiple execution domains in kernel space. Furthermore, UnderBridge may also be generalized to other kernel scopes with more efforts in the future. On one side, the proposed abstraction of the execution domain can be extended to accommodate different kernel modules in monolithic kernels, and the IPC gates can still be used to handle interactions between those modules. On the other side, our design can also be applied in kernels written in memory-safe languages [11, 24] to isolate some unsafe code (e.g., the code with "unsafe" tag in Rust).

Accelerating IPCs. Optimizing the performance of IPC in microkernels is continuously studied for a long time [19, 36, 44, 50, 52, 82]. For example, LRPC [19] eliminates the scheduling overhead during an IPC by using the thread-migration model [36, 37]; seL4 [44] provides the fast-path IPC, which also avoids scheduling and passes arguments through registers. Nevertheless, even with these software-based optimizations, IPC-intensive applications on microkernels still suffer from the IPC overhead.

Recent studies present new designs to accelerate IPCs with advanced hardware features. SkyBridge [62] utilizes the *vmfunc* to allow a process to invoke a function in another process directly without the kernel's involvement. XPC [30] is a hardware proposal that accelerates IPCs by implementing efficient context switch and memory granting. dIPC relies on another hardware proposal [78] to put processes into the same address space and thus make IPCs faster. Our design shows better performance than SkyBridge and requires no hardware modification.

Usage of Intel MPK. Intel MPK/PKU has been utilized by [39, 64, 77] to achieve efficient intra-process isolation, which can build mutual-distrusted execution environments in a single user process. There are two main differences between UnderBridge and them. First, UnderBridge retrofits MPK that designed for user space in kernel space to improve the IPC performance for microkernels (the first effort to our knowledge) and also faces more security challenges. Applications can still utilize MPK in user space as they want since they have different page tables from the kernel. Second, UnderBridge authenticates the caller of each MPK gate for enforcing IPC capability to prevent illegal domain switches, while prior work allows arbitrary domain switches.

There exist two concurrent studies [38, 73] to UnderBridge. They also propose to utilize MPK in kernel mode but with different goals and designs. IskiOS [38] leverages MPK to defend against code-reuse attacks (e.g., protecting shadow stacks) in the kernel. Sung et al. [73] uses MPK to enhance the isolation for a unikernel while does not solve the security challenges identified in UnderBridge.

MPK-like features on other architectures. Tagged memory [28, 43, 85], which can provide MPK-like features, is added in other architectures, which brings the potential to make UnderBridge more general to those architectures. Recently, the RISC-V security community also considers enhancing the PMP (physical memory protection) isolation with the tagged memory mechanism [12]. However, UnderBridge cannot work on current ARMv8 (aarch64). Yet, ARM v8.5 has included memory tagging extensions [10], by extending which with more mechanisms, we may provide a similar mechanism workable on future ARM platforms.

8 Conclusion

This paper introduces UnderBridge, a redesign of the runtime structure of microkernel OSes for faster OS services. To demonstrate UnderBridge's efficiency, we have built a prototype microkernel named ChCore and ported it to three existing microkernels. Performance evaluations showed that UnderBridge can achieve better performance in IPC-intensive workloads compared with prior work.

9 Acknowledgement

We sincerely thank our shepherd Antonio Barbalace and the anonymous reviewers for their insightful suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206, 61972244, and U19A2060) and a grant from the Science and Technology Commission of Shanghai Municipality (No. 19511121100). Yubin Xia is the corresponding author.

References

[1] Apple ios security-ios 12.1. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Referenced December 2019.

[2] Fiasco.oc repository. https://l4re.org/download/snapshots/. Referenced December 2019.

[3] Fuchsia. https://fuchsia.dev/fuchsia-src. Referenced December 2019.

[4] Fuchsia repository. https://fuchsia.dev/fuchsia-src/development/source_code. Referenced December 2019.

[5] Ian cutress: Analyzing core i9-9900k performance with spectre and meltdown hardware mitigations. https://www.anandtech.com/show/13659/analyzing-core-i9-9900k-performance-with-spectre-and-meltdown-hardware-mitigations. Referenced December 2019.

[6] Intel corporation. engineering new protections into hardware. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html. Referenced December 2019.

[7] Intel software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Referenced December 2019.

[8] Kernel page table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation. Referenced December 2019.

[9] Linux kernel cves. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Referenced December 2019.

[10] Memory tagging in armv8.5-a. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a. Referenced May 2020.

[11] Redox operating system. https://www.redox-os.org/. Referenced December 2019.

[12] Risc-v isa specification. https://riscv.org/specifications/. Referenced May 2020.

[13] sel4 performance report. http://sel4.systems/About/Performance/. Referenced December 2019.

[14] sel4 repository. https://github.com/seL4/seL4. Referenced December 2019.

[15] Sqlite. https://www.sqlite.org/index.html. Referenced December 2019.

[16] xv6 repository. https://github.com/mit-pdos/fscq/tree/master/xv6. Referenced December 2019.

[17] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[18] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[19] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[20] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 769–784, New York, NY, USA, 2019. ACM.

[21] 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, pages 559–572, New York, NY, USA, 2010. ACM.

[22] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[23] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[24] Cody Cutler, M Frans Kaashoek, and Robert T Morris. The benefits and costs of writing a {POSIX} kernel in a high-level language. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 89–105, 2018.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth In-*

ternational Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[26] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[27] Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. Curios: improving reliability through operating system structure. pages 59–72, 2008.

[28] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2014.

[29] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. ACM.

[30] 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*, ISCA '19, pages 671–684, New York, NY, USA, 2019. ACM.

[31] Adam Dunkels. lwip-a lightweight tcp/ip stack. *Available from World Wide Web: http://www. sics. se/ adam/lwip/index. html*, 2002.

[32] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[34] Úlfar 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*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[35] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, New York, NY, USA, 1996. ACM.

[36] Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 9–9, Berkeley, CA, USA, 1994. USENIX Association.

[37] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association.

[38] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Iskios: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[39] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 489–503, Berkeley, CA, USA, 2019. USENIX Association.

[40] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[41] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. Lightweight capability domains: Towards decomposing the linux kernel. *SIGOPS Oper. Syst. Rev.*, 49(2):44–50, January 2016.

[42] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[43] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[44] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[45] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin,

Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[46] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452, New York, NY, USA, 2017. ACM.

[47] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[48] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1441–1454, New York, NY, USA, 2018. Association for Computing Machinery.

[49] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[50] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[51] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[52] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[53] Jochen Liedtke. A persistent system in real use - experiences of the first 13 years. pages 2 – 11, 01 1994.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[55] 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 *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 973–990, Berkeley, CA, USA, 2018. USENIX Association.

[56] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 49–64, Berkeley, CA, USA, 2016. USENIX Association.

[57] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619, New York, NY, USA, 2015. ACM.

[58] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 26:1–26:16, New York, NY, USA, 2018. ACM.

[59] 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.

[60] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[61] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[62] 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*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.

[63] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[64] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 241–254, Berkeley, CA, USA, 2019. USENIX Association.

[65] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Poly-

chronakis. xmp: Selective memory protection for kernel and user space. In *Proceedings of 41st IEEE Symposium on Security and Privacy*, S&P '20, 2020.

[66] Franklin Reynolds. An architectural overview of alpha: A real-time, distributed kernel. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Berkeley, CA, USA, 1992. USENIX Association.

[67] O. Schwahn, S. Winter, N. Coppik, and N. Suri. How to fillet a penguin: Runtime data driven partitioning of linux code. *IEEE Transactions on Dependable and Secure Computing*, 15(6):945–958, Nov 2018.

[68] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[69] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[70] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[71] 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, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.

[72] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[73] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–156, 2020.

[74] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[75] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. A policy-centric approach to protecting os kernel from vulnerable lkms. *Software: Practice and Experience*, 48(6):1269–1284, 2018.

[76] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science on Experimental Computer Science*, ecs'07, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[77] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1221–1238, Berkeley, CA, USA, 2019. USENIX Association.

[78] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. Codoms: Protecting software with code-centric memory domains. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480. IEEE, 2014.

[79] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 16–31, New York, NY, USA, 2017. ACM.

[80] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[81] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 33–47, 2014.

[82] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, September 2016.

[83] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[84] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*,

SOSP '05, pages 31–44, New York, NY, USA, 2005. ACM.

[85] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[86] Xi Xiong, Donghai Tian, Peng Liu, et al. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, volume 11, 2011.

[87] 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 *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[88] 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.