VDom: Fast and Unlimited Virtual Domains on Multiple Architectures

Ziqi Yuan
Zhejiang University
Hangzhou, China
yuanzqss@zju.edu.cn

Siyu Hong
Zhejiang University
Hangzhou, China
hongsy@zju.edu.cn

Rui Chang*
Zhejiang University
Hangzhou, China
crix1021@zju.edu.cn

Yajin Zhou
Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

Wenbo Shen
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Kui Ren
Zhejiang University
Hangzhou, China
kuiren@zju.edu.cn

ABSTRACT

Hardware memory domain primitives, such as Intel MPK and ARM Memory Domain, have been used for efficient in-process memory isolation. However, they can only provide a limited number of memory domains (16 domains), which cannot satisfy the compelling need for more isolated domains inside the address space of a process. Existing solutions to virtualize memory domains are either intrusive (need the modification to existing hardware), or incur a large performance overhead.

In this paper, we propose *VDom*, a fast and scalable memory domain virtualization system that supports unlimited memory domains. *VDom* leverages separate address spaces to provide an unlimited number of virtual domains, and optimizes related memory management operations. To map virtual domains to hardware domains, we design a domain virtualization algorithm, which manages address spaces and domain maps for threads to efficiently access other domains that are unmapped in the current address space. According to our evaluation on real Intel and ARM platforms, on real-world server applications (httpd and MySQL), *VDom* incurs less than 2.65% performance overhead, which is lower than the overheads of the state-of-the-art software approaches (libmpk and EPK). In random domain access tests, *VDom* is comparable to EPK and has significantly higher efficiency than libmpk.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Systems security**.

KEYWORDS

Operating System, Software Security, In-Process Isolation, Memory Domain Virtualization

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575735

ACM Reference Format:
Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2023. VDom: Fast and Unlimited Virtual Domains on Multiple Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575735

1 INTRODUCTION

Applications tend to run many mutual distrust compartments in the same address space, leading to the compromise of the whole application if one compartment is compromised. Accordingly, in-process memory isolation with efficient domain switch [13, 26, 33, 42, 43, 55, 67, 68] has been proposed to mitigate the issue.

Among a variety of in-process isolation software and hardware primitives, memory domain offers user applications a domain-based and per-thread view of memory pages, e.g., one thread can only access a limited number of memory pages inside the process's address space. With the prevalence of two representative hardware memory domain primitives, i.e., Intel Memory Protection Key (MPK) [4] and ARM Memory Domain [1] (for 32-bit programs only), there has been a long stream of research on leveraging domains for software compartmentalization [30, 37, 56, 57, 75], data vaults [19, 32, 36, 52, 59], JIT code protection [48, 53], and data race detection [12].

Though the hardware memory domain primitives can provide efficient in-process isolation, they can only support a limited number of memory domains. For instance, both MPK and ARM Memory Domain only support 16 domains. Additionally, some OS kernels reserve domains for special memory regions such as kernel and IO, leaving even fewer domains for user-space applications. At the same time, user applications are requiring more isolated memory domains inside the address space (§3.1), either because more third-party libraries can coexist in a process or parallel server applications need a larger number of simultaneously accessible domains to protect sensitive data, such as cryptographic keys, user credentials, and critical data spilled on stacks. Besides, the prevalence of data plane libraries and in-memory databases that directly operate on devices and persistent memory objects (PMO) in user space requires scalable privilege separation as well.

To circumvent the above issue, researchers have recently proposed memory domain virtualization systems that are roughly falling into the following three categories: hardware approaches [22, 72], hypervisor-based approaches [29, 50], and disabled-page-table-entry-based approaches [17, 48, 57]. Hardware-based approaches modify processor designs to support up to 1024 domains per process. Though they are usually the most efficient solution, the intrusive hardware modification hinders their wider adoption. Hypervisor-based approaches (e.g., EPK [29]) combine Intel VMFUNC [5] feature with MPK. They use VMFUNC instead of MPK when more domains are required, or utilize every 16 protection keys in each extended page table (EPT). Though VMFUNC allows virtual machine (VM) guests to directly switch between up to 512 EPTs without trapping, it still takes more cycles than MPK. Worse still, each VMFUNC switch slows down as the total number of EPTs increases. Besides, due to nested paging, IO virtualization, and software complexity, running applications in VMs incurs extra overhead [25, 60]. Disabled-page-table-entry-based approaches keep a per-process virtual to physical domain map, or allocate each processor core a unique domain. When more domains are required, they evict an old domain, disable page table entries (PTE) or page middle directories (PMD) of the evicted memory, and flush the TLB entries. This (e.g., libmpk [48]) can incur more than 70% overhead in server applications due to waiting for free domains and excessive process-level TLB flushes (§3.2).

Therefore, how to provide a fast and scalable memory domain virtualization approach is still an ongoing research effort. With such an approach, we can provide a large number of domains with low performance overhead so that real-world applications can enforce finer-grained in-process isolation. In other words, we want to solve the following research question: *How to efficiently virtualize the hardware memory domain primitives for an unlimited number of isolation domains*?

In this paper, we propose *VDom*, a fast and unlimited memory domain virtualization solution. We first analyze (§3) the demand for more domains and locate the two root causes of the performance overhead of the slow disable-page-table-entry-based approach, i.e., busy waiting for free domains and excessive TLB flushes. To avoid such overhead, *VDom* is designed (§5) to use separate address spaces and the domain virtualization algorithm for better performance on scalable isolation. Specifically, each address space offers additional hardware domains and is tagged by an address space identifier (ASID). The virtualization algorithm automatically groups threads in distinct address spaces and reduces unnecessary TLB flushes. Thus, *VDom* securely and efficiently isolates software components in monolithic applications that access many mutually distrusted objects.

We implemented a prototype of *VDom* on both Intel and ARM architectures, and evaluate its compatibility, security, and efficiency. The evaluation (§7) indicates that *VDom* efficiently offers scalable virtual memory domains. It incurs a minimal overhead (less than 2.65%) on server applications, and has significantly higher efficiency than libmpk on random domain access tests. Moreover, the evaluation shows *VDom*'s compatibility with existing OS kernels and memory domain sandboxes.

In summary, this paper makes the following main contributions.

- **New findings**. We find that a large number of domains are needed in real-world applications, and the challenges to provide fast domain virtualization are how to reduce the overhead caused by waiting for free domains and excessive TLB flushes.
- **Fast domain virtualization with unlimited domains**. We leverage separate address spaces to map different groups of virtual domains to the hardware domains in each page table. We propose a domain virtualization algorithm that switches page global directory, and puts a thread in a private address space before domain evictions to reduce unnecessary TLB invalidation.
- **Real platform evaluation**. We implement a prototype for X86 and ARM based on Linux, and evaluate *VDom* on real platforms. The results show *VDom* is fast, compatible, and secure when providing unlimited domains. On server applications, the overheads are less than 2.18% and 2.65% on X86 and ARM, respectively. On random domain access tests, *VDom* is as efficient as EPK and much faster than libmpk.

2 BACKGROUND

Memory Domain Primitive. Memory domain is an extension to page-based memory permission. It provides the domain-based thread-local view on different page groups in the same address space. PTEs in different page table levels and TLB entries are tagged with the domain identifier of the pages, while the access permissions to pages of different domains are stored in a per-core (hardware thread) register. During memory access, the processor gets the domain identifier of the virtual address, checks the access permission to that address in the register, and raises an exception if any violation is detected. Intel, ARM, and IBM Power memory domains vary in granularity (4KB on Intel and Power, 2MB on ARM), scalability (16 domains on Intel and ARM, 32 domains on Power), types of access permissions, and privilege (user-space on Intel, kernel on ARM and Power) to write permission registers.

Address Space Identifier. In mainstream operating systems, each process has a whole virtual address space. When a core switches from one process to another, its TLB entries should be invalidated to avoid the stale address translation of the prior process. This leads to frequent TLB flushes. To avoid such overhead, address space identifier (ASID), also called process context identifier (PCID) for X86, is added to TLB entries and compared with the identifier in the page table base register when the processor looks up TLB for address translation. Currently, Linux kernel supports the address space identifier feature for X86 and ARM.

3 MOTIVATION

3.1 Apps Need More Isolation Domains

After studying previous usage scenarios of the memory domain and other heavier isolation primitives, we are motivated by the compelling need for fast and scalable memory isolation.

Libraries. Applications depend on many libraries that include vulnerable code and in-library secrets. Prior work [76] shows an average npm package implicitly depends on around 80 other packages. Many compartmentalization papers [34, 45, 61, 62] selectively isolate the libraries to keep the security-performance balance. We studied the number of libraries in different types of programs. To list a few, vscode, synaptic, team-viewer (desktop) use 78, 82, 61 libraries; ghost, strider, libvirtd (server) use 131, 76, 76 libraries;

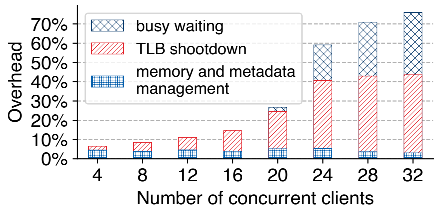


Figure 1: Overhead breakdown of libmpk on httpd that isolates each OpenSSL key with a unique memory domain.

curl, network-manager (utility) use 43, 58 libraries. These depended libraries may be vulnerable. Among 81 libraries used by chrome, vulnerabilities, such as CVE-2021-33560 in libgcrypt and CVE-2021-43527 in libnss, are found in more than 16 libraries. Also, the least privilege principle requires libraries to protect each secret (e.g., keys) with a private domain. Thus, even if the code is exploited in one domain, unrelated secrets are still inaccessible. Applications may generate many unrelated secrets in these libraries. For example, in servers using OpenSSL, more than 8,000 keys are allocated to respond to 4,000 requests.

Threads. Inter-thread privilege separation protects the private stacks and per-thread sensitive data, enhancing stability [58] and security. Prior work leverages memory domains to protect the stacks [32] and per-thread credentials [57]. To boost the throughput, many servers, such as Apache and MySQL, utilize thread pools containing tens or hundreds of threads and run them in parallel.

Persistent memory. Persistent memory objects (PMO) are frequently used as non-volatile files to store private user data and Linux supports up to 1024 file descriptors in a process. Memory corruption on PMOs is long-lived across processes and causes more severe damage than corruption on DRAM. In prior work, each unrelated PMO owns a private domain for fine-grained access control [72], requiring scalable isolation.

3.2 Slow Memory Domain Virtualization

Among the prior domain virtualization approaches, the disabled-page-table-entry-based approach does not involve hardware modification or additional VM overhead. However, current work of this approach has a massive performance drop, losing the efficiency of the hardware memory domain primitives. To observe the causes of the high overhead, we take libmpk [48] as an instance. When more than 16 domains are required, it evicts the free virtual domain for the incoming one. During the eviction, libmpk disables the pages of the evicted domain by `mprotect` with `PROT_NONE`. If all mapped domains are used by other threads, libmpk has to wait for them to release a domain. We analyze a libmpk's evaluation case with higher concurrency. We protect each private key in OpenSSL in a separate 4KB domain and run httpd with 25 server threads to transfer 16KB data. Figure 1 shows the overhead breakdown. Busy waiting and TLB shootdowns across all cores running httpd make up most of the slowdown as concurrency scales up.

4 THREAT MODEL

VDom is a kernel-user co-design that provides scalable user-space isolation domains. Therefore, the kernel and *VDom* APIs are assumed to be trusted. Also, we assume that the underlying hardware is correctly manufactured. The loading and initialization of user-space programs are also inherently trusted in *VDom*.

Untrusted components in user-space programs may attempt to arbitrarily read or write memory. *VDom* guarantees that unauthorized read and write on protected memory pages never succeed. Specifically, if a thread has not claimed its permission on a virtual domain, any attempt to access pages assigned with that domain fails. Address space integrity is enforced. Thus, malicious syscall invocations that modify PTE domain bits are intercepted. Also, *VDom* shows compatibility with memory domain sandboxes [32, 64] that mitigate unsafe permission register updates, kernel-based confused deputy attacks [21], and control-flow hijacking that illegally updates the permission register. Since how the application code invokes API to change permission on a domain depends on the concrete trust model, sanitizing the call sites and arguments of *VDom* API is out of scope.

Physical attacks, IO attacks, side-channel attacks, and micro-architectural attacks are beyond our scope.

5 DESIGN

VDom intends to virtualize the memory domain primitives with the following requirements:

- **Unlimited domains**: To meet the demand for a large number of domains, domain virtualization must provide unlimited domains, ensuring that a thread can always obtain a new virtual domain regardless of the limited number of physical domains (unless the domain identifier integer overflows).
- **Low overhead**: Domain virtualization must incur negligible overhead because the hardware primitives are famous for efficiency. In general cases, like server applications, virtual domains should be faster than existing methods such as VMFUNC-based approach.

Since busy waiting and TLB flushes are major overheads, we propose the key idea of grouping threads into separate ASID-tagged address spaces with private domain maps. **Unlimited domains** means that a thread can always allocate a domain. If the thread requires fewer domains than an address space can provide (e.g., 16 on Intel), it can obtain a physical domain from its separate address space without racing against threads in distinct address spaces. Otherwise, it can switch to another address space, or evict an old virtual domain and remap the new one. To achieve **low overhead**, first, we choose the lightweight memory domain primitives. Second, extra TLB miss caused by separate page tables incurs lower overhead than that in VMs. Third, compared to normal thread switches, switching between ASID-tagged address spaces incurs the minimal overhead of page global directory (pgd) updates without TLB flushes. Finally, when a thread that exclusively owns its address space remaps a virtual domain, only local TLB invalidation is needed.

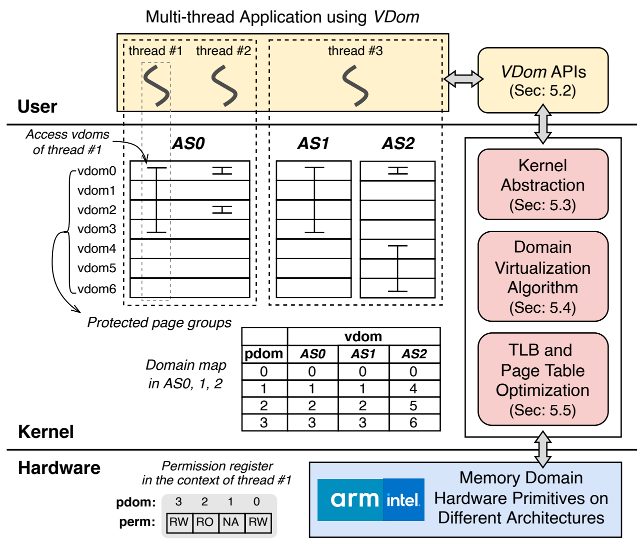

Multi-thread Application using *VDom*
thread #1
thread #2
thread #3
VDom APIs (Sec: 5.2)
User
Access vdoms of thread #1
AS0
AS1
AS2
vdom0
vdom1
vdom2
vdom3
vdom4
vdom5
vdom6
Protected page groups
Kernel Abstraction (Sec: 5.3)
Domain Virtualization Algorithm (Sec: 5.4)
TLB and Page Table Optimization (Sec: 5.5)
Domain map in AS0, 1, 2
vdom
pdom AS0 AS1 AS2
0 0 0 0
1 1 1 4
2 2 2 5
3 3 3 6
Kernel
Hardware
Permission register in the context of thread #1
pdom: 3 2 1 0
perm: RW RO NA RW
arm intel
Memory Domain Hardware Primitives on Different Architectures

Figure 2: *VDom* architecture overview.

5.1 *VDom* Architecture

Our concrete goal is to map unlimited virtual domains (**vdom**) to physical domains (**pdom**) in separate address spaces and provide fast domain switch with low overhead.

To this end, we design *VDom*, as demonstrated in Figure 2. Supported by similar architectural memory domain primitives and per-core permission registers, applications can isolate mutually distrusted page groups in unlimited numbers of virtual domains through *VDom* APIs (§5.2). Threads are grouped into separate address spaces (denoted as AS0 to AS2 in the figure) for fast access to an unlimited number of domains. Such address spaces maintain private domain maps to bookkeep the relation between vdoms and pdoms. The *VDom* kernel introduces two major abstractions to denote these virtual domains and address spaces (§5.3). We propose a domain virtualization algorithm to automatically manage threads and address spaces (§5.4). When threads require more virtual domains than an address space can accommodate, the algorithm balances the minimal cost of switching address spaces and modifying page tables along with local-only TLB invalidation. We leverage ASID to achieve lower overhead. We also present several optimization strategies for TLB and page table manipulation (§5.5).

5.2 *VDom* APIs

VDom offers user-space processes APIs listed in Table 1. To utilize *VDom*, `vdom_init` is first called. An application obtains a unique vdom for a frequently-accessed (e.g., basic libraries and master keys) or common memory region by `vdom_alloc`. A process isolates a group of pages in the vdom via `vdom_mprotect`. For hardware memory domain, changing the access rights on pdoms in the per-thread permission register completes a domain switch. To virtualize the permission register, *VDom* introduces a per-thread array called virtual domain register (VDR), every 2 bits of which represents the access right to memory protected by the corresponding vdom. A thread invokes `vdr_alloc` to limit the number of separate address

Table 1: *VDom* APIs and description.

API	Description
vdom_init()	Initialize *VDom* for the process.
vdom_alloc(freq)	Allocate a frequently-accessed or common vdom.
vdom_free(vdom)	Free the vdom for the process.
vdom_mprotect (addr, len, vdom)	Assign the process's memory pages containing any part within [addr, addr+len-1] with the vdom.
vdr_alloc(nas)	Give the thread a permission register, and limit the number of address spaces that the thread can efficiently switch between.
vdr_free()	Free a thread permission register.
wrvdr(vdom, perm)	Write the calling thread's permission on vdom.
rdvdr(vdom)	Read the calling thread's permission on vdom.

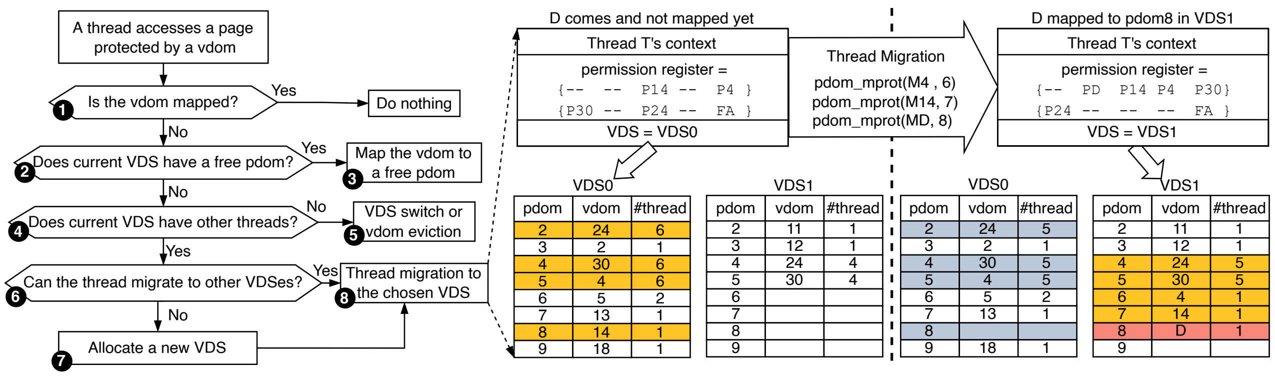
spaces it owns at most, and exclusively owns a VDR before calling `rdvdr` or `wrvdr` to read or update the permission on a vdom. In addition to Intel MPK's full access, write disable (WD), and access disable (AD) permissions, *VDom* introduces the pinned type. A pinned vdom is access-disabled but less likely to be evicted in the hybrid least-recently-used (HLRU) eviction policy (§5.5). `vdom_free` frees a vdom, while `vdr_free` releases the VDR of the calling thread.

On top of the APIs that share similar semantics to the Linux protection key (or memory domain) counterparts, *VDom* supports thread-local memory protection via the per-thread VDR and the underlying permission register, i.e., PKRU on Intel or DACR on ARM. According to the values of local VDRs, all threads in a process independently have their permissions on different vdoms. Moreover, frequent domain switches within a thread are efficiently supported by updating the permission bits in VDRs. Unlike syscalls that update the process address space (e.g., `mprotect`, `mmap`), a thread only changes the access rights of itself via `wrvdr`.

5.3 Kernel Abstraction

VDom kernel maintains the metadata of vdoms and separate address spaces via two abstractions, per-process virtual domain metadata (VDM) and per-address-space virtual domain space (VDS). VDM maintains the vdom allocation bitmap and efficiently searches pages protected by a given vdom. In particular, to balance memory space and efficiency, VDM has a hierarchical structure called virtual domain table (VDT), whose last-level entries point to chained virtual memory areas protected by the indexing vdom. VDS, which represents a separate address space, is responsible for mapping multiple vdoms to limited pdoms in its private page table and storing additional context for memory management.

As the number of isolated domains grows, *VDom* automatically groups threads into distinct VDSes. Guided by the domain virtualization algorithm, these VDSes generate private domain maps based on the domain access sequence of their threads, but share all unprotected memory regions. Specifically, address translation is shared across VDSes for all virtual addresses, including those of protected memory areas mapped to different pdoms in separate page tables. Thus, though threads in each VDS have a separate domain map on protected pages and can access distinct domains in parallel, cross-thread synchronization and process-level memory operations are supported without any application modification. In the OS kernel, *VDom* allocates a descriptor for each VDS to bookkeep the pgd



VDS0 (before)

pdom	vdom	#thread
2	24	6
3	2	1
4	30	6
5	4	6
6	5	2
7	13	1
8	14	1
9	18	1

VDS1 (before)

pdom	vdom	#thread
2	11	1
3	12	1
4	24	4
5	30	4
6		
7		
8		
9		

VDS0 (after)

pdom	vdom	#thread
2	24	5
3	2	1
4	30	5
5	4	5
6	5	2
7	13	1
8		
9	18	1

VDS1 (after)

pdom	vdom	#thread
2	11	1
3	12	1
4	24	5
5	30	5
6	4	1
7	14	1
8	D	1
9		

Figure 3: *VDom* memory domain virtualization algorithm workflow (left) and a concrete example of thread migration (right). In the right subfigure, thread T migrates from VDS0 to VDS1 when activating vdom D. In the permission register, P*x* is the permission to vdom*x*. FA means full access right. M*x* in pdom_mprot denotes pages protected by vdom*x*. #thread in the domain map denotes the total number of threads in a VDS that accesses the vdom.

and domain map. Since pdoms are fewer than vdoms, the domain map is indexed by pdom and stores the (pdom, vdom) pairs to avoid sparsity. Furthermore, the descriptor contains a CPU bitmap and a unique context identifier, aiming to trace cores executing threads in the VDS for minimal TLB shootdowns and leverage ASID for better performance in contemporary OS kernels.

5.4 Domain Virtualization Algorithm

The domain virtualization algorithm divides threads into partially shared address spaces for **unlimited domains** and fewer TLB flushes (**low overhead**). When several threads in the same VDS require more vdoms than the pdoms in the address space, the algorithm creates a new VDS and migrates a thread to it. If a thread in a VDS needs more vdoms, it is switched to another VDS, or vdom eviction happens. The input event of the algorithm is defined as a thread T accessing the memory MD protected by vdom D. We define that a pdom is free if no vdom is mapped to it. Evicted pages in a VDS are mapped to a predefined access-never pdom.

Workflow. Figure 3 (left) demonstrates the flow chart of the domain virtualization algorithm. If vdom D is not mapped in the current VDS ❶, D requires a free pdom. If thread T stays in a VDS with free pdoms ❷, *VDom* allocates a free pdom, maps D to that pdom ❸, and updates the domain map. Leveraging the underlying memory domain hardware, the OS kernel assigns PTEs of all present pages protected by the vdom with the selected pdom. If the current VDS runs out of free pdoms, the algorithm first checks the number of threads sharing the current VDS. If T is the only thread in its VDS ❹, VDS switch or vdom eviction happens ❺. If the current VDS is shared across several threads, T checks every existing VDS ❻ to decide if T can migrate to that VDS ❼. If no existing VDS can accommodate T, a new VDS is allocated and initialized to be the destination of the thread migration ❽.

Thread migration. A concrete example of thread migration ❽ is shown in Figure 3 (right). Assume the hardware offers 10 pdoms, with pdom0 as default and 1 as access-never. Before migration, T shares VDS0 with another 5 threads and vdom0, 4, 14, 24, 30 are mapped to pdom0, 5, 8, 2, 4 in VDS0, respectively. The physical permission register of T stores the access rights on mapped vdoms. Assume that D is not mapped in VDS0. From ❶, *VDom* reaches ❻ for lack of free pdoms. Then, the algorithm tries to accommodate T in VDS1, in which vdom0, 24, 30 are already mapped and pdom6, 7, 8, 9 are free. T fits in VDS1 by mapping vdom4, 14, D to pdom6, 7, 8. The highlighted entries in domain maps show how the domain maps and numbers of accessing threads are updated during migration. Additionally, the permission register of T is synchronized to stay consistent with the new domain map. For instance, permission bits `P24` are moved in the permission register in line with the remapping of vdom24 from pdom2 to 4. To complete the migration, pages protected by vdom4, 14, D are assigned with pdom6, 7, 8 in their PTEs in VDS1, and the pgd of T is switched.

VDS switch or domain eviction. Though switching pgd is faster than modifying PTEs, pgd switch may not always be optimal. For example, if a thread swaps strings in two protected PMOs in distinct VDSes, each memory access incurs a pgd switch, which is prohibitively expensive. In contrast, if the PMOs are mapped in the same VDS by domain eviction, the following PMO access only involves updating the permission registers. Therefore, *VDom* balances the overhead of vdom eviction and VDS switch when the thread T asks for more domains ❺. In particular, if the vdom D is frequently-accessed (defined in `vdom_alloc`), or some other mapped vdoms are accessible according to the permission register, the domain virtualization algorithm evicts an old domain and remaps D in the current VDS. The kernel walks VDT to efficiently find all memory areas protected by the old vdom and disables related PTEs by the access-never pdom. Such eviction solely flushes local TLB entries. Otherwise, *VDom* first tries to find D in other VDSes of T and switches pgd if successful. If not, another VDS is allocated to make the most of additional page tables within the number defined in `vdr_alloc`, or vdom eviction happens in a chosen VDS of T.

5.5 TLB and Page Table Optimization

VDom optimizes TLB and page table operations to efficiently support multiple address spaces and occasional domain evictions. *VDom* kernel keeps track of the ASIDs and CPU bitmaps of all VDSes to reduce excessive inter-processor TLB flushes. Moreover, TLB range flush instructions ensure invalidation of minimal virtual address ranges during the eviction. However, some processors spend proportional time to the range size. Thus, to balance the cycles consumed by range flushes and later TLB misses, *VDom* invalidates all entries in a given ASID if the evicted vdom protects a large chunk of memory.

Domain evictions on large ranges involve proportional numbers of PTE updates. To further reduce eviction overhead, we optimize page table manipulation. If a vdom of continuous non-huge pages that range across 2MB is evicted, the kernel directly disables page middle directory (PMD). Accessing the illegal PMD triggers page fault. Later, when the page group is to be remapped, the HLRU policy first checks which pdom (denoted as pdom*x*) the vdom is mapped to last time. If the vdom that currently maps to pdom*x* is inaccessible and not pinned, it is evicted. Otherwise, the page group finds the victim by LRU. Note that if all vdoms are pinned when access-disabled, *VDom* strictly follows LRU policy within a VDS. By remapping a large domain to the same pdom*x*, *VDom* saves cycles on updating numerous PTEs.

6 IMPLEMENTATION

To test the scalability and efficiency of our domain virtualization approach on prevalent architectures, we implement a *VDom* prototype for Intel and ARM based on Linux kernel version 5.17. Programmers can utilize *VDom* when setting the `HAS_VDOM` flag in Kconfig. Summarized by git diff, 4290 lines of code across 68 source files are modified in Linux. The APIs contain about 400 lines of code across 4 files. Among the added code, about 70% is architecture-independent, while X86 and ARM have different page fault handlers and ASID management methods. The following subsections introduce the implementation details, including Linux extended data structures (§6.1) and VDS management (§6.2) for Intel X86 and ARM. Since MPK efficiently updates the permission register in user space, the secure API call gate is implemented on Intel (§6.3).

6.1 Linux Data Structures

To describe VDSes, we modify `task_struct` and `mm_struct` in Linux. Basically, the per-thread `task_struct` has two extra fields: a pointer to the VDS the thread stays in and a pointer to the VDR of the thread. When the thread can efficiently switch between several VDSes (determined by `nas` in the `vdr_alloc` API), an array of pointers to VDSes and their corresponding values in the architectural permission register are also recorded in `task_struct`. In Linux, `mm_struct` represents one address space. In contrast, we decide to use it for all VDSes. This design is superior to using a separate `mm_struct` for each VDS in that only page tables require extra synchronization. Simply using the `clone` syscall with deep page table copy also needs maintenance when changing the process state, including its `mm_struct`. In *VDom*, `mm_struct` stores the VDM of the process and chains all VDSes together via a linked list. Additionally, `vds_struct` keeps track of each VDS. Apart from the pgd, domain map, CPU bitmap, and context identifier, TLB generation is added in X86 `vds_struct` for the X86-specific ASID management in Linux.

6.2 VDS Memory Management

We modify page table management and page fault handling routine in the kernel. Each VDS has a unique pgd but shares the same view of virtual memory except for the hardware memory domain bits in PTEs. VDS memory synchronization keeps Linux metadata of virtual memory, such as the red-black tree of virtual memory areas (VMA), consistent with the actual page tables. *VDom* updates the page tables of VDSes lazily through page fault just like demand paging when more permissions to a page are granted, while synchronizes page tables of all VDSes eagerly on permission revocation. To be specific, whenever the kernel frees the process memory, changes the protection bits of a page group, walks page tables with an operation determined by a function pointer, or when kswapd reclaims page frames, eager synchronization is activated. Note that the OS kernel changes its virtual memory metadata along with page table update. Since all VDSes share the same `mm_struct`, the original Linux code completes metadata management without extra synchronization effort.

The page fault handler deals with permission fault and VDS demand paging. Accessing unmapped domains and memory access violation trigger protection key fault and page domain fault on Intel and ARM, respectively. Linux kernel identifies the vdom of the fault address through the extended `vm_flags` in VMA and inspects the per-thread VDR. Any access violation results in `SIGSEGV`. Otherwise, pgd switch or domain eviction is triggered for the unmapped vdom. On the other hand, `handle_mm_fault` is responsible for lazy demand paging for *VDom*. Similar to SMV [33], the kernel finds the pgd of the trapped thread and updates both the VDS page table and the per-process shadow page table to keep the consistency of the process address space.

6.3 Intel APIs Protection

VDom leverages the fast user-space PKRU update instruction on Intel. Hence, to protect critical data, such as VDRs, in *VDom* APIs from memory corruption and control-flow hijacking, *VDom* uses the access-never domain (pdom1).

In user space, a VDR can solely be accessed by its owner thread in the trusted API library. During initialization, *VDom* library assigns the VDR pages with pdom1 and locks them through the whole process lifetime. A program has full access to pdom1 right after entering the API library and no access permission before returning to the untrusted user code. Moreover, `wrvdr` identifies the calling thread's VDR from a read-only page mapped by the kernel, rather than a pointer passed by an argument. Furthermore, when a thread is in the API library, some critical data such as the new permission to a vdom is spilled on the stack, which is also protected by pdom1.

VDom securely shares VDR pointer and domain map with APIs via processor core numbers. In `wrvdr` and `rdvdr`, each thread needs to efficiently get its VDR and domain map. Since `gettid` is too slow and thread-local storage (`fs`) is writable in user space, *VDom* binds each running thread to a particular processor core. During context switch and thread migration, the kernel checks the processor number. Then, the corresponding cacheline-aligned per-core entry in a shared page is filled with the next thread's pointer to VDR and its domain map. Moreover, inter-core scheduling is disabled between getting the core number and accessing the VDR pointer.

```
1  lib_entry:
2      xor  %ecx, %ecx
3      rdpkru
4      and  $0xfffffff3, %eax       # full access to pdom1
5      wrpkru
6      mov  %rsp, %rax              # spill old rsp in rax
7      mov  cpunode_seg, %edx
8      lsl  %dx, %edx               # load segment limit
9      and  cpunode_mask, %edx      # get core number
10     lea  shm_oft(%rip), %rdi     # secure sharing page
11     shl  log($64), %rdx          # offset from the page
12     add  %rdi, %rdx
13     mov  (%rdx), %rdi            # thread's VDR address
14     add  stack_oft, %rdi         # new rsp offset to VDR
15     mov  %rdi, %rsp              # switch stack
16     push %rax                    # push the old rsp
17 inline_func:
18     ...
19 lib_exit:
20     pop  %rsp                    # pop the old rsp
21     xor  %ecx, %ecx
22     rdpkru
23 make_eax_pkru:                   # arithmetic in regs ->
24     ...                          # to update active vdoms
25     xor  %edx, %edx
26     or   $0x4, %eax              # access disable for pdom1
27     mov  %edx, %ecx              # clear before wrpkru
28     wrpkru
29     and  $0xc, %eax
30     cmp  $0x4, %eax              # check is pdom1 disabled
31     jne  illegal
32     retq                         # return from old stack1
```

Figure 4: Call gate for *VDom* API library on Intel X86.

Figure 4 shows more details on API memory protection and secure sharing. Take `wrvdr` as an example. At the entrance, the call gate: (1) sets full access to pdom1 in PKRU (line3-5); (2) gets the core number through `lsl` instruction that reads the global descriptor table (line7-9); (3) gets the page for secure sharing based on `rip` (line10); (4) finds the cacheline-aligned (64-byte-aligned) entry in the page to get the VDR of the thread according to the core number (line11-12); (5) determines the stack address to switch to (line13-14); (6) switches `rsp`, pushes the old `rsp` and executes the regular prologue (line15-17). Before the gate exits to the untrusted user code, the PKRU update for the target vdom and disabling access permission to pdom1 are merged in one `wrpkru` (line23-28) to boost performance. To defend against control-flow hijacking which controls the value in `eax` before `wrpkru`, the value in `eax` is compared with the legal value (line29-31).

7 EVALUATION

This section answers the following questions:

- Is *VDom* compatible with other Linux subsystems and existing memory domain sandboxes (§7.1)?
- Can *VDom* defend against potential attacks under the proposed threat model (§7.2)?
- Does the modified kernel slow down the whole system (§7.3)?
- How much overhead do micro-operations of *VDom* incur (§7.5)?
- How fast can protected server applications and the random domain access test run (§7.6)? To better understand the performance of *VDom*, we choose one representative from each motivating aspect (§3.1) in application benchmarks.

Table 2: *VDom* ports an example from each type of defense of existing memory domain sandbox.

Example	Type	Arch
Insert watchpoint before make code pages with PKRU update instructions executable ❶	binary scan	X86
Check fixed PKRU permission before switch ❷	call gate	X86
Block unchecked read on protected memory through `process_vm_readv` ❸	syscall filter	X86 ARM

Our evaluation environments are Dell PowerEdge T440 with an Intel Xeon Gold 6230R CPU (2.10GHz 26 cores 52 threads) and 64GB memory, and Raspberry Pi 3 Model B with a 1.2GHz quad-core 64-bit ARM Cortex-A53 and 1GB memory. We test on Linux 5.17.0 with X86_64 generic Kconfig for Intel and ARMv7l raspi Kconfig for ARM. The Linux distributions for Intel and ARM are Ubuntu 18.04 and Ubuntu 20.04, respectively.

7.1 Compatibility Evaluation

Other Linux subsystems. To examine the compatibility of *VDom* with other Linux subsystems, we run the Linux Test Project (LTP) [6] on the original and our modified kernel, respectively. We pass memory management, file system, disk IO, scheduler, and IPC test suites in LTP on both kernels.

Memory domain sandboxes. Existing memory domain sandboxes monitor sensitive syscalls that change the pdom of memory pages and defend against kernel-based confused deputy attacks. To defeat such attacks in *VDom*, similar syscall filters can be applied as well and the virtual address of the trusted library is locked once loaded to enforce the sole entry of *VDom*-related syscalls.

MPK sandboxes additionally sanitize unsafe `wrpkru` by binary inspection and protect the call gate against arbitrary PKRU updates caused by control-flow hijacking. *VDom* call gate is compatible with MPK sandboxes by replacing original `wrpkru` instructions with inline `wrvdr` calls (no indirect jumps). In the call gates between every two domains, existing MPK sandboxes [32, 59, 64] compare `eax` with the fixed legal permissions. Indeed, the domain virtualization algorithm does not generate fixed maps between vdoms and pdoms. To determine `eax`'s legality, *VDom* can check the shared domain map again after `wrpkru` to find out whether the target vdoms are active and dynamically constructs the expected PKRU value.

In summary, three types of defense are taken into consideration: (1) filters that block the malicious syscalls that help escape the sandbox; (2) binary inspection that identifies unsafe `wrpkru` and `xrstor`; (3) call gate that defeats arbitrary PKRU write via control-flow hijacking. For simplicity, we choose one concrete defense from every aspect that the state-of-the-art MPK-based sandbox, Cerberus [64], summarizes to show adequate compatibility. As listed in Table 2, sandbox-enhanced *VDom* correctly handles unsafe and hijacked PKRU updates ❶ ❷, and intercepts syscalls that access protected pages directly as confused deputy ❸.

7.2 Security Evaluation

Security analysis. Typically, the security of memory domain is enforced by three software layers: (1) kernel and APIs that link the memory domain semantics to the underlying page tables and permission registers; (2) trusted user-space sandbox that prevents malicious components from escaping the protection of memory domain; (3) application that correctly invokes APIs. As a first-layer software, *VDom* must secure itself and eliminates new escapable surface.

First, *VDom* kernel keeps address space integrity against malicious remapping and reassigning vdom to protected memory. Once a virtual memory area is assigned with a vdom, the user-space application cannot reassign another vdom to this area until process termination. Second, no similar syscalls that might act as confused deputies are added by *VDom*. The third attack surface is VDS metadata and page table corruption. Since the kernel maintains all metadata related to VDSes, malicious user attackers cannot corrupt these critical data without kernel hacking. Fourth, particularly on Intel processors, X86 user-space APIs modify both VDR memory arrays and PKRU registers, which are protected via the access-never pdom and secure kernel-user sharing (§6.3). To prevent insecure `wrpkru` and `xrstor` from arbitrarily updating the permission on the access-never pdom, *VDom* can port the binary inspection from Hodor [32]. By analysis, *VDom* is secure.

Penetration Tests. Both in-thread attacks and cross-thread attacks on random vdoms are tested in a sample program with multiple VDSes. *VDom* supports no access, read-only and read-write permissions. Programs terminate immediately on both architectures once they attempt to access vdoms with AD in VDRs, or to write with WD permission, showing the efficacy of *VDom*.

In our evaluation, *VDom* is immune to X86 *VDom* user-space API VDR and stack corruption, no matter whether the attacker directly attempts to overwrite the data or firstly tries to change the memory domain flags of related pages. Filling the PKRU register with hijacked `eax` in API exit causes segmentation fault as expected, which also shows that reusing `wrpkru` never gives untrusted application code control over data in the API library. Hence, the tests indicate that *VDom* offers the same security enhancement as the original memory domain.

7.3 Performance Impact On Linux

For applications not using *VDom*, to measure the performance on our modified Linux, we run UnixBench [2] on the original and *VDom* kernel, respectively. Note that UnixBench is a microbenchmark focusing on kernel performance, which means the performance drop (if any) of modified Linux is exaggerated compared to applications. On UnixBench single-thread and parallel test suite, *VDom* has similar scores (98.5% to 101.8% marks) on X86 and ARM compared to the baseline kernels.

7.4 Performance Comparison Methodology

We use the following methodology to compare *VDom* with libmpk and EPK in the performance benchmarks on the Intel platform.

Libmpk. Libmpk fails to support multi-threading, due to data race and the lack of a per-thread view of metadata. We fix these bugs without changing the key logic and port libmpk to Linux 5.17.

EPK. We run the in-VM (simulated) EPK-hardened applications and compare the overhead against that of *VDom*-enhanced applications out of VM. The application binaries are identical.

Table 3: Average cycles of common operations.

Operation	X86 Cycles	ARM Cycles
empty API call return	6.7	16.5
empty syscall return	173.4	268.3
update PKRU or DACR	25.6	18.1
VMFUNC [46]	169	undefined
fast `wrvdr` API call return	68.8	406
secure `wrvdr` API call return	104	406
secure `wrvdr` with 4KB eviction	1,639	2,274
secure `wrvdr` with 2MB eviction	1,605	3,159
secure `wrvdr` with 64MB eviction	8,097	11,778
secure `wrvdr` with VDS switch	583	723

Since EPK is designed and implemented based on KVM [38], we set up the host using KVM in Linux 5.17 and QEMU [14]. Both applications and benchmark processes are executed in the guest Ubuntu 18.04 based on Linux 5.17 kernel. The kernels for the vanilla applications, *VDom*, and the VM have the same configurations, boot arguments (no KPTI), and are compiled by the same toolchain. We allocate 50 hardware threads and 56GB of memory to the VM, which is quite adequate for the tested applications. To explore the performance of VMs, efficient IO virtualization approaches are used. A network interface card is passed through. In fact, the loopback network used in the evaluation (but not in the real world) eliminates the overhead of network virtualization. We also pass through a Samsung solid-state disk storage device, which is identical to the one used to evaluate *VDom*, to the guest kernel via vfio-pci to achieve the best VM speed. Indeed, other user-space virtual machine monitors, such as Firecracker [11] and Cloud Hypervisor [3], exist. According to our experiment with several monitors, the tuned VM on QEMU achieves similar or the best performance.

We simulate the overhead of domain switches in EPK because the code or binary is not publicly available. As reported in EPK [29], the average cycles consumed by different types of domain switches (i.e., MPK switch and VMFUNC switch across distinct total numbers of EPTs) are compared with *VDom*'s in microbenchmarks, because EPK is evaluated on a similar Intel Xeon CPU. In applications, we simulate EPK by inserting the reported cycles in every domain switch depending on the number of EPTs and whether the switch involves VMFUNC. For instance, to simulate a VMFUNC-based switch, 350 cycles or 830 cycles are inserted into the application. Otherwise, 97 cycles are added. We achieve the accurate cycle insertion in EPK simulation by increasing a counter by one again and again until the counter reaches a predefined threshold. The deviation ranges from -11 to +5 cycles. Our methodology ignores the overhead incurred by extra TLB misses from multiple EPTs. Hence, the actual overhead should be no less than the simulated one.

7.5 Microbenchmarks

We measure the cost of (1) domain switch, (2) different access patterns on distinct numbers of vdoms, (3) memory synchronization, (4) VDS context switch.

Domain switch. The costs of domain switches in different scenarios and other related operations are shown in Table 3. X86 *VDom*

Table 4: Average cycles cost by `wrvdr` and counterparts on sequential and switch-triggering accesses of 2MB (512 pages) vdoms. X86f and X86s mean the fast and secure `wrvdr` API. X86e and ARMe mean *VDom* evicts old domains rather than switches VDSes.

# of vdoms	3	4	15	16	29	32	64	70
VDom X86f seq	70	73	82	151	121	141	138	134
VDom X86f trig	70	75	82	530	552	566	704	701
VDom X86s seq	107	104	113	183	152	171	161	166
VDom X86s trig	105	106	113	573	611	623	771	765
VDom X86e seq	69	70	82	301	1,565	1,594	1,598	1,605
libmpk seq	102	103	150	30,609	30,909	30,877	30,721	30,704
EPK seq [29]	97	97	101	111	NA	115	162	NA
EPK trig [29]	97	97	101	NA	NA	350	830	830
VDom ARM seq	406	423	491	486	536	480	490	533
VDom ARM trig	408	433	668	662	695	714	779	811
VDom ARMe seq	408	421	1,613	1,895	3,137	3,161	3,187	3,185

Table 5: The overheads of allocating and synchronizing 4KB pages across different numbers of VDSes.

# of VDSes	2	4	8	16	32
X86 overhead (%)	3.8	8.9	20.9	38.8	56.1
ARM overhead (%)	19.7	33.8	undefined	undefined	undefined

protects the user library by the access-never pdom in the secure `wrvdr` API call, while the fast call relinquishes the call gate for efficiency. All application benchmarks (§7.6) use the secure API. On ARM processors, updating the DACR register is privileged and secure. The evaluation shows the `wrvdr` API is particularly fast when the vdom is mapped in the current VDS. Moreover, the page table optimization accelerates 2MB eviction to reduce the overhead of necessary eviction on large memory domains.

Accessing domains. Our synthetic benchmark accesses various numbers of vdoms (each 2MB domain has 512 pages) according to sequential and switch-triggering patterns. Take the 64-domain benchmark as an example. The sequential pattern accesses vdoms in order (i.e., iterates from vdom0 to 63), while the switch-triggering pattern causes a VDS switch on each access via traversing vdoms with strides. For eviction-based methods (i.e., libmpk and *VDom* eviction), the sequential access has the same performance as the switch-triggering access. We also compare libmpk and EPK to show the efficiency of *VDom*. As shown in Table 4, switching VDS (without kernel page table isolation) is faster than libmpk and comparable to EPK. Thanks to ASID, the heavy TLB flush instructions are virtually not needed during address space switches. Additionally, since *VDom* can work without VMs and EPTs, when the application owns more than 3 address spaces, a VDS switch takes similar cycles to an EPT switch. Moreover, due to our optimization on page table operations, for 2MB domains, eviction in *VDom* is faster than that in libmpk.

Memory synchronization. Memory synchronization in *VDom* takes extra cycles on managing PTEs and TLB entries. To measure the overhead, we run a multiple-address-space application that progressively allocates 4KB pages. One address space holds the data, and the code in other address spaces (VDSes) immediately accesses the data after initialization. Table 5 demonstrates that the synchronization overhead is proportional to the number of total address spaces the application owns. Since the evaluated ARM processor has only 4 cores, the overhead of memory synchronization for more than 4 VDSes is dominated by the scheduler. Notice that if there is no data access from other address spaces, the cost is close-to-zero thanks to the demand paging mechanism.

Context switch. Multiple page tables in the same `mm_struct` complicate context switch. Accordingly, when switching to a process not using *VDom*, on Intel and ARM, the `switch_mm` function spends about 451.9 and 1442.1 cycles in total. Compared to the original Linux, *VDom* slows down context switch by 6% and 7.63%. Due to additional metadata maintenance, an average switch to a VDS takes 771.7 and 1545.1 cycles on Intel and ARM platforms, respectively.

7.6 Application Benchmarks

OpenSSL: isolate many in-library secrets. OpenSSL [9] is a widely-used project for cryptography operations and the TLS protocol. Memory disclosure vulnerabilities in OpenSSL, such as CVE-2011-4576, CVE-2014-0160, and CVE-2016-2176, allow attackers to breach cryptographic keys. We put each private key structure (e.g., `EVP_PKEY`) into a separate 4KB vdom when allocation. Since the physical memory consumption of the vanilla httpd is about 4.5MB, even if 256 private keys are allocated, the 21.8% physical memory overhead is acceptable. Access to a protected key is enabled when libcrypto code requires to read it and disabled right after key-related operations. In total, 207 lines of code are added to the enhanced OpenSSL library.

We test HTTPS throughput of Apache HTTP server (httpd) linking the original and protected OpenSSL with ApacheBench (ab). Both programs choose the ECDHE-RSA-AES256-GCM-SHA384 cipher suite and 1024-bit secret key. We start one httpd worker that spawns 40 threads and use Event Multi-Processing Model. In each run, we start several ab instances simultaneously to simulate different numbers of concurrent clients. Each ab instance sends 10,000 requests to httpd with a file size of 1KB, 64KB, or 128KB. Figure 5 shows the average throughput of protected httpd relative to original httpd over 10 runs with various concurrent client counts and request sizes, measured after a warm-up phase. On X86, *VDom* introduces an average of 0.12% overhead for 1KB file size, and 1.92% and 2.18% overheads for 64KB and 128KB each. On ARM, the corresponding numbers are 2.50%, 1.43%, and 2.65%. Even if all private key structures are protected in the same domain, the lowerbound overheads on Intel range from 0.86% to 1.03%. Running the unprotected httpd in VM incurs 6.03%, 7.15%, and 5.09% overheads for 1KB, 64KB, and 128KB files, respectively. On top of that, the simulated EPK introduces 6.69%, 8.17%, and 5.21% overheads when compared to the vanilla version running in the host. Our evaluation result demonstrates that libmpk is inefficient regardless of the file size. In each run, more than 80,000 vdoms are allocated in httpd. Thanks to separate page tables, *VDom* allows threads to access all domains without busy waiting.

MySQL: separate many threads. MySQL is an open-source relational database management system. It is a single-process multi-thread program, which spawns a new thread or reuses a thread in

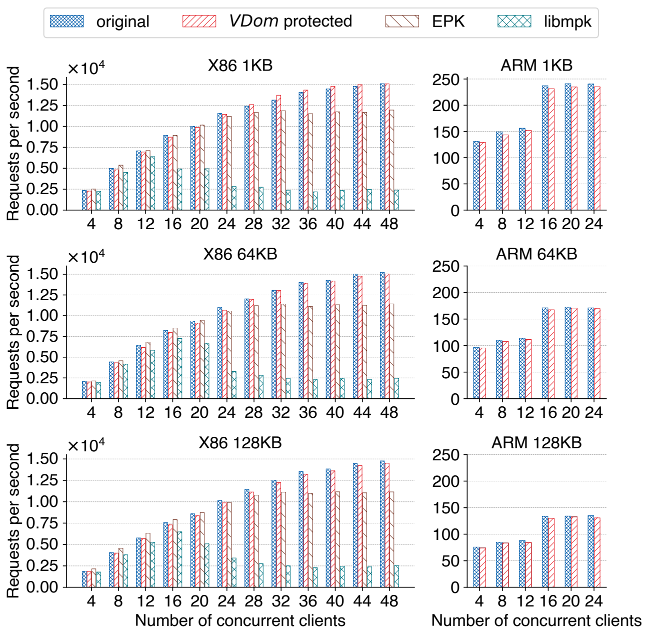


Figure 5: Throughput of original, *VDom*, EPK, and libmpk httpd on X86 and ARM.

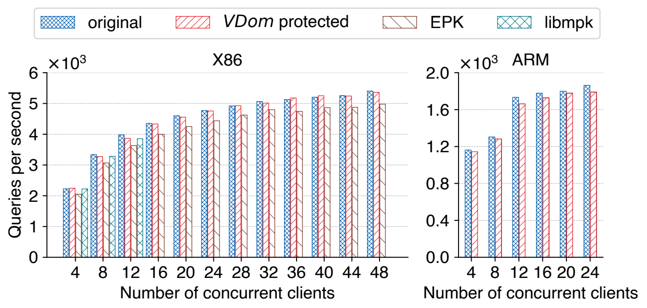


Figure 6: Throughput of original, *VDom*, EPK, and libmpk MySQL on X86 and ARM.

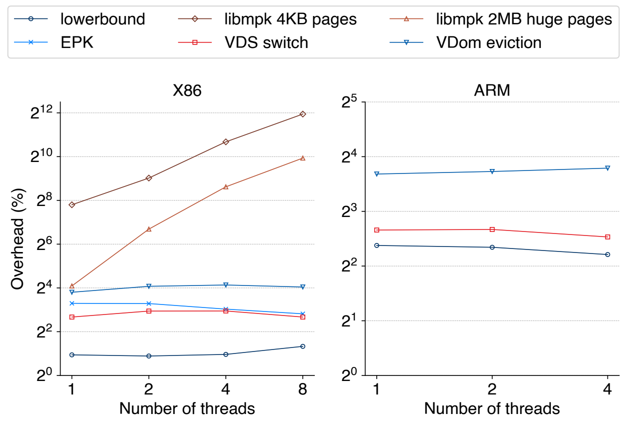


Figure 7: Overheads for String Replace on X86 and ARM.

the thread cache to handle an incoming client connection. We modify MySQL to protect the stack of each connection handler thread. *VDom* puts each thread's stack into a separate vdom, ensuring that a thread's stack cannot be accessed by other threads. This prevents compromised threads from having arbitrary access to other threads' stacks, changing their execution flow or stealing data stored on the stack. By default, MySQL allows more than 100 clients to connect simultaneously, which means that a corresponding number of connection handler threads are created in the mysqld process. We also use *VDom* to protect in-memory data for MySQL MEMORY storage engine that creates in-memory tables to achieve fast access and low latency. *VDom* ensures that the database can only be accessed by the engine. The MEMORY storage engine stores records in the data structure named HP_PTRS. Thus, *VDom* isolates all HP_PTRS structures into a vdom, and updates VDRs before and after accessing the structure in MEMORY storage engine code for least privilege. Altogether, 91 lines of code are added to MySQL.

We benchmark performance impact on MySQL using sysbench. We run the OLTP read-write script of sysbench on an in-memory database comprising 10 tables, each with 100,000 rows of data. Figure 6 shows the throughput of original MySQL and MySQL hardened by *VDom* with different concurrent client counts. At each client count, we run MySQL 10 times and take the average. *VDom* introduces an average of 0.47% overhead on X86 platform, and 2.59% overhead on ARM. In the VM environment, the vanilla MySQL suffers a 6.89% throughput loss. If we consider both VM overhead and protection, the simulated EPK has an average overhead of 7.33%. Unlike *VDom* or EPK, libmpk allows only 15 vdoms to be activated simultaneously. Since connection handler threads run in parallel, libmpk cannot provide per-thread protection for MySQL when the number of concurrent clients exceeds 14 (because one domain is for in-memory data).

String Replace: protect many PMOs. Persistent memory is non-volatile and has performance similar to DRAM when serving as main memory [35]. Data in PMO is long-lived, so the consequences triggered by illegal read or write to an attached PMO are more severe. We use a multi-PMO benchmark and protect each PMO with a vdom like the prior study [72]. The benchmark has 64 PMOs of 2MB size. Each PMO is filled with strings and each string is 512 bytes in length. Multiple threads are launched in the benchmark. Each thread randomly picks a string and performs a substring search and replacement operation on that string. When a thread needs to read the string, *VDom* grants WD permission to the corresponding PMO; when it needs to replace the substring, full access permission is granted. We evaluate the performance of both VDS switch and domain eviction approaches by limiting the address spaces each thread can access. We run the benchmark 10 times, and each run performs 4,000,000 operations for each thread.

Figure 7 shows the overheads of different numbers of threads on X86 and ARM platforms, where 2^2 on the y-axis means 4% overhead, 2^3 means 8%, etc. The line of lowerbound represents the overhead of using one physical domain to protect all PMOs. We protect 512 4KB pages in *VDom* and VM approaches. For libmpk, we try both 4KB pages and 2MB huge pages. The lowerbound overheads are 2.06% and 4.97% on X86 and ARM, respectively. In

VDom, the eviction mechanism brings average overheads of 16.21% and 13.31%, while VDS switch only slows down the application by 7.03% and 6.15%. Though the random access PMO test has a high eviction (or VDS switch) rate (e.g., each String Replace operation usually takes around 10,000 cycles on X86), the overheads of *VDom* are acceptable. In contrast, the overhead of libmpk grows with the number of parallel threads. For example, libmpk with huge pages incurs a 17.73% performance loss when running a single thread. However, the overhead increases to 977.77% if 8 threads are running in parallel. Worse still, if 4KB pages are allocated, the 8-thread libmpk introduces a 3941.95% slowdown. Although the virtualization overhead of the application is only 2% because the PMO benchmark is a simple and pure-user-space program, EPK slows down the program by 8.71% in total compared to our baseline. Thus, *VDom* is faster than libmpk and comparable to EPK in the PMO benchmark.

7.7 Limitations

In *VDom*, the underlying hardware page tables enhance security at page granularity. To protect fine-grained data, programmers have to change the memory layout. Although VDS switch without TLB flush and vdom eviction save cycles compared to two `mprotect` syscalls, they still take hundreds and thousands of cycles, respectively. Thus, unlike hardware-based approaches, frequent pgd switch and vdom eviction are expensive, which are thankfully uncommon in real-world applications. Also, our page table optimization makes a balance between performance and granularity on the virtual address. Sharing protected pages with legacy libraries may be hindered by the modified memory layout. Like other processor-side isolation-based protection, *VDom* is vulnerable to IO attacks due to the limitations of the memory management unit. Though isolated by hardware, in-memory sensitive data plaintext is vulnerable to cold boot attack [31]. Moreover, the security of *VDom* relies on the ported sandbox and policies set by programmers.

8 RELATED WORK

8.1 Memory Domain

Memory domain virtualization provides user-space applications with hundreds and thousands of domains. Shreds [17], libmpk [48], and μTiles [57] disable related PMDs or PTEs when a vdom is evicted. Domain Virtualization [72] maintains a hardware 2-dimensional permission array indexed by thread identifier and domain number. EPK [29] and xMP [50] are virtualization-based methods. However, program in VM runs slower due to IO virtualization and nested paging. Additionally, *VDom* also supports 32-bit ARM and future AMD CPUs [7] without VMFUNC.

Memory domain itself has been widely studied. ARMlock [75] and FlexDroid [54] are built on ARM Memory Domain. ERIM [59], Hodor [32], Enclosure [28], FlexOS [40], CubicleOS [51], Jenny [52], ZoFS [23] and UnderBridge [30] rely on Intel MPK for single address space isolation. Donky [53] and CODOM [63] modify hardware for secure switches and fine-grained data sharing in cross-domain calls.

8.2 Hardware-Enforced Data Protection

Traditionally, data protection is divided into randomization and isolation approaches. Among the isolation methods, Dune [13], Secage [43], Enclosure [28] and FlexOS [40] isolate compartments in distinct EPTs. However, switching rings and VMFUNC are heavier than memory domain. MemSentry [39] and BOGO [74] rely on Memory Protection Extension (MPX) [47] and aim at isolation and memory safety, respectively. HAKC [44] approximates kernel least privilege separation by isolating code and data into several partitions via ARM Memory Tagging Extension (MTE) [8] and Pointer Authentication (PA) [10], while *VDom* protects user-space applications. Researchers have also proposed more versatile and secure data protection primitives by hardware modification. CHERI [68, 69] isolates program compartments by hardware capability and fat pointers. HDFI [55] and DataSafe [18] make use of tagged memory that contains extra security information. IMIX [26] encodes permission to secure pages in instructions rather than permission registers for near-to-zero cost without thread-local flexibility. For higher entropy than today's randomization, Morpheus [27] rerandomizes secrets with a hardware churn unit frequently, while RegVault [71] extends PA for register-grained full randomization.

Apart from software attacks, data protection against malicious kernels, physical attacks, and side-channel attacks is also studied. Hence, encrypted enclaves [41, 49] and SoCs [20] are used to store secret data. These techniques often have stronger threat models than memory domain but slow down applications.

8.3 Software-Based Memory Isolation

Software-based memory isolation builds jails to prevent illegal memory access upon generic instructions and kernel abstractions, such as address masks, buffer bounds, and virtual memory. SFI [65], BGI [16], XFI [24], and Native Client [73] limit memory access in the same address space by source code instrumentation and binary rewriting. However, frequently masking addresses and checking bounds without specific secure hardware cause prohibitive overhead.

Multiple virtual address spaces are also studied. Arbiter [66], SMV [33] use per-thread page tables to achieve memory isolation between threads. Wedge [15], LWC [42] provide each compartment in a process with its own page table. However, switching between components requires kernel mediation. Secret-free hypervisor [70] utilizes separate page tables to ensure that a guest VM domain can only access its own secrets and explicitly identifies non-secret data.

9 CONCLUSION

In this paper, we present the motivation, design, and implementation of *VDom*, a fast and scalable memory domain virtualization system that provides user-space applications with unlimited isolation domains. *VDom* introduces separate page tables for scalable domains and the domain virtualization algorithm, and leverages TLB and page table optimization for performance. We implement a prototype for X86 and ARM, and port several applications. The evaluation demonstrates that *VDom* is more efficient in server applications compared to prior domain virtualization work. It offers scalable and simultaneously accessible domains. The VDSes reduce TLB shootdowns. Our generic design requires no hardware modification or architecture-dependent feature. Moreover, *VDom* enhances security as hardware memory domain primitives and has compatibility with the Linux subsystems and sandboxes. Although *VDom* is slower than hardware-based domain virtualization in some extreme cases, our evaluation shows that the efficiency on real-world applications with many vdoms allows wide adoption.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd Professor Nathan Dautenhahn for their insightful comments that greatly helped improve the presentation and evaluation of the paper. We also want to thank the anonymous artifact evaluation reviewers for their careful inspection of the artifact and guidance of *VDom*. This work was partially supported by the key R&D Program of Zhejiang Province (No. 2022C01165), the National Natural Science Foundation of China (NSFC) under Grant U21A20464, the National Key R&D Program of China (No. 2022YFE0113200). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code of *VDom* modified Linux kernel, user-space API libraries, all evaluation benchmarks, and scripts necessary to reproduce the paper's evaluation results.

Our work targets Intel X86 and ARM architectures (32-bit programs). On each architecture, 5 core experiments are included in the artifact, allowing readers to reproduce our paper's key results. Also, readers can build new research on top of the *VDom* kernel and library code.

A.2 Artifact Check-List (Meta-Information)

- **Program:** *VDom* modified Linux kernel and user-space API libraries, benchmarked with the security test, the microbenchmarks, and application benchmarks including httpd (with designated OpenSSL), MySQL, and a synthetic PMO benchmark.
- **Compilation:**
  - Intel X86: Ubuntu 18.04.6 LTS, GCC 7.5.0.
  - ARM: Ubuntu 20.04.4 LTS, GCC 9.4.0.
- **Binary:** Readers can build benchmark binaries using provided source code according to the `README.md` of the artifact. To build and install the modified kernels, the guides and configurations to enable *VDom* on the hardware platforms demonstrated in our evaluation are also included in `README.md`.
- **Run-time environment:** Linux kernel 5.17 upstream and Raspberry Pi versions. Need root access.
- **Hardware:**
  - Intel X86: Dell PowerEdge T440 with an Intel Xeon Gold 6230R CPU (2.10GHz 26 cores 52 threads) and 64GB memory, or any machine that supports Intel MPK.
  - ARM: Raspberry Pi 3 Model B with a 1.2GHz quad-core 64-bit ARM Cortex-A53 and 1GB memory, or any machine that supports ARM Memory Domain with ARMv7l ISA.
- **Execution:** Automated via provided benchmark scripts.
- **Metrics:** CPU cycles, requests/s, queries/s, execution time.
- **Output:** Results are output to the `results` directory under each experiments directory, including performance data and corresponding charts.
- **Experiments:** Microbenchmarks, httpd (with designated OpenSSL), MySQL, PMO benchmark, security test.
- **How much disk space required (approximately)?:** 100GB.
- **How much time is needed to prepare workflow (approximately)?:** 6-12 Hours.
- **How much time is needed to complete experiments (approximately)?:** 2-4 Hours (*automated*).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** The Linux kernel code uses Linux's original (GPLv2-only with the syscall exception) license unless specified otherwise. Other components that are not directly related to the Linux kernel (evaluation scripts, GCC plugins, etc) are licensed under the MIT license.
- **Archived (provide DOI)?:** 10.6084/m9.figshare.21354552

A.3 Description

A.3.1 How to access? The artifact can be downloaded from our archive.[1] The *VDom* modified kernel, user-space libraries, and all benchmarks are available in the archive.

In order to precisely reproduce the results of this paper in the evaluation section, we gave ASPLOS'23 reviewers access to our machines with pre-installed software, including an Intel server and a Raspberry Pi (specific models and configurations are listed in the previous section).

A.3.2 Hardware dependencies. *VDom* is evaluated on Intel X86 and ARM architectures:

- To run experiments on X86 architecture, a machine that supports Intel MPK is required, typically Intel Xeon Scalable Processors starting with the Skylake generation or the new Core Processors from TigerLake. Specifically, our machine contains an Intel Xeon Gold 6230R CPU (2.10GHz 26 cores 52 threads) and 64GB memory.
- To run experiments on ARM architecture, a machine that supports ARM memory domain and ARMv7l ISA is required. Specifically, we use a Raspberry Pi 3 Model B with a 1.2GHz quad-core 64-bit ARM Cortex-A53 CPU and 1GB memory.

A.3.3 Software dependencies. To use *VDom* APIs, the *VDom* modified Linux kernel and the user-space libraries need to be installed. The artifact has been tested with Ubuntu 18.04.6 LTS on Intel X86 and Ubuntu 20.04.4 LTS on ARM. All other dependencies required by the application benchmarks are detailed in the `README.md` of the artifact.

A.4 Installation

The setup of the environment and the creation of benchmarks consist of the following steps:

- Build and install *VDom* modified Linux kernel.
- Build and install *VDom* user-space libraries.
- Build benchmark binaries and launch tests.

The specific operations and configurations are shown in the `README.md` of the artifact.

[1]https://figshare.com/articles/software/VDom_Artifact/21354552

A.5 Experiment Workflow

Running the experiments is automated through the corresponding shell scripts in the artifact. There are 5 experiments in our artifact, including a security test, microbenchmarks, and 3 application benchmarks. To run the experiments, use the shell scripts under each experiment's directory:

```
$ ./run.sh
```

Once a benchmark is complete, results are output to the `results` directory under each experiment directory, including performance data and corresponding chart.

The execution flow of the scripts varies from experiment to experiment. Please refer to the `README.md` file in each experiment directory for a detailed explanation of the testing process.

A.6 Evaluation and Expected Results

Readers can directly view the data and graph in the `results` directory after running each experiment, and compare them with the results in the paper to verify if the results match.

Under the same machine configuration, the experimental results should be consistent with the results shown in the paper. On other machines, absolute values may vary, but ordering and trend should be similar.

A.7 Experiment Customization

We can use *VDom* APIs to build custom applications to test the availability, security, and scalability of *VDom*. Readers can refer to the code and shell scripts of the microbenchmarks to correctly leverage the APIs. Another simple approach to customizing an application is to remove the `-DTRY_ILLEGAL_ACCESS` definition in the shell script in `eval/sectest`.

A.8 Notes

Due to the performance limitation of Raspberry Pi 3 Model B and the large scale of MySQL, the throughput of MySQL on ARM is not stable. To obtain reliable results, readers need to check the original output of sysbench to filter stable output data, not just depend on the output of the script. Specifically, we select the results without abnormal data (close-to-zero) in the whole sysbench test period in our evaluation.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] ARM Memory Domains. https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Memory-access-control/Domains.

[2] byte-unixbench. https://github.com/kdlucas/byte-unixbench.

[3] Cloud Hypervisor. https://www.cloudhypervisor.org.

[4] Intel Architecture Instruction Set Extensions and Future Features. https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf.

[5] Invoke VM Function. https://www.felixcloutier.com/x86/vmfunc.

[6] Linux Test Project. http://linux-test-project.github.io/.

[7] Memory Protection Keys on Future AMD CPUs. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.

[8] Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety.

[9] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[10] Pointer Authentication and Branch Target Identification Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension.

[11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[12] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung 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*, pages 647–660, 2021.

[13] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[14] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX annual technical conference, FREENIX Track*, pages 10–5555. Califor-nia, USA, 2005.

[15] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008.

[16] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[18] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. A Software-Hardware Architecture for Self-Protecting Data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 14–27, 2012.

[19] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[20] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189, 2015.

[21] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426, 2020.

[22] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. SealPK: Sealable Protection Keys for RISC-V. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1278–1281. IEEE, 2021.

[23] 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*, pages 478–493, 2019.

[24] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[25] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, 2015.

[26] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018.

[27] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 469–484, 2019.

[28] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 255–267, 2021.

[29] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and Efficient Memory Protection Key. In *2022 USENIX Annual Technical Conference*

(USENIX ATC 22), 2022.

[30] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, 2020.

[31] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[32] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, 2019.

[33] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 393–405, 2016.

[34] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. *arXiv preprint arXiv:2108.03705*, 2021.

[35] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[36] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[37] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[38] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[39] 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*, pages 437–452, 2017.

[40] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–482, 2022.

[41] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.

[42] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, 2016.

[43] 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*, pages 1607–1619, 2015.

[44] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[45] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[46] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Function. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[47] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[48] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[49] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[50] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577. IEEE, 2020.

[51] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 546–558, 2021.

[52] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the 31th USENIX Security Symposium*, 2022.

[53] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694, 2020.

[54] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS*, 2016.

[55] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[56] 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.

[57] Zahra Tarkhani and Anil Madhavapeddy. μTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2004.04846*, 2020.

[58] Merve Turhan, Thomas Nyman, Christoph Bauman, and Jan Tobias Mühlberg. Unlimited Lives: Secure In-Process Rollback with Isolated Domains. *arXiv preprint arXiv:2205.03205*, 2022.

[59] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[60] Vincent Van Rijn and Jan S Rellermeyer. A Fresh Look at the Architecture and Performance of Contemporary Isolation Platforms. In *Proceedings of the 22nd International Middleware Conference*, pages 323–335, 2021.

[61] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Towards Fine-grained, Automated Application Compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 43–50, 2017.

[62] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[63] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-centric Memory Domains. *ACM SIGARCH Computer Architecture News*, 42(3):469–480, 2014.

[64] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.

[65] 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*, pages 203–216, 1993.

[66] Jun Wang, Xi Xiong, and Peng Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 361–373, 2015.

[67] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 592–607. IEEE, 2020.

[68] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[69] 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.

[70] Hongyan Xia, David Zhang, Wei Liu, Istvan Haller, Bruce Sherwin, and David Chisnall. A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1544. IEEE Computer Society, 2022.

[71] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. RegVault: Hardware Assisted Selective Data Randomization for Operating System Kernels. In *2022 59th ACM/IEEE Design Automation Conference*

(DAC). IEEE, 2022.

[72] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng 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.

[73] 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.

[74] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2019.

[75] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 558–569, 2014.

[76] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, 2019.

Received 2022-07-07; accepted 2022-09-22