EPK: Scalable and Efficient Memory Protection Keys

Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen, *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China, Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc22/presentation/gu-jinyu

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

EPK: Scalable and Efficient Memory Protection Keys

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

Abstract

As a hardware mechanism for facilitating intra-process memory isolation, Intel Memory Protection Keys (MPK) has been leveraged to efficiently improve the isolation, security, or performance of the software. However, it can only support 16 isolated memory domains, which significantly limits its applicability in many scenarios.

In this paper, we present EPK which leverages off-the-shelf virtualization hardware features to extend the number of available protection domains in MPK. To demonstrate the effectiveness of EPK, we apply it in three scenarios, including better memory isolation for server applications as well as Non-Volatile Memory (NVM) applications, and a fast Inter-Process Communication (IPC) mechanism for microkernels. The evaluation results show that EPK can scale to provide hundreds of isolated domains. It can outperform the state-of-the-art (libmpk) by up to two orders of magnitude and usually achieve 95% of the performance of the system with no memory isolation.

1 Introduction

Intel MPK [7] has attracted many researchers since introduced in 2019 because it offers highly-efficient intra-process memory isolation by supporting memory domains inside one application. An application can switch between different domains with a new instruction, *WRPKRU*, which can execute in the user mode directly and takes only about 28 cycles. Compared with traditional software isolation or page table based isolation, MPK can achieve much lower performance overhead, and has been adopted in many scenarios, including: 1) enhancing the isolation between different threads of the same process by giving them different domain views [13, 55, 57]; 2) hardening the security of an application by separating different components, such as untrusted third-party libraries, into different domains [24, 40, 44, 46, 50]; and 3) improving the performance of software that uses multiple page tables for isolation by substituting domains for page tables [22, 29].

However, the small number (16) of isolated memory domains supported by MPK severely undermines its usability. First, typical server applications usually serve for more than 16 clients concurrently, and it is preferable to store clients' private data in isolated domains to prevent sensitive data leakage due to vulnerabilities like Heartbleed [5]. Second, there is a growing interest in protecting persistent memory [6] data from accidental or malicious accesses [56, 57]. Long-lived persistent data is usually directly mapped into processes and then accessed via load/store instructions. Isolating the data in more domains can reduce the data exposure time and benefit stray access protection. Third, both applications and system software may contain more than 16 components that need to be isolated. For example, popular applications use scores of third-party libraries [2]; an OS consists of tens or hundreds of modules like device drivers. Besides, prior studies also indicate the performance of NVM applications (which desire isolation) [57] and microkernel OSes [22] can boost by more than 10× with more MPK domains.

To scale MPK beyond 16 memory domains, recent researchers propose either software or hardware approaches to support more MPK domains [38, 41, 57]. However, the software approach suffers from a large overhead while the hardware approaches are infeasible on commodity machines.

In this paper, we propose EPK, which extends the maximum number of memory domains supported by MPK on commodity hardware efficiently. MPK's performance advantage stems from the decoupling of domain configuration (in privilege mode) and domain switching (in non-privilege mode). Our observation is that another hardware feature, named *fast EPT-switching* (Extended Page Table switching, with *VMFUNC*), has a similar pattern, which decouples EPT configuration (in host mode) from EPT switching (in guest mode). Thus, we propose *extended protection keys* by combining MPK with fast EPT-switching, i.e., reusing the same MPK protection keys in different extended page tables (EPT). Thus, with 512 EPTs, EPK can support up to 7,680 domains.

However, there are two major challenges to the new system. The first challenge is to provide a unified abstraction for appli-

cations although combing two orthogonal hardware features. EPK still retains the abstractions of memory domain and domain switching inherited from MPK while hiding the EPTs from applications, by elaborately managing domain mappings in multiple EPTs and developing a library to provide easy-to-use APIs. The second challenge is to enable one thread to simultaneously access memory domains across different EPTs, as the original MPK allows to access multiple domains together. To this end, EPK leverages another existing hardware feature named virtualization exception (VE) to switch the EPTs for the thread transparently when a domain access causes EPT violations.

We implement EPK prototype and apply it in the above scenarios. On the one hand, EPK can work like the original MPK for mitigating the memory errors and thus facilitates efficient intra-process memory isolation (Section 5). On the other hand, it can also isolate untrusted software components [46, 57] (Section 4 and Section 6) by further preventing illegal domain switching.

Experiments on server applications and persistent memory applications show that EPK's overhead is usually around or below 5%. Compared with the state-of-the-art (libmpk) [38], the performance improvement can be up to two orders of magnitude. Furthermore, we incorporate EPK in a microkernel OS, a representative of large software. A microkernel OS runs system components like file systems and device drivers in user processes for embracing better isolation [20, 26, 30]. Nevertheless, costly inter-process communication (IPC) is required for the interaction between different OS components [22, 31, 37, 45]. EPK can provide enough isolated domains for running different OS components and the fast domain switch for IPCs. Thus, we propose a high-efficient IPC mechanism named HyBridge that can improve the performance of three well-known microkernels, seL4 [10], Google Zircon [4], and Fiasco.OC [3], and outperform two state-of-the-art IPC designs, SkyBridge [37] and UnderBridge [22].

In summary, this paper makes the following contributions: 1) a scalable and efficient intra-process memory isolation mechanism named EPK; 2) a real implementation and evaluation on Linux; 3) a new IPC design based on EPK for microkernel OSes with better performance.

2 Background and Motivation

2.1 Hardware Background

MPK. Intel MPK [7] can divide the virtual memory space of one process into 16 memory domains. By leveraging previously unused bits of the page table entry, each memory page is tagged with a four-bit protection key as the domain ID and exclusively belongs to one of the 16 domains. A new 32-bit register, *PKRU*, is introduced to specify the access permissions (read-only, read-write, none) on the 16 domains (two bits for one domain). Because the register is per-core, concurrent threads in the same process can have different access permissions on different domains. During runtime, MMU transparently checks the permissions. A non-privileged instruction called *WRPKRU* can update this register to change the access permissions.

MPK in the VM. The hardware feature of MPK is also usable in a VM. Protection keys are still tagged in the page tables of applications instead of EPTs. From the perspective of applications and the OS, the usage of of MPK is just the same no matter in the VM or not.

Extended Page Table (EPT) and VMFUNC. Intel hardware virtualization technology employs EPT for memory virtualization. For a guest virtual machine (VM), the guest page table maps guest virtual addresses (GVA) to guest physical addresses (GPA) while the EPT maps GPAs to host physical addresses (HPA) and thus aids in the seamless translation of GVAs to HPAs. The guest VM's OS (runs in non-root mode ring zero) controls the guest page table, while the hypervisor (runs in root mode) manages the VM's EPT. *VMFUNC* is a hardware virtualization extension that provides VM functions for VMs. EPT pointer (EPTP) switching is currently the only VM function provided, allowing the guest VM (both Ring-0 and Ring-3) to directly load a new EPTP. The loadable EPTP can only be chosen from a list of EPTPs (up to 512) configured by the hypervisor. Note that TLB entries are tagged with the EPT base addresses to avoid flushing the TLB when switching the EPT.

Virtualization Exceptions (VE). EPT violations usually trigger VMExits, after which the hypervisor can fill the EPT mappings. Yet, Intel virtualization technology also supports converting EPT violations into VE without VMExits. With VE enabled, the hypervisor can configure bit 63 of certain EPT paging-structure entries to make EPT violations on some GPAs to cause VE and others to cause VMExits as before.

2.2 Motivation

Software fault isolation (SFI) can enhance memory isolation for applications [15, 19, 27, 36, 48, 58] by instrumenting and restricting memory accesses. Nonetheless, it may result in non-negligible runtime performance overhead and is inflexible (e.g., hard to be fine-grained). Many studies can avoid such disadvantages [25, 27, 32, 35, 39] using the MMU. They isolate different memory partitions of a process in different page tables or extended page tables and thus utilize MMU to check memory accesses at the page granularity.

Instruction	Cost (cycles)
Write *CR3* (no TLB flush)	226
VMFUNC (switch EPT)	146
WRPKRU	28

(a)

Solution	Overhead
LwC-simulate	70%
EPT-based	12%
ERIM	3%

(b)

Table 1: (a) Instruction cost. (b) The overhead of isolating session keys in one isolated domain.

However, constructing different memory domains with page tables is not free. Switching between different domains requires changing the page table through specialized hardware instructions. Table 1(a) presents the direct cost of the related instructions. We design an experiment to isolate each client's session key in separate domains in the NGINX web server [9] to show the corresponding performance overhead. ApacheBench (ab) [1] generates the workload: 300 concurrent clients send requests to the server for a file. As presented in Table 1(b), light-weight contexts (lwC) [32], as a representative of page-table-based solutions, will lead to approximately 70% overhead if we isolate all the session keys in a separate context (i.e., a new page table) and switch to that context when accessing those keys. Similarly, for the EPT-based solution, we create a new EPT for isolating all the session keys and use *VMFUNC* instruction to switch to that EPT when accessing them. Although noticeably better than the page-table-based solution, such an EPT-based solution still introduces around 12% performance overhead. In contrast, ERIM [46] only adds about 3% overhead by utilizing MPK to construct an isolated memory domain for storing the session keys, which can demonstrate the efficiency of MPK.

Yet, MPK can only support at most 16 memory domains, limiting its usage. Take the web server for example: it is preferable to separate clients' data in different memory domains, guaranteeing the isolation between multiple clients. Recent work [38, 41, 57] also identifies and addresses this limitation of MPK. Two studies [41, 57] propose non-trivial hardware extensions for efficiently supporting scalable domains, which are not achievable on current platforms.

libmpk [38] gives the illusion of multiple memory domains by exposing virtual keys to applications and maintaining the mapping between virtual keys and the 16 real keys (one key for one domain). When all 16 real keys are exhausted and a new virtual key is required, libmpk will evict a mapped real key and remap it to the new virtual key. But the key eviction may incur a large overhead. For instance, if we protect each client's session key in a different memory domain (300 domains in total) provided by libmpk in the above NGINX experiment (rather than storing all keys in one domain), the overhead becomes about 20%. The overhead consists of both direct costs, i.e., the expensive key eviction procedure involving modifying page table entries, flushing TLBs, etc., and indirect costs, e.g., TLB misses due to flushing.

More seriously, libmpk's domain switch cost increases as domain memory gets larger, as shown in Table 2. The micro-benchmark keeps switching to one domain randomly. When the domain number increases from 32 to 64, more key eviction occurs, resulting in higher overhead. As one domain contains more memory pages, the switch cost gets more expensive due to flushing more TLBs and updating more page table entries. The cost turning point (from 33 to 34) is because Linux flushes all TLBs together instead of one at a time when the number of TLBs to flush exceeds 33.

Domains \ Pages	*16*	*33*	*34*	*64*	*1K*	*128K*
15	185	184	188	188	187	185
32	6,576	11,173	4,090	5,270	42,912	5.1×10^6
64	9,959	16,573	6,308	8,068	79,012	9.6×10^6

Table 2: The CPU cycles of domain switches in libmpk. The page size is 4k.

	Memory Access Cost	Domain Switch	Memory Domain Number	Multi -domain Access	Multi -thread Support	Hardware Changes
SFI	High	Fast	Many	No	Yes	Zero
lwC	Low	Slow	Many	No	Yes	Zero
Donkey	Low	Fast	1,024	Yes	Yes	Heavy
libmpk	Low	Slow	Many	No	No	Zero
MPK	Low	Fast	16	Yes	Yes	Zero
EPK	Low	Fast	7,680	Yes	Yes	Zero

Table 3: Comparison of different approaches.

In brief, MPK-based intra-process memory isolation shows attractive performance advantages but can only support a limited number of isolated domains. Therefore, we intend to overcome this limitation while retaining MPK's performance and flexibility advantages. As described in Table 3, SFI-based and page-table-based approaches (e.g., lwC) have performance issues and do not allow one thread to simultaneously access different domains. Existing hardware approaches (e.g., Donkey [41]) are hard to be implemented on commercial x86/ARM architectures due to intrusive hardware modifications. For example, to support 1024 domains, Donkey takes 10 bits in the page table entry as the domain ID, which is at least incompatible with the upcoming 5-level page table. libmpk makes several contributions like implementing fast *mprotect* by using MPK. But, its extension on the MPK domain number has both performance and flexibility issues. It cannot support multi-threading well, in particular, because it is difficult to maintain a consistent view of active domains across different threads.

3 The EPK Mechanism

According to prior studies on MPK-based intra-process isolation, the common usage model of MPK is as follows. An application (process) creates memory domains by binding different protection keys (pkey) to them as the domain IDs and separates the memory data into different domains. An application thread acquires/releases the access permission of one specific domain before/after accessing the data in it, which reduces the chances of the isolated memory being affected by vulnerabilities (e.g., leakage caused by buffer overflow) or faults (e.g., wild writes). Acquiring the domain access permis-

sion is efficiently achieved by executing *WRPKRU* instruction, which is referred to as switching to that domain. Releasing the permission is a reverse procedure that also makes use of *WRPKRU*. EPK still inherits such a usage model while supporting more memory domains.

Extended Protection Keys. The root cause of why MPK can only support 16 memory domains for one application is that each domain needs to exclusively take one pkey while the hardware only supports 16 pkeys. So, to extend the number of memory domains, the high-level idea of EPK is allowing multiple memory domains to use the same pkey at the same time. However, simply reusing the same pkey for different domains does not guarantee memory isolation. Therefore, EPK proposes *extended protection key*, which extends a pkey with different EPT indexes (get more keys), and then assigns different extended protection keys to different memory domains.

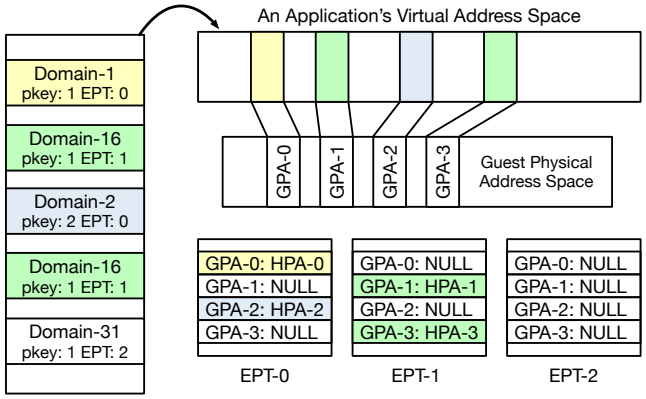


Figure 1: The memory mapping overview for an application.

As depicted in Figure 1, EPK allows an application to partition its virtual address space into different memory domains, with each domain containing discrete memory pages. A domain exclusively takes one *extended protection key* as its domain ID, which is composed of a pkey (1-15) and an EPT index (0-N, $0 \leqslant N < 512$)[1]. EPK requires an application to run within a VM where cloud applications usually run in, and multiple EPTs need to be created for the VM. Each EPT can hold 15 domains for an application (domain-0 is used as the shared domain), and the 15 domain IDs (extended protection keys) have the same EPT index but different pkeys. For example, domain-1 and domain-2 are both in EPT-0 and use pkey-1 and pkey-2, respectively. The same pkey can be shared by domains in different EPTs concurrently, e.g., domain-1, domain-16, and domain-31 can all use pkey-1 because they will be mapped in EPT-0, EPT-1, and EPT-2, separately. Memory isolation between domains within the same EPT is achieved through the use of distinct pkeys. To achieve memory isolation between domains in different EPTs, EPK ensures that each domain's mappings only exist in one EPT. Specifically, the memory pages belonging to one isolated memory domain are tagged with the domain's pkey in the application's page table and are only mapped in the domain's EPT. Other memory pages, i.e., the global code and data of an application, are tagged with pkey-0 and mapped in all the EPTs (domain-0).

Although all the 512 EPTs are shared among different applications, it is worth mentioning that each application can construct 7,680 domains (15×512) since it has an individual guest page table.

Domain Switching. When an application thread needs to access some domain, it retrieves the permission by setting the PKRU value and choosing the corresponding EPT (switching to the domain). Switching between domains within the same EPT can be finished by executing one *WRPKRU* instruction. Switching between domains in different EPTs involves one additional *VMFUNC* instruction for EPT switching. Since both these two instructions are non-privileged, the domain switches are efficiently finished in user mode (one exception case will be explained in Section 3.2). From the perspective of programming, EPK provides easy-to-use interfaces (Section 3.3) through a user-level library for applications to create/delete domains, add/remove memory pages to/from domains, and switch domains. Applications can simply use the interfaces similar as programming on the original MPK.

Challenges. Although the idea sounds simple, there are two implementation challenges for combining the hardware features. First, how to make a VM seamlessly run with different EPTs, and how to differentiate a legal EPT violation caused by on-demand domain paging with an illegal one due to an unauthorized access? (Section 3.1). Second, given that MPK allows one thread to access multiple domains simultaneously, how to support such a flexible feature when multiple EPTs are in use (access domains mapped in different EPTs simultaneously)? (Section 3.2).

Threat Model. We assume the guest OS, hypervisor, and hardware are trusted, and EPK is correctly implemented. For the case of reducing the memory exposure time (Section 4), we assume the unreliable code may contain memory corruption bugs, which is similar to the existing work [38, 57]. For the case of isolating mutual-distrusted software components (Section 5 and (Section 6)), we assume the untrusted code or mutual-distrusted code may contain exploitable vulnerabilities like memory corruption and even use ROP to abuse *WRPKRU/VMFUNC* for illegal domain switches. So, EPK further integrates the mechanism of secure switching from previous systems [22, 46] (Section 6.1 explains how to avoid illegal domain switches). Other attacks, like side-channel attacks and rowhammer attacks, are not considered.

3.1 Extended Page Table Management

Traditionally, a VM has a single EPT that maps the GPAs of both the guest OS and applications to HPAs. Differently, EPK necessitates the creation of multiple EPTs for a VM based on two principles. *Principle-1:* GPAs that are not allocated for

[1] Domain-ID (extended protection key) = EPT-index $\times$ 15 + pkey.

memory domains should be mapped uniformly across EPTs. Thus, the VM can always run normally in any EPT. *Principle-2:* Each memory domain's GPAs should be mapped in only one EPT. As previously stated, this is for domain isolation.

GPA to Domain Association. Since the hypervisor is in charge of constructing EPTs, the first problem is how it can tell whether one GPA belongs to some memory domain or not. A straightforward solution is letting the guest OS, the GPA manager, share the information about which GPAs are allocated for memory domains with the hypervisor. Nevertheless, this entails non-trivial modifications to both the hypervisor and the guest OS. An alternative solution is to divide the whole GPA space into two halves and allocate GPAs for domain memory from one half, allowing the hypervisor to easily determine whether a GPA belongs to a domain. This solution still adds a significant amount of complexity to GPA allocation in the guest OS.

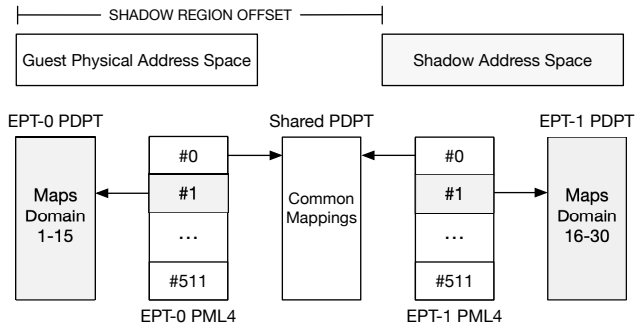


Figure 2: The EPT structures. PML4 is the top-level EPT page, and PDPT is the second top-level page.

To address this problem, EPK proposes the following design. Instead of partitioning the GPA space, EPK creates the illusion that there is a shadow address space (GPA) in the VM by simply adding a fixed offset (*SHADOW REGION OFFSET*) to the GPAs allocated to memory domains, as illustrated in the top half of Figure 2. As a result, the fixed offset becomes the boundary between the GPAs for memory domains and other GPAs. Based on this boundary, EPK constructs the EPTs, as shown in Figure 2. It sets the offset to 512 GB since an EPT PML4 entry can point to 512 GB GPA range. The entire GPA space is pointed by the first entry of each EPT PML4[2], and the shadow address space is pointed by the second entry of each EPT PML4. The first PML4 entry of different EPTs points to a shared PDPT, implying that the non-domain GPA mappings are always the same in different EPTs and thus satisfies *Principle-1*. By sharing this PDPT, the hypervisor can reduce the space overhead of multiple EPTs. More importantly, it does not need to explicitly synchronize an EPT update (e.g., adding a new mapping for the guest OS) across all EPTs, which is expensive. The second PML4 entry of different EPTs points to different PDPTs for adding the GPA mappings for memory domains, which is a prerequisite of *Principle-2*.

Illegal EPT Violations. The second problem is the hypervisor cannot determine if an EPT violation (EPT fault) within the shadow address space is legal or not. Assume an application thread executes in EPT-1 while accessing Domain-1 in EPT-0, resulting in an EPT violation. The hypervisor cannot decide whether to add the mapping because it does not know which domain the faulting address belongs to, i.e., whether the GPA should be mapped in the current EPT. Simply adding the mapping regardless of semantics will violate *Principle-2*. Instead, EPK chooses to avoid any legal EPT violation within the shadow address space (except accessing domains across EPTs which will be explained in Section 3.2). Specifically, the guest OS is required to invoke one new hypercall (a hypervisor interface provided to the VM) to fill the EPT mapping when a legal domain page fault happens, which eliminates the following EPT violation. The guest OS can check the legality of a domain page fault because applications tell it the semantics of domain mappings via the corresponding interfaces (explained in Section 3.3). As such, the hypervisor only needs to add a simple hypercall to add the EPT mapping, and EPT violations within the shadow address space must be illegal. Together with the carefully designed EPT structure, *Principle-2* can be met now. Furthermore, because it avoids original VMExits caused by EPT violations, this hypercall-based solution incurs no additional overhead.

EPT-ID Access. When a domain page fault occurs, the guest OS needs to check whether the faulting thread has the access permission according to the current PKRU value and EPT-ID. However, because the domain switches are performed in user mode, the guest OS is unaware of the changes of PKRU and EPT-ID. The guest OS can directly read the PKRU register but cannot get EPT-ID (the third problem). EPK enables the guest OS to efficiently retrieve the EPT-ID by subtly mapping one special guest physical page (named EPT-ID-Page) across different EPTs. During VM initialization, the guest OS allocates the EPT-ID-Page and passes its address to the hypervisor. The hypervisor maps the EPT-ID-Page to different host physical pages in different EPTs (in different PDPTs) and stores the corresponding EPT-ID in each physical page. Therefore, the guest OS can always obtain the current EPT-ID by simply reading the EPT-ID-Page (first four bytes).

3.2 Multi-Domain Access Support

MPK supports 16 domains and allows one thread to access any of them by configuring the PKRU register. Nevertheless, it is non-trivial to support this flexible feature in EPK since there are domains across different EPTs.

Accessing multiple domains in the same EPT can still be accomplished simply by configuring PKRU. To transparently support accessing multiple domains in different EPTs, EPK

[2]For simplicity, we assume the size of the GPA space is smaller than 512 GB. The fixed offset can be adjusted to support larger GPA space.

further employs another hardware feature named VE (virtualization exception). The hypervisor converts EPT violations in the shadow address space into VEs which will be handled in the guest OS. The VE handler in the OS can switch the EPTs for one thread and thus help it to seamlessly access multiple EPTs. Specifically, when a thread needs to acquire the access permission of domains across multiple EPTs simultaneously, it needs to inform the kernel of the domain information. Suppose the thread needs to access domain-A in EPT-1 and domain-B in EPT-2 and first runs in EPT-1. As running in EPT-1, it can directly access domain-A but will trigger an EPT violation when attempting to access domain-B. Because domain-B is in the shadow address space, the corresponding EPT violation will be caught by the VE handler instead of causing expensive VMExits. Since the OS knows that the thread can access domain-B, the VE handler will switch to EPT-2 by using VMFUNC and setting PKRU to the required value. After that, the thread can be restored and continue to access domain-B. A similar procedure happens when it later accesses domain-A in EPT-2. Thereby, EPK gives an illusion that one thread can access domains in multiple EPTs at the same time.

Two points are worth mentioning. First, EPK only converts EPT violations to VEs within the shadow address space, which has no interference on the VM's original execution. Second, different from getting access to domains in the same EPT or a specific domain in other EPTs (fast path), getting access to multiple domains in different EPTs requires the kernel involvement (slow path).

3.3 System Components in Linux/KVM

EPK's prototype implementation on Linux/KVM mainly consists of three components: a user library, a kernel module in the guest OS (Linux), and a hypercall handler in the hypervisor (KVM).

Figure 3 lists the main library interfaces available to applications. The first two functions invoke the kernel module through *ioctl* to allocate and free domain IDs. *alloc_domains* can get multiple domain IDs, and the kernel module will try

```
/* Allocate domain IDs with affinity */
int alloc_domains(int num, int dom_ids[]);

/* Free domain IDs */
int free_domains(int num, int dom_ids[]);

/* Allocate a virtual memory range for a domain */
void *domain_mmap(int dom_id, void *addr, size_t len,
                  int prot, int flags);

/* Remove some mappings */
int domain_munmap(void *addr, size_t len);

/* Retrieve the access permission of a domain */
int domain_begin(int id, int prot);

/* Release the domain permission */
int domain_end(int id);
```

Figure 3: The APIs provided by the user library of EPK.

to return the domains that are located in the same EPT. This is because some domains may have affinities, i.e., they are likely to be traversed together. Properly utilizing affinity in the applications can benefit the performance. Although it is non-trivial in general, achieving locality is straight forward in some cases. For example, in Section 4.2, a simple locality-aware request dispatching scheme can make Memcached embrace the affinity benefits; in Section 5.2, simply letting one thread work on the warehouses within the same EPT is enough.

domain_mmap first invokes *mmap* and then informs the kernel module about the domain mapping information. The kernel module records the information by using Linux's *rbtree* and validates domain page faults based on it. Huge page mapping is also supported through setting the *flag* argument. The last two interfaces are responsible for switching memory domains and are purely implemented in user mode except for accessing multiple domains in different EPTs. It is also necessary to know the current EPT-ID in user mode. For example, switching domains in the same EPT requires no VMFUNC. EPK does this by reusing the EPT-ID-Page During its initialization, the library asks the kernel module to map the EPT-ID-Page as read-only into the application. Besides, a domain memory allocator based on [34] is also provided.

Since servicing invocations from applications and recording the domain-related information, the kernel module provides a routine that aids in handling domain page faults. We insert a hook in the Linux page fault handler for invoking this routine. When a page fault occurs, the page fault handler still executes as before (e.g., allocates a free page) but invokes this routine just before setting the GPA of the newly allocated page in the page table entry. The routine then checks whether the page fault occurred within the domain regions and whether it was legal. If it is a legal domain page fault, the routine updates the GPA by adding *SHADOW REGION OFFSET* to it and invokes the hypercall to fill the mapping for the updated GPA in the EPT (as described in Section 3.1). Finally, the routine returns and the page fault handler sets the updated GPA in the page table entry. Another simple hook is added to the OS schedule function (i.e., *__schedule*). It saves/restores the EPT-ID for threads of applications that use EPK. Specifically, it saves the current EPT-ID in the thread's *task_struct* when scheduling out such a thread and restores the EPT-ID with *VMFUNC* (if necessary) when scheduling in the thread. Moreover, we add the VE handler for transparently supporting flexible multi-domain access.

In KVM, besides enabling VE and VMFUNC, we extend the hypercall handler to provide two additional functions for the guest kernel module. The first is to map the EPT-ID-Page, and the second is to add the EPT mapping for the VM's shadow address space. Furthermore, to support reclaiming the pages mapped in the shadow address space, the hypervisor needs to first disable VE on the pages to reclaim and record the reclaim information When swapping back the pages, the hypervisor needs to re-enable VE on the pages. Besides these,

the hypervisor can reclaim the pages as before. Yet, this reclaiming mechanism is not supported in the current implementation of EPK.

EPK only requires minor modifications on Linux/KVM. Our prototype only adds 250 lines of code (LOC) in KVM, 13 LOC in guest OS, and 600 LOC in guest kernel module.

4 Case Study: Protecting Server Applications

Experiment Setup. All the experiments in this paper are conducted on a Dell PowerEdge R640 server with Intel Xeon Gold 6138 CPU. Hyper-threading is disabled, and the CPU frequency is fixed to 2.0GHz. The L2 TLB has 1536 entries. In Section 4 and Section 5, we implement and evaluate EPK on Linux/KVM-4.19.88 (both the guest OS and the hypervisor). The experiments are conducted in a VM (20 CPUs and 80GB memory), and the loopback network is used. All experiments use 4k memory pages without explicit statements.

Comparison Systems. Besides the native performance (run benchmarks with no isolation), we compare the performance of EPK with libmpk [38], lwC [32], and a *VMFUNC*-only solution. We evaluate libmpk in single-thread experiments since it does not support multi-threading. Since lwC is implemented on FreeBSD, we simulate its performance on Linux. Specifically, we first measure its switch cost (around 6,000 cycles, which corresponds to the reported data in Table 2 in ERIM [46] and Table 2 in lwC [32]) and then add such switch cost in the benchmarks (i.e., waiting for 6,000 cycles when switching context is needed). Note that the simulated performance will be better than the actual performance because the indirect cost of switching address space is ignored. We also implement a *VMFUNC*-only solution that provides one memory domain in one EPT and leverages *VMFUNC* for domain switches. The experiment of libmpk is conducted in host, while all the other systems run in the VM.

4.1 Micro-benchmarks

Domain Num	*3*	*4*	*8*	*15*	*16*	*32*	*64*
libmpk(128 pages)	184	184	186	188	12,991	13,148	13,048
VMFUNC	350	831	830	836	834	849	830
EPK	97	97	100	101	111	115	162

Table 4: The average cost (in cycles) of domain switches.

We leverage different solutions to create multiple memory domains and evaluate the domain switching cost (shown in Table 4). The test program initially runs in domain-0 (not counted in the domain number) and switches between the created domains in order (sequential access). The number of iterations is 100,000 and we measure the average cost. libmpk's switch cost gets much higher when the domain number is above 15 (domain-0 takes one protection key). Besides, its switch cost is severely influenced by the size of protected memory. When each domain contains 128 pages, its switch cost becomes more than 10,000 cycles if the domain number exceeds 16, which is even 100× slower than EPK. With the domain size increasing, its switch cost will enlarge due to more page table updates during key eviction, as shown in Table 2. In contrast, the switch time of the other two approaches is immune to the domain size.

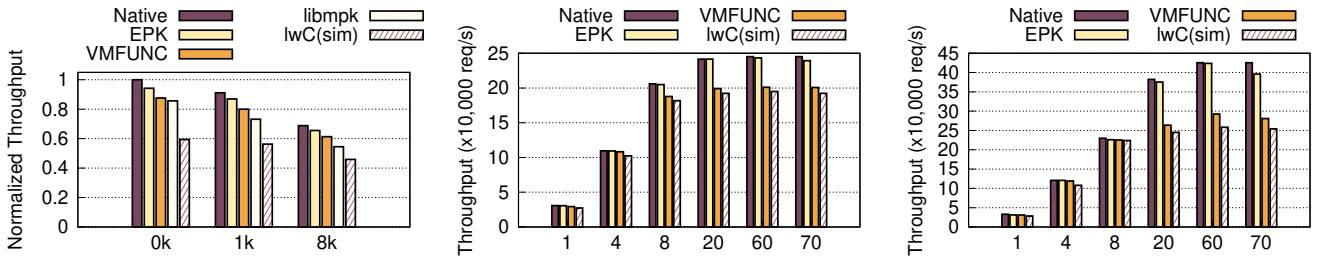
The *VMFUNC*-only solution uses one EPT for domain-0, and its switch cost is about 350 cycles which mainly comes from two *VMFUNC* instructions when the domain number is no more than 3 (the total EPT number is no more than 4). However, its cost increases to around 830 cycles when the domain number exceeds 3. This is because TLB entries are tagged with EPT base addresses, and the involvement of more EPTs may decrease the TLB hit rate. Specifically, accessing the same memory page in different EPTs generates different TLB entries and then may exceed the capacity of the corresponding TLB set. EPK shows the lowest average switch cost since most switches are based on *WRPKRU*. When the domain number is less than 16, it outperforms libmpk because the latter one involves virtual protection key management (although no key eviction). When the domain number exceeds 60, the average cost of EPK increases since there are more than 4 EPTs.

Yet, in the *worst case*, EPK needs both *WRPKRU* and *VMFUNC* for switching to one domain, and takes around 860 cycles for traversing domains like the above, which means the performance of EPK may converge with the *VMFUNC*-only solution with random switches.

4.2 Macro-benchmarks

NGINX. Introducing intra-process memory isolation to server applications brings the potential to achieve higher security or reliability. We first apply different solutions to a widely-used web server, NGINX [9] v1.12.1, to evaluate the performance overhead. We isolate SSL session keys as ERIM [46] does (including preventing the abuse of domain switching), except that we store per-client session keys in different domains rather than in one domain. We leverage ab [1] to generate the workload: 300 clients keep sending file requests one by one. The server thread is fully loaded. The total domain number is 300 and each domain contains 5 memory pages.

Figure 4a shows the evaluation results. The throughput is normalized because libmpk is implemented on Linux 4.14.2 and the native throughput differs on Linux 4.14.2 and 4.19.88 (EPK) (while KPTI is disabled on both, other mitigations on CPU vulnerabilities are key factors). EPK imposes overhead from 4.3% to 5.8% compared with native and outperforms other solutions. The overhead of the *VMFUNC*-only solution varies from 11.0% to 12.4%. Notice that the NGINX serving thread handles client requests in order. Thus, most domain



(a) NGINX. X-axis: each request size. (b) Memcached SET. X-axis: the number of users. (c) Memcached GET. X-axis: the number of users.

Figure 4: (a) shows the performance of protecting session keys in NGINX web server. (b) and (c) show the performance of isolating different users in Memcached (omit libmpk since it lacks the support of multi-threading).

switches in EPK need no EPT changing, making EPK outperform the *VMFUNC*-only solution. When storing each session key in an individual context in lwC, the overhead is 37.1% on average merely due to the explicit cost of domain switches. The overhead incurred by libmpk ranges from 14.5%-18.9% (23.4% to 33.2% if in the virtualization environment) due to the involvement of page table modifying and TLB flushing. In the cases of infrequent switching and small domain size (5 pages), libmpk will not lead to too much overhead.

Memcached. We evaluate Memcached [8] 1.6.9 and use libMemcached as the client library in this experiment. Memcached is a well-known key-value store and usually runs as a multi-thread server application. Arbiter [49] suggests that it is preferred to isolate data from different clients in Memcached for security-sensitive cases. Like Arbiter, we enable Simple Authentication and Security Layer (the SASL configuration) in Memcached and then isolate data stored by different clients. Besides, we slightly modify the request dispatching scheme of Memcached so that the requests from one client are always dispatched to the same worker thread for leveraging the domain affinity provided by EPK. The worker thread switches to the client's corresponding domain before handling a request and exits that domain before sending back the reply. We create a different number of client threads, and each of them uses libMemcached for sending SET/GET requests. The sizes of key and value are 32 bytes and 256 bytes, separately. There are four worker threads (default configuration) on the server-side, and the clients will be evenly partitioned to them. In this experiment, the max domain number is 70 and each domain contains about 2,000 memory pages.

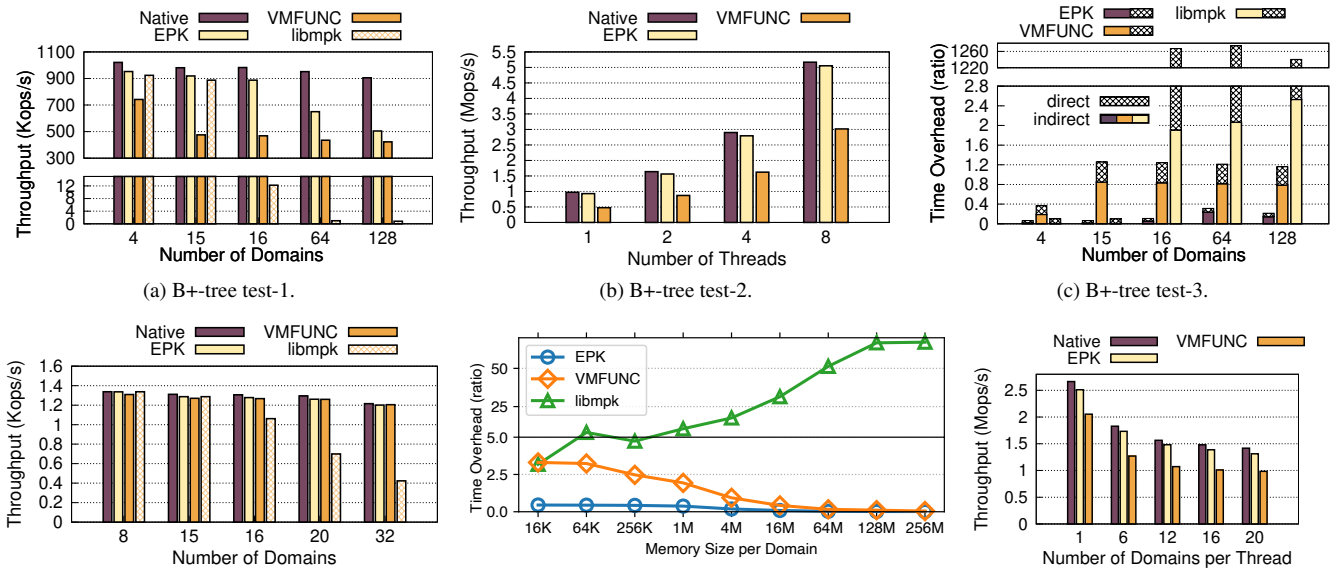
Figure 4b and 4c show the throughput of Memcached. As before, lwC leads to the highest overhead due to its expensive switch cost. When the client number is no more than 60, EPK incurs at most 0.7% overhead on the throughput of SET operations. The overhead on the throughput of GET operations is slightly higher (up to 2.9%) because the GET operations are lighter than the SET ones. The extremely low overhead is because no EPT switches happen on the critical path. EPK allows each worker thread to create 15 domains in one EPT, and thus four worker threads can handle 60 clients (60 domains) without switching EPTs. When the number of clients exceeds 60, the overhead of EPK becomes larger because some worker threads need to handle requests from more than 15 clients, and then EPT switches happen.

In contrast, the *VMFUNC*-only solution incurs a much larger overhead, i.e., up to 17.9% and 34.0% overhead for the throughput of GET and SET operations, separately. The overhead mainly comes from TLB misses, as explained in Section 4.1. For validation, we further carry out two experiments marked as *VMFUNC-test-1* and *VMFUNC-test-2* in Table 5. The former one is that the worker thread switches to the target EPT and immediately switches back before handling a request. So, all the requests are handled in EPT-0. The latter one is that each worker thread always switches to EPT-1 for handling requests. So, all the requests are handled in EPT-1. Both of them show close-to-native performance, and the TLB miss number is not significantly enlarged. However, the *VMFUNC*-only solution causes many more TLB misses and then leads to the highest overhead. The overhead of TLB miss in NGINX is not obvious because the worker thread of NGINX only switches to other EPTs when accessing session keys while the worker thread of Memcached executes most logic in different EPTs.

	Throughput (×10K req/s)	**dTLB/iTLB misses**
Native	24.5	1 / 1
VMFUNC-test-1	24.4	1.1 / 2.4
VMFUNC-test-2	24.0	1.2 / 2.4
VMFUNC	20.1	9.5 / 29.1

Table 5: The throughput and TLB misses (normalized) when evaluating Memcached SET operation with 60 clients.

Since libmpk does not support multi-thread, we evaluate libmpk in Memcached with a single worker thread. When there are 60 domains, the overhead of the above test exceeds 80%, which is significantly higher than that in NGINX because each domain contains about 2,000 pages.



(a) B+-tree test-1. (b) B+-tree test-2. (c) B+-tree test-3.

(d) Linked List test. (e) Hashtable test. (f) TATP benchmark.

Figure 5: B+-tree (a) Single-thread and random access, (b) Multi-thread and each thread operates on 15 domains, (c) Time breakdown when single-thread and sequential access. (d) Linked List (low switch frequency). (e) Hashtable benchmark (different domain memory size). (f) TATP benchmark. Omit libmpk (single-thread support) in the multi-threaded cases.

5 Case Study: Isolating NVM Data

To embrace the low access latency of NVM, applications usually map NVM into the address space and access it through load/store instructions. Bringing intra-process memory isolation to protect NVM data (e.g., reducing the data exposure time) has also been investigated by recent work [56, 57]. In this section, we evaluate the benchmarks similar to existing NVM studies [23, 33, 56, 57], using DRAM as NVM.

5.1 Data Structure Benchmarks

We first experiment on B+-tree. We map each B+-tree in an individual domain and create different numbers of threads to do lookup or insert operations (the ratio is 1:1 and other ratios show similar performance trends). Domain switches occur before and after an operation. Each tree initially has 500,000 key-value pairs, and each tree node has up to 32 child nodes. In this experiment, the max domain number is 128 and the size of each domain is about 128MB.

The overhead of lwC in the above benchmarks is always around 80% because one B+-tree operation takes just about 2,100 cycles.

Figure 5a shows the throughput when a single thread operates on a *randomly* selected tree (i.e., randomly switching to a domain). If the domain number is less than 16, EPK and libmpk bring about 7% and 11% overhead, individually. When the domain number exceeds 15, libmpk introduces unacceptable overhead (throughput drops by 99.8%) due to the substantial cost of key eviction (as the domain size is not small), and EPK can outperform it by two orders of magnitude. Note that the kernel version has minor effects on the native performance since this benchmark rarely issues system calls. The *VMFUNC*-only solution incurs 27% overhead when the domain number is 4. Compared to EPK, its higher overhead comes from two sources: one is *VMFUNC* is slower than *WRPKRU*; the other is more TLB misses (its dTLB and iTLB misses are 1.34× and 3.34× of EPK's, respectively). When the domain number increases to 64 and 128, EPK's overhead also increases to 32% and 44% because more EPTs and EPT switches are required. Specifically, when there are 64 domains, 78% of domain switches in EPK involve EPT switches. If accessing different domains sequentially instead of randomly, EPK's overhead is below 10% (3% for huge page) when the domain number is no over 60.

Figure 5b shows the performance when there are multiple threads on different cores and each thread accesses 15 different domains. EPK's overhead remains below 5% as the thread number increases, which is significantly lower than that of the *VMFUNC*-only solution (41% to 51%).

We further analyze the overhead of the three approaches in terms of the time cost (the part that exceeds the native time): the switching time (direct cost) and the rest time overhead incurred by the pollution on CPU internal structures including TLBs and caches (indirect cost). The experiment is one thread operates on the tree in each domain sequentially to complete a fixed amount of operations. Figure 5c presents the breakdown. The *VMFUNC*-only solution brings about 1.2× time overhead when the domain number exceeds 8. Its indirect cost remains around 0.8× because the TLB miss rate is almost stable. The page table updating operations of libmpk leads to both high direct cost and indirect cost (not only incurs TLB misses

but also leads to intensive cacheline pollution). EPK causes 0.23× (0.13× for huge page) indirect cost when there are 64 domains due to more than 4 EPTs, which is still better than others, and causes much lower cost for fewer domains.

Note that the domain switch frequency is proportional to the throughput in the presented benchmarks. We also conduct an experiment on Linked List (Figure 5d): each list is separated into one domain, and one thread performs 10 operations (search, insert, delete) in a random list for each time. The switch frequency is less than 1,400 times per second. libmpk still introduces 65.1% performance overhead when there are 32 domains and each domain is 256 MB. The other two approaches cause unnoticeable overhead.

Last, Figure 5e shows the overhead (in terms of the time cost) of different approaches as the domain size increases. In this experiment, each hash table resides in one domain (32 domains), and one thread keeps performing an operation (search or insert) in one random domain. We gradually increase the domain size by adding more buckets/key-value pairs in each hash table. The overhead of libmpk increases as the domain size grows, as expected, whereas the overhead of EPK and the *VMFUNC*-only solution decreases because the native performance decreases when more memory involves. Specifically, when each domain is 256 MB, the overhead of the latter two are 1.3% and 4.7%, respectively.

Virtualization Cost. Virtualization brings performance overhead to applications, especially when the working set is large and TLB misses are frequent. For example, when the domain size in hash table is configured as 16KB or 128MB, the virtualization overheads are 2.1% and 9.0%, respectively. When a VM application uses EPK, the virtualization cost is not accounted on EPK. Otherwise (in bare metal), the virtualization cost should be included in the overhead of EPK. Nevertheless, a thin virtualization layer instead of a full-fledged hypervisor can minimize the virtualization cost [22].

5.2 OLTP Benchmarks

TATP [43] is an online transaction processing (OLTP) benchmark. In the experiment, we use the above B+-tree as the data store and create four threads to execute transactions (three read-only and three read-write ones). We store a fixed amount of initial data in different domains, and each thread switches to the corresponding domain before executing one transaction. The max domain number is 80 and the size of each domain is 512MB. Figure 5f presents the throughput as the domain number for each thread increases from 1 to 20. The native throughput is in a decreasing trend along with the increase of domain number because more data weakens the cache locality. EPK's overhead is within 7%, while the *VMFUNC*-only solution incurs up to 32% overhead. We also run single-thread TATP with libmpk. Similar to B+-tree test-1, the overhead of libmpk is over 99% when the domain number exceeds 15.

TPC-C [18] is another OLTP benchmark in which there are multiple warehouses. We isolate different warehouses as well as their associated data in different domains. The max domain number is 128 and the size of each domain is 400MB. According to its specification, 7.2% of the transactions update multiple warehouses simultaneously. There are also four threads executing the transactions. When each thread operates on less than 16 domains, EPK achieves almost the same throughput as the native (0.6% overhead). The overhead is lower than that in TATP because the transactions in TPC-C are more heavyweight. When each thread operates on 32 domains, the overhead of EPK becomes 3.2% as VEs are triggered for supporting transparent multi-domain access. The other approaches are infeasible in this experiment due to the lack of the support of multi-domain access.

6 Case Study: Boosting IPCs in Microkernels

6.1 HyBridge

Different from monolithic OSes which run all the OS modules in the kernel-level, microkernels leave minimal functionalities in the kernel while running all other OS modules (referred to as system servers below) such as file systems, network stacks, and device drivers into separated user-level processes. Inherently, microkernels embrace better security and fault isolation, but leads to non-negligible communication cost at runtime. Specifically, since system servers are user-level processes, the interactions between two servers or between an application and a server require inter-process communication (IPC). In contrast, on monolithic OSes (e.g., Linux), the interaction between two OS modules only requires function calls, and the interaction between applications and the OS can be as fast as about 150 cycles (*syscall* and *sysret*). So there has been a long line of research to reduce the cost of IPC to bridge the performance gap between microkernels and monolithic OSes.

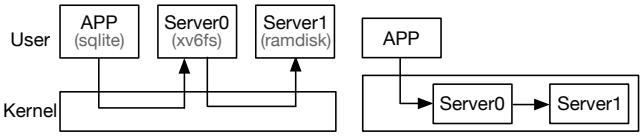


Figure 6: Traditional IPC flow on microkernels is shown on the left, and IPC with UnderBridge is shown on the right.

A most recent IPC design called UnderBridge [22] retrofits Intel MPK to optimize (synchronous) IPC. For reducing the cost of IPC between an application and a server, it pulls system servers from user-level processes into the kernel address space as shown in Figure 6. Besides, it leverages Intel MPK to ensure the isolation between system servers in the kernel, and the IPCs between them are based on *WRPKRU* and thus greatly optimized. However, due to the limitation of MPK memory domains, it can only run limited system servers in the kernel and accelerate IPCs to them (issue-1). Also, although it can reduce the privilege switches during IPCs between ap-

plications and servers, the page table switches are still needed because it requires a separate kernel page table (issue-2).

Since EPK can construct even thousands of isolated memory domains efficiently and enable fast domain switch at user-level, we propose EPK-based HyBridge for boosting IPCs for microkernels, which is inspired by UnderBridge while fixing the two issues of UnderBridge. As shown in Figure 7, system servers run at user-level, and each one exclusively takes one or more memory domains for holding its own memory, including code, data, stack, and heap. Thus, one system server cannot access others' private memory, just like when they are isolated in different processes while IPCs are based on domain switches.

Cross-server IPC. The cross-server IPCs only happen between system servers that need to interact with each other. For example, a file system communicates with a disk driver while a network stack does not. This also matches the *domain affinity* in EPK. Therefore, the microkernel can run the related system servers in the same EPT. When two servers establish an IPC connection, the microkernel will map an IPC gate, i.e., a piece of code, for them. During an IPC invocation, the gate will transfer the control flow from the caller to the callee. Specifically, it saves the caller's execution states, then executes *WRPKRU* to switch to the callee's domain, and restores the callee's execution states. Similarly, it does the reverse procedure when the IPC returns. HyBridge also allows two servers to share memory for exchanging data by assigning a free memory domain to them, e.g., shared memory domain 4 in Figure 7.

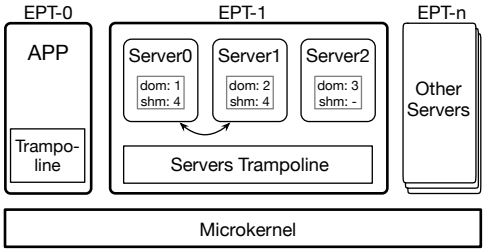


Figure 7: The overview of HyBridge. The numbers after colons are domain IDs. Shared memory is short as shm.

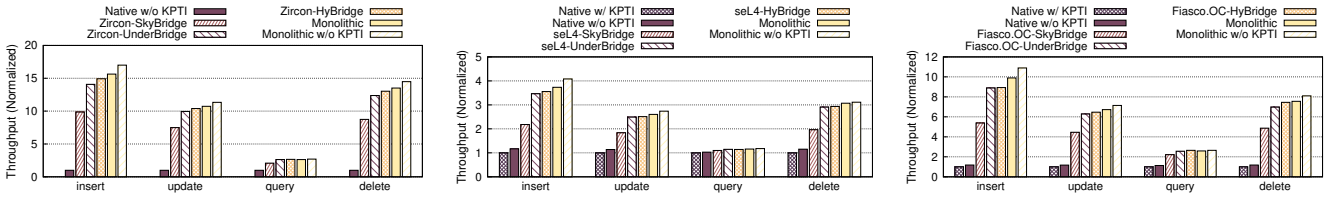
Application-to-server IPC. Applications execute in different processes (in EPT-0) just like before while several system servers can run in one process (across one or more EPTs), which means each application has a unique guest CR3 (GPA) while multiple servers share one. Since an application and a server run in different EPTs, the IPCs between them need EPT switching. HyBridge attaches a trampoline in the EPTs for running servers and maps the trampoline into an application when it asks for establishing an IPC connection with some server. The trampoline plays the role of the IPC gate and uses *VMFUNC* to switch between the caller and the callee. Though *VMFUNC* can directly switch EPT, it does not change guest CR3. However, for an application-to-server IPC, the caller and callee use different CR3 (CR3-App and CR3-Server). So, besides mapping the trampoline, HyBridge also maps CR3-App (GPA) to the HPA of CR3-Server in the server's EPT during the IPC establishment. In this way, the HPA mapping for the guest CR3 is transparently changed after executing *VMFUNC*, i.e., the guest page table is switched from the application to the server. When an application invokes an IPC, the trampoline saves the caller's execution states (executes in EPT-0), executes *VMFUNC* (switches the EPT), and restores the callee's execution states (executes in server's EPT).

Security Enforcement. Besides memory isolation, HyBridge employs additional security mechanisms to achieve the same security guarantee as original microkernels. Compared with original IPC designs, HyBridge makes an untrusted system server have two more potential attack vectors. One is that a server may bypass the memory isolation by maliciously executing *WRPKRU* or *VMFUNC* and then access others' memory. The other is that a server may issue arbitrary IPCs to other servers by maliciously executing the trampoline code without the corresponding capabilities.

HyBridge eliminates the two attack vectors as follows. First, it utilizes binary scanning and rewriting to ensure that each server contains no *WRPKRU* or *VMFUNC* instructions during binary loading. Meanwhile, it adds sanity checks in the IPC gates for ensuring the argument of *WRPKRU* is legal, which is similar as ERIM [46]. So, a compromised or malicious server cannot illegally execute these two instructions to retrieve unauthorized memory permissions even with return-oriented programming (ROP). Second, HyBridge uses a token-based mechanism to authenticate IPC invocations as SkyBridge [37] does. Considering control flow hijacking, trampolines can be executed arbitrarily or it is even possible to jump into the middle of the trampoline, i.e., using *VMFUNC* to switch to any EPT. Although they cannot be misused to break the memory isolation, an untrusted server may issue arbitrary IPCs by invoking them. To prevent this, HyBridge lets a server generate a random 64-bit token for a registered client (another server or an application) when building the IPC connection, and a client needs to pass the token during IPCs for authentication. The server only serves the IPC requests with legal tokens, so the problem of arbitrary IPCs can be avoided. Moreover, HyBridge also prevents the occurrence of *VMFUNC* in applications by scanning and rewriting the binary code. Thus, an application can only switch to system servers through the mapped trampoline.

6.2 Experiments

We implement HyBridge on three well-known microkernels, Zircon [4], seL4 [10], and Fiasco.OC [3], to assess its effectiveness. Besides, we also compare it with SkyBridge [37] which runs system servers in different EPTs and implements kernel-bypass IPCs based on VMFUNC, and UnderBridge [22]. We deploy the thin virtualization layer from SkyBridge while applying extensions needed by three IPC de-



(a) Zircon. (b) seL4. (c) Fiasco.OC.

Figure 8: Normalized throughput of SQLite3 on different microkernels. KPTI is short for kernel-page-table-isolation.

signs. We evaluate the performance of SQLite3 v3.23.0 [11], a database application, after applying different IPC mechanisms on different microkernels. For severing SQLite3, we run two system servers, a file system named xv6fs [16] and a RAMdisk. When SQLite3 operates a storage file, it will first invoke the xv6fs server by an application-to-server IPC; then, the xv6fs will access RAMdisk through cross-server IPCs. We also simulate the performance of a monolithic kernel by running system servers in the kernel and connecting them with function calls.

Figure 8a, 8b and 8c present the normalized throughput on the three microkernels. The native performance of each microkernel is set as the baseline. Because Zircon has the slowest native IPC among the three microkernels since it includes scheduling overhead in IPCs, HyBridge can provide the highest speedup for it, i.e., more than 9× speedup for three database operations. The performance improvement of query operations is relatively small because SQLite3 has an internal cache of recent data and may handle the queries without issuing IPC requests. For seL4 which optimizes IPC performance extensively, HyBridge can also improve the throughput (except query) to more than 2.5× of the native.

Besides, HyBridge can outperform SkyBridge by up to 66% because most IPCs issued from SQLite3 to xv6fs involve multiple cross-server IPCs between xv6fs and RAMdisk, whereas the cross-server IPCs are more lightweight in HyBridge. Specifically, an cross-server IPC takes 110 and 437 CPU cycles in HyBridge (*WRPKRU*-based) and SkyBridge (*VMFUNC*-based), respectively. In this benchmark, HyBridge only shows slightly higher performance than UnderBridge since cross-server IPCs dominate, while it has more advantage over UnderBridge in the application-to-server IPC (e.g., 527 vs. 723 CPU cycles when implemented on our research microkernel, ChCore [22]) owing to no CR3 changing.

7 Other Related Work

Many studies [15, 19, 27, 36, 42, 48, 58] leverage instruction instrumenting to achieve memory isolation, which may incur non-trivial overhead. Many other studies [25, 27, 32, 35, 39] utilize the memory management unit (MMU) to check memory accesses efficiently. Specifically, they divide a process into different compartments and assign each one an individual (extended) page table. However, switching between compartments requires (extended) page table switching, which can be costly when the cross-boundary invocation is frequent. Twizzler [14] is a pioneer data-centric OS for NVM and uses EPT/VMFUNC to create different memory domains for NVM isolation. Differently, EPK focuses on solving the challenges of combining MPK and EPT/VMFUNC and outperforms a VMFUNC-only solution. Besides, recent work [28, 51] harnesses hardware features like Supervisor-Mode Access Prevention (SMAP) or underused intermediate privilege levels (Ring1 and Ring2 on x86) to achieve efficient intra-process memory isolation. Yet, they can only provide two isolated memory domains.

Prior work [21, 47, 52, 54] also proposes architecture designs to facilitate efficient intra-process memory isolation, which, however, is not achievable on commodity machines. PLB [53] proposes architecture changes which differs from Intel MPK for supporting scalable domains but requires virtually indexed cache which may cause performance issues. Besides Intel, both ARM (ARMv7) and AMD propose similar features of memory domains and face the same scalability problem of the domain number. The basic idea of EPK is feasible to be extended to them as they also support 2-stage address translation. Yet, for efficiency, hardware-assisted fast switching (currently commercially unavailable) of stage-2 page table is needed on the two architectures.

An orthogonal study [17] shows that some system calls can be used to break the MPK isolation, so the OS may need to be aware of MPK in the future or the applications needs to incorporate other mechanisms like system call filtering [12].

8 Summary

This paper presents EPK which first combines the usage of MPK and hardware virtualization features to achieve scalable and efficient intra-process memory isolation. The case studies demonstrate various potential usages of EPK.

9 Acknowledgement

We sincerely thank the anonymous shepherd and reviewers for their insightful suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206 and No.U19A2060), Huawei, and STCSM (No. 21511101502). Yubin Xia is the corresponding author.

References

[1] https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] The 20 best linux apps ever. https://helpdeskgeek.com/linux-tips/the-20-best-linux-apps-ever/.

[3] Fiasco.oc repository. https://l4re.org/download/snapshots/.

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

[5] The heartbleed bug. https://heartbleed.com/.

[6] Intel optane technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

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

[8] Memcached. https://www.memcached.org.

[9] Nginx. https://nginx.org.

[10] sel4 repository. https://github.com/seL4/seL4.

[11] Sqlite. https://www.sqlite.org/index.html.

[12] Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022. USENIX Association.

[13] A. Ahmad, S. Lee, P. Fonseca, and B. Lee. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 647–660, New York, NY, USA, 2021. Association for Computing Machinery.

[14] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller. Twizzler: a data-centric os for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020.

[15] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.

[16] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. 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.

[17] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, Aug. 2020.

[18] T. P. P. Council. http://www.tpc.org/tpcc/. *TPC Benchmark C*.

[19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. 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.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. 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.

[21] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, Baltimore, MD, Aug. 2018. USENIX Association.

[22] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, July 2020.

[23] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.

[24] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. 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.

[25] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 393–405, New York, NY, USA, 2016. Association for Computing Machinery.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. 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.

[27] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. 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.

[28] H. Lee, C. Song, and B. B. 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, pages 1441–1454, New York, NY, USA, 2018. ACM.

[29] H. Lefeuvre, V.-A. Bădoiu, c. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu. Flexos: Making os isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 79–87, New York, NY, USA, 2021. Association for Computing Machinery.

[30] J. 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.

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

[32] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. 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.

[33] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[34] R. Liu and H. Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–6, 2012.

[35] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. 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.

[36] S. McCamant and G. 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.

[37] Z. Mi, D. Li, Z. Yang, X. Wang, and H. 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.

[38] S. Park, S. Lee, W. Xu, H. Moon, and T. 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.

[39] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xmp: Selective memory protection for kernel and user space. In *Proceedings of 41st IEEE Symposium on Security and Privacy*, S&P '20, 2020.

[40] V. A. Sartakov, L. Vilanova, and P. 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*, ASPLOS 2021, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[41] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, Aug. 2020.

[42] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. 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.

[43] N. Simo, W. Antoni, m. Markku, and R. Vilho. http://tatpbenchmark.sourceforge.net/. *Telecom Application Transaction Processing Benchmark*.

[44] M. Sung, P. Olivier, S. Lankes, and B. 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.

[45] D. 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.

[46] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. 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.

[47] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *2014 ACM/IEEE 41st Inter-*

national Symposium on Computer Architecture (ISCA), pages 469–480. IEEE, 2014.

[48] R. Wahbe, S. Lucco, T. E. Anderson, and S. 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.

[49] J. Wang, X. Xiong, and P. Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 361–373, USA, 2015. USENIX Association.

[50] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. *ieee symposium on security and privacy*, 2020.

[52] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, Sept. 2016.

[53] J. Wilkes and B. Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. In *Technical Report HPL–92–55*. Hewlett-Packard Laboratories, Mar. 1992.

[54] E. Witchel, J. Cates, and K. 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.

[55] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 159–172. USENIX Association, July 2020.

[56] Y. Xu, Y. Solihin, and X. Shen. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 987–1000, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Y. Xu, C. Ye, Y. Solihin, and X. Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692. IEEE, 2020.

[58] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. 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.