True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy

Alex Markuze Adam Morrison Dan Tsafrir

Computer Science Department
Technion—Israel Institute of Technology

Abstract

Malicious I/O devices might compromise the OS using DMAs. The OS therefore utilizes the IOMMU to map and unmap every target buffer right before and after its DMA is processed, thereby restricting DMAs to their designated locations. This usage model, however, is not truly secure for two reasons: (1) it provides protection at page granularity only, whereas DMA buffers can reside on the same page as other data; and (2) it delays DMA buffer unmaps due to performance considerations, creating a vulnerability window in which devices can access in-use memory.

We propose that OSes utilize the IOMMU differently, in a manner that eliminates these two flaws. Our new usage model restricts device access to a set of shadow DMA buffers that are never unmapped, and it copies DMAed data to/from these buffers, thus providing sub-page protection while eliminating the aforementioned vulnerability window. Our key insight is that the cost of interacting with, and synchronizing access to the slow IOMMU hardware—required for zero-copy protection against devices—make *copying preferable to zero-copying*.

We implement our model in Linux and evaluate it with standard networking benchmarks utilizing a 40 Gb/s NIC. We demonstrate that despite being more secure than the safest preexisting usage model, our approach provides up to 5× higher throughput. Additionally, whereas it is inherently less scalable than an IOMMU-less (unprotected) system, our approach incurs only 0%–25% performance degradation in comparison.

Categories and Subject Descriptors D.1.3 [*Operating Systems*]: Security and Protection

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA..

ACM 978-1-4503-4091-5/16/04. . . $15.00.
http://dx.doi.org/10.1145/2872362.2872379

Keywords IOMMU; DMA attacks

1. Introduction

Computer systems face the threat of a *DMA attack* [8, 45] in which a hardware device compromises the host system using direct memory access (DMA) operations to arbitrary physical memory locations. An attacker can mount a DMA attack by (1) remotely taking over a device, e.g., by exploiting a network interface card (NIC) firmware vulnerability [20, 21], or (2) physically introducing a malicious device, e.g., by intercepting hardware shipments [3] or plugging in a malicious iPod [8, 11, 12, 19]. The attacking device can then use DMAs to gain access to the host OS [8, 11, 12, 19, 20], install a backdoor [3, 16, 47], steal sensitive data [11, 12, 17, 19, 45], or crash the system [33, 46]. An errant device [33] or buggy driver [9, 18, 23, 49] might also inadvertently mount a DMA attack.

OSes leverage hardware I/O memory management units (IOMMUs) [5, 7, 25, 30] to implement *intra-OS protection* [50] and prevent arbitrary DMAs. The IOMMU treats the address in a DMA as an I/O virtual address (IOVA) [36] and maps it to a physical address using OS-provided mappings, blocking the DMA if no mapping exists. OSes use transient IOMMU mappings that restrict device DMAs only to valid DMA targets [6, 14, 26, 39]. The OS maps a DMA buffer when a driver requests a DMA. The device now "owns" the buffer and OS code may not access it. When the device notifies the OS that the DMA has completed, the OS destroys the mapping—preventing further device access—and OS code may access the buffer again. Systems that isolate drivers as untrusted out-of-kernel components [15, 23, 35, 43] apply the same technique to protect data in the trusted kernel.

In practice, unfortunately, implementations of the above intra-OS protection design *do not fully protect from DMA attacks*. They suffer from two problems: lack of *sub-page protection* and providing only *deferred* protection.

No sub-page protection IOMMU protection works at the granularity of pages [5, 7, 25, 30]. Mapped data—e.g., network packets—is usually allocated with standard kernel

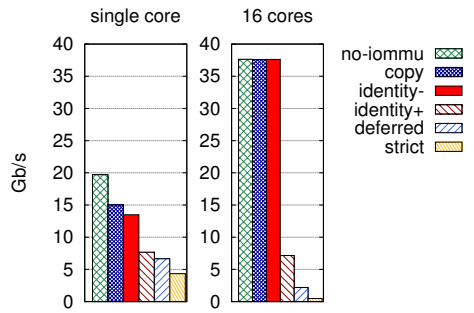


Figure 1: *IOMMU-based OS protection cost: Linux TCP throughput (1500 B packets) over 40 Gb/s ethernet, measured with single/multiple netperf instances. (copy is our intra-OS protection; identity± is recent work [42] tackling another Linux bottleneck.)*

`malloc`, which can satisfy multiple allocations from the same page [13]. Thus, sensitive data that should not be accessible to the device may get co-located on a page IOMMU-mapped for a device.

Deferred protection Unmapping an IOVA requires invalidating the IOTLB, a TLB that caches IOMMU mappings. IOTLB invalidation is expensive, e.g., requiring ≈ 2000 cycles on an Intel Sandy Bridge machine [37]. Worse, IOTLB interaction is serialized with a lock, which becomes a bottleneck for concurrent IOTLB invalidations [42]. As a result, OSes by default implement *deferred* protection, in which IOTLB invalidations are batched to amortize their cost, instead of *strict* protection that invalidates on each unmap. Figure 1 depicts the issue for Linux, our case study here, and followup work [42] that attempted to address it; other OSes make similar trade-offs [42].

Deferred protection weakens security by creating a window in which a device can access unmapped IOVAs that have since been reused. While exploiting this window is not in the scope of this paper, we remark that it is not theoretical—we have been able to crash Linux using it.

True DMA attack protection We propose an alternative intra-OS protection design addressing the two flaws above. We use the IOMMU to restrict device access to a set of permanently-mapped *shadow DMA buffers* and *copy* DMAed data to/from these buffers, thereby achieving byte-granularity protection while never needing to invalidate the IOTLB. Our key insight is that *copying is typically cheaper than invalidating the IOTLB*, for two reasons. First, in many I/O workloads DMA buffers are simply small enough that copying them costs less than an IOTLB invalidation—e.g., 1500-byte ethernet packets. Second, in multicore workloads, the IOTLB-related lock contention under strict protection significantly increases invalidation costs and makes even larger copies, such as 64 KB, profitable.

Still, copying huge DMA buffers costs more than an IOTLB invalidation. Huge buffers, however, occur in workloads such as GPU usage, whose DMA frequency is low enough that the standard zero-copy strict protection does not incur much overhead. (Basically, most time is spent on data transfer/processing.) In these cases, we copy only the sub-page head/tail of the buffer (if any), and map/unmap the rest.

iommu protection model		*sub-page protect*	*no vulnerability window*	*single core perf.*	*multi core perf.*
strict	Linux	✗	✓	✗	✗
	FAST'15 [38]	✗	✓	✓	✗
	ATC'15 [42]	✗	✓	✓	✗
defer	Linux	✗	✗	✓	✗
	FAST'15 [38]	✗	✗	✓	✗
	ATC'15 [42]	✗	✗	✓	✓
copy (shadow buffers)		✓	✓	✓	✓

Table 1: *Our contribution relative to the state of the art.*

Performance We focus on 40 Gb/s networking, as it represents demanding I/O workloads that trigger millions of DMAs per second. We evaluate our design in Linux using standard networking benchmarks. Our protection scheme achieves comparable throughput to state-of-the-art deferred protection [42]—which is less secure—and provides throughput 5× higher relative to strict protection—which is still less secure—while reducing CPU consumption by up to 2.5×.

The maximal I/O throughput that our protection scheme can scale to is inherently less than the throughput achievable in a native execution without an IOMMU, due to the CPU and memory traffic overhead incurred by copying. (For example, if the native execution requires 100% CPU to sustain the I/O throughput, our scheme's overhead would lead to reduced throughput.) We show, however, that on modern hardware our scheme *scales to the demanding 40 Gb/s rates*: it incurs only 0–25% throughput degradation and less than 20% increased CPU use.

Contributions To summarize, we make three contributions: (1) observing that copying DMA buffers is preferable to IOTLB invalidation; (2) providing a truly secure, fast, and scalable intra-OS protection scheme with strict sub-page safety; and (3) implementing the new scheme in Linux and evaluating it with networking workloads at 40 Gb/s. Table 1 highlights the differences between our newly proposed scheme and the state-of-the-art (further deails provided below).

2. Background: IOMMU-Based OS Protection

IOMMUs prevent DMAs not authorized by the OS from accessing main memory (see § 2.1). IOMMUs can provide inter- and intra-OS protection [48, 50]. Inter-OS protection is used by hypervisors to prevent a guest OS from directing DMAs at the memory of another guest OS or of the hypervisor. This form of protection uses static IOMMU mappings that reflect a guest's physical-to-host address mappings [50].

It thus does not frequently change IOMMU mappings and is not our focus. We focus on *intra-OS* protection, in which the OS protects itself from attacks by devices (§ 2.2).

2.1 IOMMU Operation

We describe the IOMMU operation in the Intel x86 architecture [30]; other architectures are similar [5, 7, 25]. The IOMMU treats the target address of a DMA as an *I/O virtual address* (IOVA) [36] and attempts to translate it to a physical address. Translations are done based on per-device *IOVA mappings* that the OS creates, which also include the type of access allowed to the mapped IOVA—read, write or both. The DMA is then routed to the physical address to which its IOVA maps; if no valid mapping exists, the DMA is blocked and the OS is notified. IOVA mappings are at page granularity; they are maintained in a per-device page table (similar to a standard MMU page table).

The IOMMU has an IOTLB that caches IOVA mappings. The OS must therefore *invalidate* any IOTLB entries associated with an IOVA mapping after modifying or destroying the mapping. The IOMMU supports both *global* invalidations, which invalidate every cached mappings, and invalidations of specific IOVA pages (i.e., at page granularity). The OS invalidates the IOTLB by posting an invalidation command to the IOMMU's *invalidation queue*, a cyclic buffer in memory from which the IOMMU reads and asynchronously processes IOMMU commands. The OS can arrange to be notified when the invalidation completes. It does this by posting an *invalidation notification* instructing the IOMMU to update a memory location after invalidating the IOTLB, which allows the OS to busy wait on this location.

2.2 Intra-OS Protection via the DMA API

OSes base IOMMU-based intra-OS protection on a *DMA API* [6, 14, 26, 39] that a driver must use to authorize an expected DMA. The basic idea is that, before programming a device to issue a DMA to some buffer, the driver invokes a `dma_map` call to map the buffer in the IOMMU. Once the DMA completes, the driver invokes a `dma_unmap` call to destroy the mapping.[1] We describe the Linux DMA API [14, 36, 40] in detail; other OSes are similar [6, 26, 39].

• `dma_map`: This operation receives a buffer's address and size, as well as desired device access rights (read, write or both). It allocates a sufficiently large IOVA interval *I* from the device's IOVA space, creates a mapping from *I* to the buffer in the device's IOMMU page table, and returns *I*'s starting IOVA. From this point, the device can access the buffer and so the OS/driver are not allowed to modify the buffer and should not expect to read valid data from it.

• `dma_unmap`: This operation receives an IOVA that maps to a DMA buffer. It unmaps the buffer in the IOMMU by looking up the IOVA interval *I* containing the IOVA and then removing the mappings of *I* from the device's IOMMU page table. *I* is deallocated, so that future `dma_map` calls can reuse it. From this point, the driver/OS can access the buffer again, and the device should not attempt to access any IOVA in *I*.

• `dma_alloc_coherent` of shared buffers: DMA buffers through which a driver posts commands and receives responses, such as DMA descriptor rings and "mailbox" data structures, need to be accessed simultaneously by both driver and device. The `dma_alloc_coherent` operation facilitates this—it allocates memory for such use and maps it in the IOMMU. The driver/OS can access this memory but must do so defensively, since the device can also access it. `dma_alloc_coherent` allocates memory in pages, and so the pages of a buffer it allocates are never shared with any other allocation. Buffers allocated with `dma_alloc_coherent` are freed using `dma_free_coherent`, at which point they are unmapped as in `dma_unmap`.

2.2.1 Strict vs. Deferred Protection

To prevent a device from accessing an unmapped buffer, `dma_unmap` must invalidate the IOTLB after removing an IOVA mapping from the IOMMU page table. IOTLB invalidation is an expensive operation for two reasons: First, the hardware is slow—invalidation can take $\approx$ 2000 cycles to complete [37]. Second, the IOMMU invalidation queue is protected by a lock, which becomes a bottleneck for concurrent IOTLB invalidations [42].

As a result, *strict* protection—in which an `dma_unmap` invalidates the IOTLB—adds prohibitive overheads for high-throughput I/O workloads such as 10–40 Gb/s networking (§ 6). These workloads generate millions of DMAs per second, and often run on multiple cores because the desired throughput cannot be achieved by a single core [34].

To avoid these overheads, OSes by default trade off security for performance and use *deferred* protection, in which the IOTLB is invalidated asynchronously *after* the `dma_unmap` returns. In Linux, for example, `dma_unmap` batches IOTLB invalidations by adding the unmapped IOVAs to a global list. The IOTLB is then invalidated after batching 250 invalidations or every 10 milliseconds, whichever occurs first. The unmapped IOVAs are also deallocated at this time, so that they can be reused later. Other OSes make similar compromises [42].

Unfortunately, this approach still imposes unacceptable overheads on multi-core workloads, because the global list of pending invalidations is itself lock-protected and becomes a bottleneck [42]. To address this, IOTLB invalidations must be batched locally on each core instead of globally [42]. This, however, increases the vulnerability window in which a device can access unmapped IOVAs [42].

3. Assumptions & Attacker Model

Assumptions Our focus is protecting the OS from *unauthorized* DMAs to targets not mapped with the DMA API (see § 2.2). Attacks carried out with authorized DMAs—

[1] The API also has map/unmap operations for non-consecutive scatter/gatter lists, which work analogously.

e.g., a malicious disk tampering with file contents—are out of scope. Peer-to-peer attacks mounted by a device against another device [53] are also out of scope, as peer-to-peer traffic does not go through the IOMMU. [2] We do not consider interrupt-based attacks [51], which are prevented with low overhead using the IOMMU's interrupt remapping feature [53].

We assume that the IOMMU is secure and trustworthy. We also assume that it correctly identifies the device issuing a DMA (i.e., DMAs cannot be spoofed)—this holds on PCIe-only systems and systems that do not have multiple devices behind PCI or PCI-to-PCIe bridges [44, 53]. Finally, we assume that the IOMMU prevents DMAs from compromising the OS during boot—this is true on platforms with a secure boot process, e.g., Intel's TXT feature [1, 28].

Attacker model The attacker controls a set of DMA-capable hardware devices but cannot otherwise access the OS. [3] We thus assume that device drivers are trusted—in particular, to correctly use the DMA API—which is the case in the commodity OSes we focus on. Note, however, that our shadow buffers design can be implemented in systems that isolate drivers as untrusted components [15, 23, 35, 43], in which case this assumption is not needed.

4. Current Intra-OS Protection Weaknesses

The current Intra-OS protection deployed by commodity OSes does not fully protect from DMA attacks, due to two weaknesses: lack of *sub-page protection* and providing only *deferred* protection by default.

No sub-page protection Because IOMMU protection works at page-level granularity (§ 2.1), it cannot protect memory at the byte-level granularity specified by the DMA API. Thus, a device can access any data co-located on a page in which a DMA buffer begins or ends. DMA buffers are typically allocated with standard kernel `malloc` calls, which co-locate multiple allocations on the same page [13]. This makes it possible for data that should not be accessible to the device to reside on a page that gets mapped in the IOMMU.

We argue that addressing this weakness by fixing the relevant `malloc` callers to allocate DMA buffers in quantities of pages will be hard and problematic. First, one would have to reason about every `malloc()` call and determine if a subset of the buffer could end up being used for DMA, because DMA buffer allocations can occur outside of drivers (e.g., in the socket or and storage layers). This requires significant effort, is error-prone, and adds maintenance overhead. In contrast, having the DMA API provide byte-level protection is simpler and automatically benefits the entire kernel. Moreover, allocating DMA-able buffers in page quantities would impose significant memory overhead, proportional to the allocated data. In contrast, the memory overhead of shadow buffers is proportional only to the DMAs *in flight*.

Deferred protection Due to performance reasons, OSes by default implement deferred protection, which performs IOTLB invalidations asynchronously instead of invalidating the IOTLB on each `dma_unmap` (§ 2.2.1). A DMA buffer may thus remain accessible to the device after a `dma_unmap` of the buffer returns. Several attacks can be carried out in this window of vulnerability. For example, after an incoming packet passes firewall inspection, a malicious NIC can modify the packet into a malicious one [15]. Moreover, an unmapped buffer may get reused by the OS, exposing the device to arbitrary sensitive data.

Such attacks appear feasible. We have observed that overwriting an unmapped DMA buffer within 10 μs of its `dma_unmap`[4] can cause a Linux kernel crash—and with deferred protection, buffers can remain mapped for up to 10 milliseconds. This indicates it may be practical to compromise the OS through such an attack vector. While we leave exploiting deferred protection to future work, we contend that OSes require strict protection for true security.

5. Intra-OS Protection via DMA Shadowing

Here we describe our secure intra-OS protection design, which provides strict protection at sub-page (byte-level) granularity and requires no changes to the DMA API. The basic idea is simple: we restrict a device's DMAs to a set of *shadow DMA buffers* that are permanently mapped in the IOMMU, and copy data to (or from) these buffers from (or to) the OS-allocated DMA buffers. Our design thus obviates the need for unmappings and IOTLB invalidations, but adds the additional cost of copying. However, the insight driving our design is that because of the slow IOMMU hardware and the synchronization required to interact with it, copying is typically preferable to an IOTLB invalidation (as shown in § 6).

Realizing the DMA shadowing idea poses several challenges that we discuss in detail below: The design must carefully manage NUMA locality and synchronization issues to minimize overheads (§ 5.3), and it must handle DMA buffers whose size is such that copying would impose prohibitive overheads (§ 5.5).

5.1 Design Goals

Apart from secure intra-OS protection, DMA shadowing sets to achieve the following goals:

• **Transparency:** Implementing DMA shadowing must not require changes to the DMA API, allowing the design to be easily integrated into existing OSes. We discuss this when

[2] Peer-to-peer attacks can be prevented using PCIe Access Control Services (ACS) [41, § 6.13], as explained by Zhou et al. [53].

[3] For example, the attacker cannot exploit a kernel vulnerability from userspace to reconfigure the IOMMU or authorize DMAs to arbitrary targets.

[4] The overwrite was caused by a bug in our copy-based intra-OS protection mechanism (§ 5), whose effect—a DMA buffer being overwritten after being unmapped—is similar to a DMA attack.

`iova_t acquire_shadow(`*buf*, *size*, *rights*`)`	Acquires a shadow buffer and associates it with the OS buffer *buf*. The acquired shadow buffer is of at least *size* bytes, and device access to it is restricted as specified in *rights* (read/write/both). Returns the IOVA of the shadow buffer. The pool guarantees that if a page of the shadow buffer holds another shadow buffer, then both shadow buffers have the same access rights.
`void* find_shadow(`*iova*`)`	Looks up the shadow buffer whose IOVA is *iova* and returns the OS buffer associated with it.
`void release_shadow(`*shbuf*`)`	Releases the shadow buffer *shbuf* back to the pool, disassociating it from its OS buffer.

Table 2: *API of shadow DMA buffer pool (§ 5.3).*

presenting the design in § 5.2. We do, however, extend the API to allow optional optimizations (§ 5.4).

• **Scalability:** The design must minimize synchronization, particularly coarse-grained locks, so that it does not impose bottlenecks on multi-core I/O workloads (see § 2.2.1). We address this in § 5.3.

• **Generality:** While our focus is on the high-throughput I/O workloads that are most sensitive to DMA API performance, our design must support all workloads—providing intra-OS protection only from *some* devices will not offer full protection from DMA attacks. This boils down to supporting huge DMA buffers efficiently, which we discuss in § 5.5.

5.2 DMA Shadowing Implementation of the DMA API

We implement the DMA API as a layer on top of a *shadow buffer pool* that is associated with each device. This pool manages buffers that are permanently mapped in the IOMMU and accessible only to the associated device. The shadow buffer pool interface is shown in Table 2; its implementation is described in § 5.3.

The DMA API operations acquire a shadow buffer from the pool, copy data to (or from) the OS buffer from (or to) the shadow buffer, and finally release the shadow buffer, as follows:

• `dma_map` acquires a shadow buffer of the appropriate size and access rights from the pool. The shadow buffer will be associated with the mapped OS buffer until the shadow buffer is released back to the pool. If the `dma_map` is for data meant to be read by the device, the `dma_map` now copies the OS buffer into the shadow buffer. Finally, it returns the shadow buffer's IOVA.

• `dma_unmap` finds the shadow buffer associated with the OS buffer, based on its IOVA argument. If the `dma_unmap` is for a buffer meant to be written to by the device, `dma_unmap` now copies the contents of the shadow buffer into the OS buffer. (The `dma_unmap` arguments include the size and access rights of the buffer, so this is easy to do.) It then releases the shadow buffer and returns.

The scatter/gather (SG) operations are implemented analogously, with each SG element copied to/from its own shadow buffer.

• For `dma_alloc_coherent` and `dma_free_coherent`, we use the standard DMA API implementation with strict protection. These are infrequent operations (typically invoked at driver initialization and destruction) that are not performance critical. Furthermore, the standard implementation already provides byte-level protection by guaranteeing that `dma_alloc_coherent` memory is allocated in page quantities.

Security DMA shadowing maintains the security semantics of the DMA API, even though the device can always access all the shadow buffers: The implementation reads from an OS buffer at `dma_map` time and writes at `dma_unmap` time, and the pool guarantees that every IOMMU-mapped page holds only shadow buffers with the same access rights. Thus, bytes a device reads can only come from data previously copied from an OS buffer mapped for read or read/write access. Similarly, bytes a device writes are either overwritten by some later copy and never observed by the OS, or are copied out on `dma_unmap`, implying that the target OS buffer was mapped for write access.

DMA shadowing allows a device compromised at some point in time to read data from buffers used at earlier points in time. This does not constitute a security violation, since our attacker model (§ 3) assumes that a device is *always* controlled by the attacker, which implies that the OS *never* places sensitive data in a shadow buffer.

5.3 Shadow Buffer Pool

Each device is associated with a unique shadow buffer pool. The shadow buffer pool is essentially a fast and scalable multi-threaded segregated free list memory allocator [31] that provides the DMA API with shadow DMA buffers (Table 2). It carefully manages NUMA locality and uses lightweight fine-grained synchronization, enabling multiple pool (and hence DMA API) operations to run concurrently. The pool assigns shadow buffers with IOVAs that encode information about the shadow buffer and its free list. It leverages this information to implement `find_shadow` in $O(1)$ time.

Pool design A pool maintains a unique set of free lists from which shadow buffers are acquired. Each list holds free shadow buffers of a particular size *and device access rights*.

That is, for each shadow buffer size class, the pool maintains three free lists—read, write or both. Acquisitions of shadow buffers whose size exceeds the maximum free list size class are described in § 5.5.

Each core maintains its own set of free lists, to allow fast concurrent free list operations (we describe how this is implemented later on). Both shadow buffers and free list metadata are allocated from the core's NUMA domain.

Shadow buffers are *sticky*, in the sense that a shadow buffer acquired from some free list always returns to the same list upon release, even if it is released by a different core. This prevents shadow buffers allocated on one NUMA domain from being acquired later on another NUMA domain, where they would be more expensive to access. In addition, keeping a shadow buffer in the same free list means that its IOMMU mapping never changes. Otherwise, we would have to modify the mapping and invalidate the IOTLB when a shadow buffer moves between free lists with incompatible access rights.

Shadow buffer metadata Each NUMA domain maintains an array of shadow buffer metadata structures for each size-class. (Using an array allows looking up a metadata structure based on its index, which is encoded in the IOVA, as explained later on.) The metadata of *free* shadow buffers—those not currently acquired by the DMA API—doubles as nodes in a singly linked list from which shadow buffers are acquired/released (Figure 2). When a shadow buffer is acquired, its metadata is updated to point to the OS buffer being shadowed. This allows finding the OS buffer when the DMA API needs it (i.e., the `find_shadow` operation). The metadata is not mapped in the IOMMU and is not accessible to the device.

Free list synchronization The free list supports concurrent acquires and releases of shadow buffers with minimal synchronization and cross-core cache coherence traffic. Shadow buffers are acquired from the free list only by the core that owns the list. However, they may be released back to the free list by other cores.

To support these operations efficiently, the free list maintains pointers to the head and tail of the list on distinct cache lines (Figure 2). Shadow buffers are acquired from the head and released to the tail. Acquires are lockless, with the owner core simply removing the head node from the list. Shadow buffer releases are done under a lock that is co-located on the tail pointer's cache line. The released shadow buffer's metadata node is appended to the linked list. If the list was previously empty, the head pointer is updated as well. This is safe to do because when an acquire operation finds the list empty, it allocates a new shadow buffer and returns it, relying on a later release to add the shadow buffer to the free list.

In addition, to avoid false sharing, the free list contains a second read-only pointer to the shadow buffer metadata array, which resides on a separate cache line and from which non-owner cores read.

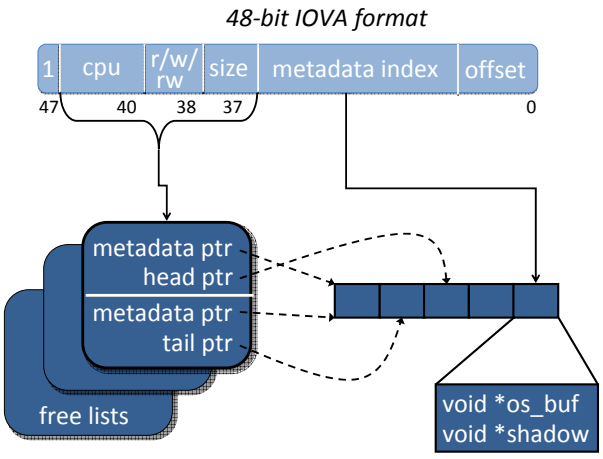


Figure 2: *Shadow buffer IOVA and free list structures. Bits* 37–46 *of a shadow buffer's IOVA encode its free list, and bits* 0–36 *encode its metadata structure. For free shadow buffers, the* `os_buf` *metadata field points to the next node in the free list.*

Shadow buffer allocation Free lists are initially empty. A shadow buffer is allocated when a core tries to acquire a shadow buffer from an empty free list. The core then:

- allocates a new shadow buffer from its NUMA domain,
- stores its address in the next unused metadata node in the domain's metadata array,[5]
- obtains an IOVA for the buffer by encoding the list and metadata node (as described below),
- maps the IOVA to the new shadow buffer in the device's IOMMU page table, and returns.

To provide byte-level protection, the pool guarantees that if several shadow buffers are co-located on the same page, the device has the same access rights to all of them. It achieves this by allocating memory for shadow buffers in page (4 KB) quantities. For small shadow buffers, the pool breaks up allocated pages into multiple shadow buffers. One of these shadow buffers is then returned as usual, and the rest are placed in a private cache to satisfy future allocations. (We do not want to insert them into the free list, to avoid having to synchronize with shadow buffer releases.)

IOVA encodings A shadow buffer's IOVA uniquely identifies its metadata structure. The IOVA encodes the shadow buffer's free list—identified by the owner core id, size class, and access rights—and the index of the metadata structure in the metadata array of the owner core's NUMA domain.

x86 IOVAs are presently 48 bits wide [30]. We reserve the MSB to signify that the IOVA encodes shadow buffer metadata. The remainder bits can be divided in numerous ways. Figure 2 shows the encoding used in our prototype

[5] This next-unused index is lock-protected. Shadow buffer allocation is an infrequent operation, so this lock does not pose a contention problem.

implementation, which supports shadow buffers of two size-classes, 4 KB and 64 KB: 7 bits for core id, 2 bits for access rights (read/write/both), 1 bit for size class, and 37 bits to encode the metadata structure index. Notice that for a size class C, the least significant $\lceil \log_2 C \rceil$ bits are not used in the index encoding, as they are used for addressing within the shadow buffer. Thus, when decoding an IOVA we first identify the appropriate size class and then extract the metadata index. We remark that one can have more size classes by using less bits for the index and/or core id fields.

The half of the IOVA space with MSB clear serves as a fallback, in case a NUMA domain exhausts a shadow buffer metadata array. In such a case, we allocate the metadata using kernel `malloc` and the IOVA with an external scalable IOVA allocator [42]. The IOVA-to-metadata mapping in these fallback cases is maintained in an external hash table.

Memory consumption Shadow DMA buffer use corresponds to *in flight* DMAs. Because shadow buffers are not used for huge DMA operations (§ 5.5), typical shadow buffer memory consumption should be modest—for example, we observe < 256 MB consumed by shadow buffers in our experiments (§ 6). Moreover, the OS can impose a limit on the memory allocated to shadow buffers and/or free unused shadow buffers under memory pressure. (When a shadow buffer is freed, it must be unmapped from the IOMMU, including an IOTLB invalidation. This should not adversely affect performance as long as memory pressure-related freeing does not happen frequently.)

5.4 Copying Optimizations

The main overhead of DMA shadowing is in the copying to/from the OS buffers—the shadow pool operations are fast and scalable. We have explored several options for optimizing copying:

Copying hints: DMA buffers often end up not full—for example, a networking driver maps MTU-sized buffers for incoming packets, but the arriving packets can be much smaller. To avoid copying useless data, we let drivers register an optional *copying hint*, which is a function that is given a DMA buffer as input and returns the amount of data that should be copied. For example, our prototype implementation uses a copying hint that returns the length of the IP packet in the buffer. It is the driver writer's responsibility to ensure that the copying hint is fast and secure, since its input is untrusted.

Smart `memcpy`***:*** We have attempted to use optimized `memcpy` implementations that use SIMD instructions and/or streaming (non-temporal) stores [29]. However, we have found that on our machines, these do not provide an overall benefit over the standard `memcpy` implementation based on the x86 `REP MOVSB/STOSB` instruction. We suspect the reason is that this instruction is optimized in our processors, which have an enhanced `REP MOVSB/STOSB` (ERMS) feature.

5.5 Handling Huge DMA Buffers

Copying is not *always* preferable to an IOTLB invalidation—copying a large enough buffer can greatly exceed invalidation cost, even after factoring in the lock contention involved. We observe, however, that huge DMA buffers have infrequent DMA map/unmaps. This is because the rate of IO operations decreases, as more time is spent on transferring the data to/from the device and processing it afterwards. For example, while a 40 Gb/s NIC can DMA incoming 1500-byte packets at a rate of 1.7 M packets/sec, Intel's solid-state drive (SSD)—whose DMA buffers are at least 4 KB—provides up to 850 K IOPS for reads and up to 150 K IOPS for writes [27]. When DMA map/unmap rate is low, the overhead of IOMMU unmapping becomes insignificant, which makes it possible to reuse the standard zero-copy DMA mapping technique.

To maintain byte-level protection, we propose using a hybrid approach that copies only the sub-page head/tail of the OS DMA buffer, and maps the remainder in the IOMMU. One can use an external scalable IOVA allocator [42] to obtain a range of IOVAs for mapping the head/tail shadows with the remainder between them. On `dma_unmap`, this mapping is destroyed, including invalidating the IOTLB.

6. Evaluation

We implement DMA shadowing in Linux 3.19. Our implementation consists of ≈ 1000 LOC. We focus our evaluation on 40 Gb/s networking, as it represents demanding I/O workloads that trigger millions of DMAs per second. Our evaluation seeks to answer the following questions:

- How does the overhead imposed by copying DMA buffers compare to the overhead of existing IOMMU-based protection?
- What is the performance effect of DMA shadowing on I/O intensive workloads?

To this end, our DMA shadowing implementation focuses on the copying-related parts of the design. Moreover, we enable DMA shadowing only for the device being tested. We thus do not implement fallbacks to standard IOVA allocation and handling of huge DMA buffers, both of which do not occur in our benchmarks.

Evaluated systems We compare DMA shadowing (denoted *copy*) to Linux 3.19 with the IOMMU disabled (*no iommu*), i.e., with no DMA attack protection. We also compare to a Linux 3.19 variant that uses identity mappings for IOVAs [42], with both strict (*identity+*) and deferred (*identity-*) protection. We use this variant instead of the baseline Linux because it significantly outperforms Linux under both modes, as it resolves a Linux bottleneck in IOVA assignment during `dma_map` [42].

Experimental setup Our setup consists of two Dell PowerEdge R430 machines. Each machine has dual 2.40 GHz Intel Xeon E5-2630 v3 (Haswell) CPUs, each with 8 cores,

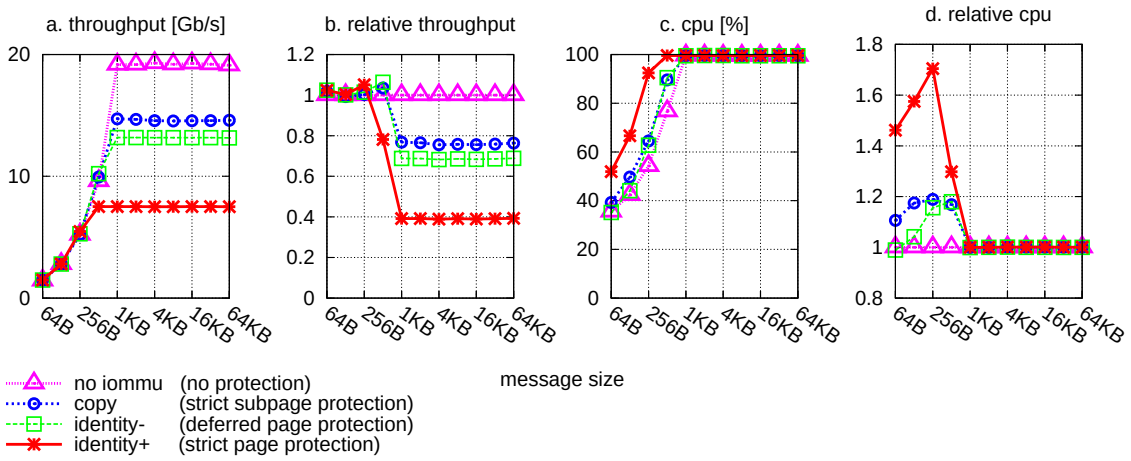


Figure 3: *Single-core TCP receive (RX) throughput and CPU utilization (netperf TCP STREAM)*

for a total of 16 cores (hyperthreading is disabled). Each machine has two 16 GB 1867 MHz DDR4 DIMMs (one per socket), for a total of 32 GB of memory. One machine runs the evaluated kernel and the other serves as the traffic generator. The traffic generator runs with its IOMMU disabled. The machines are connected back-to-back with 40 Gb/s NICs: an Intel Fortville LX710 on the evaluated machine and a Mellanox ConnectX3 on the traffic generator. The machines are configured for maximum performance, with dynamic control of the clock rate (Turbo Boost) disabled.

Methodology We configure the NIC drivers to use one receive ring per core, to avoid measurement noise from uneven load balancing between the cores. We configure even interrupt distribution between the cores for the same reason. We run measurements on idle systems. Each benchmark runs for 60 seconds, to amortize any sporadic noise. We report averages of 10 runs.

Benchmarks We evaluate TCP/IP throughput and latency using netperf [32], a standard network benchmarking tool. We evaluate both receive (RX) and transmit (TX) throughput, by performing separate experiments in which the evaluation machine is a netperf receiver/transmitter. (We further detail each experiment below, as we present it.) We additionally study an application workload using memcached [22], a popular high-performance key-value store used by web applications for object caching. We run a memcached instance per core, to avoid lock contention in multi-threaded memcached, and measure aggregated throughput under a memslap [4] workload. We use the default memslap configuration of 64-byte keys, 1 KB values, and 90%/10% GET/SET operations.

Single-core TCP throughput We measure the throughput obtained in netperf's TCP STREAM test, varying its *message size*—the amount of data that the sending machine repeatedly writes to the TCP socket. We first evaluate single-core throughput: Figure 3 shows throughput and CPU utilization in the RX test, in which the evaluation system is the receiver. For small message sizes (up to 512 B), the evaluated systems are not the bottleneck—as evidenced by the CPU utilization—and they all obtain the same throughput.[6] Thus, overheads translate into different CPU utilization: *copy* has CPU overhead of 1.1×–1.2× compared to *no iommu*, and *identity+* has CPU overhead of 1.3×–1.7×. With larger messages, the bottleneck shifts to the receiver and overheads translate into throughput. Interestingly, *copy* is the best performer after *no iommu*—outperforming *identity-* by 10% despite providing stronger protection, and obtaining 0.76× the throughput of *no iommu*. The overhead of strict protection in *identity+* is much larger, and *copy* obtains 2× its throughput.

To understand the source of these overheads, Figure 5a breaks down the average packet processing time. IOMMU-related map/unmap overhead dominates both zero-copy schemes. IOMMU page table management costs both *identity-* and *identity+* 0.17 µs, and *identity+* additionally spends 0.61 µs on IOTLB invalidation. In contrast, *copy* spends 0.02 µs on shadow buffer management and 0.11 µs on copying from the shadow buffer (`memcpy`)—that is, *copying a 1500 B ethernet packet is* 5.5× *faster than invalidating the IOTLB*.

Figure 4 shows transmit (TX) throughput[7] and CPU utilization. For messages sizes < 512 B, *copy* performs comparably to *identity+* and *identity-*, while providing better security. Unlike the RX case, however, with larger messages *copy* obtains the worst throughput—10% to 20% worse than the other designs. In addition, *copy* is the only design that

[6] The limiting factor is the sender's system call execution rate.

[7] Notice that peak *no iommu* TX throughput differs from peak RX throughput. The reason is that the RX test measures true *single-core* RX throughput—running netperf and handling interrupts on the same core—whereas in the TX test the receiver does not limit networking in this way. We do this to guarantee that the receiver is not the bottleneck in the TX test.

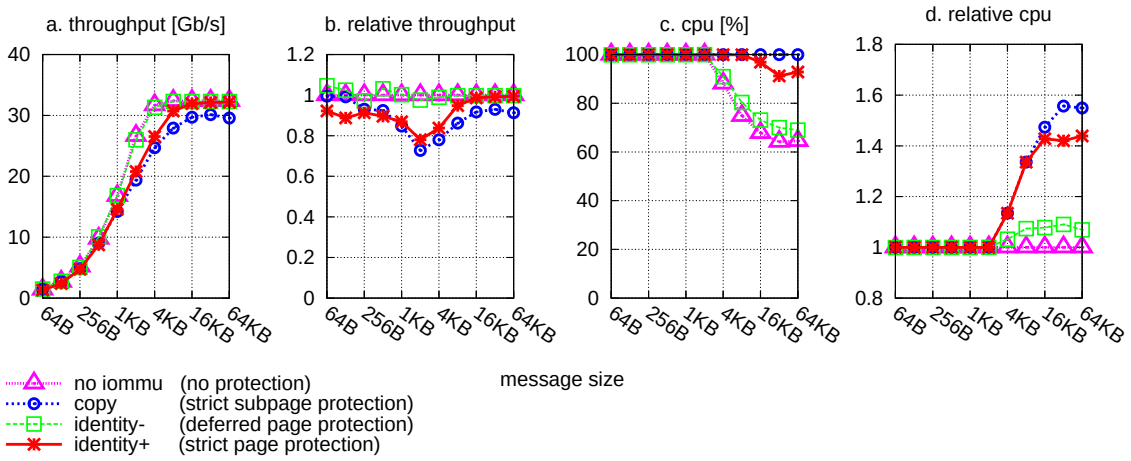
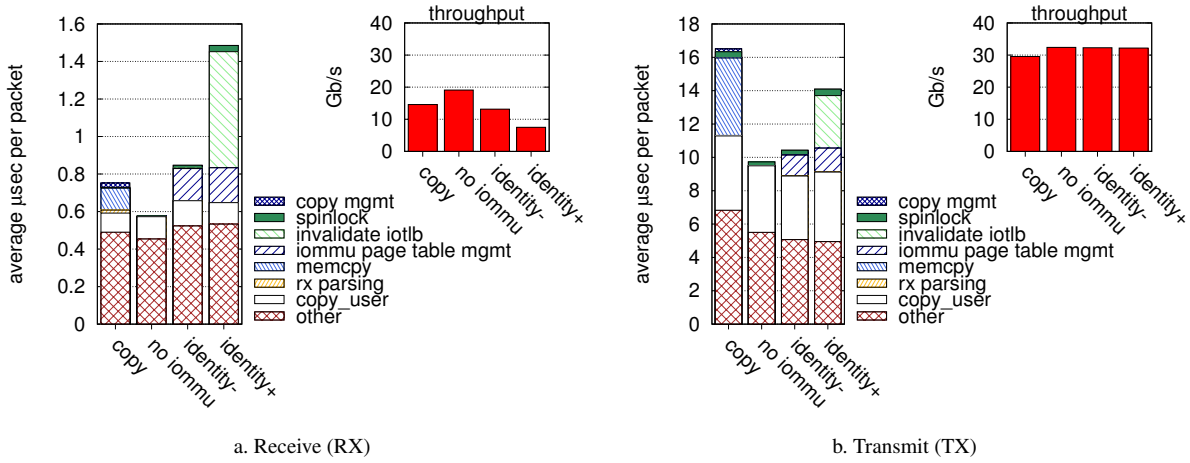


Figure 4: *Single-core TCP transmit (TX) throughput and CPU utilization (netperf TCP STREAM)*



a. Receive (RX)

b. Transmit (TX)

Figure 5: *Average packet processing time breakdown in single-core TCP throughput tests (64 KB message size).*

keeps the CPU 100% busy with 64 KB messages, a 1.4× CPU overhead over *no iommu*. The reason for *copy*'s behavior is the TCP segmentation offload (TSO) feature of the evaluated machine's NIC. With TSO, the driver can pass the NIC a packet of up 64 KB, and the NIC breaks it internally and transmits MTU-sized packets on the wire. Consequently, *copy* needs to copy 64 KB DMA buffers, as opposed to the RX case, in which the buffers are bounded by the 1500 B MTU. Figure 5b depicts the effect of this larger copy operation. First, the `memcpy` time for *copy* increases to 4.65 μs. (This is 40× that of the RX case.) Second, *copy* spends $\approx 2\,\mu$s on "other" tasks. We attribute this increase to cache pollution: A 64 KB buffer copy evicts all data from the core's 32 KB L1 cache, replacing it with data that the core will not later use (as opposed to the RX case). Interestingly, IOMMU-related overhead of *identity+* is 4.58 μs, roughly the same as the `memcpy` time in *copy*. It is the cache pollution that tips the scale slightly in favor of *identity+*.

Multi-core TCP throughput As a single core cannot sustain the 40 Gb/s wire throughput even without an IOMMU, we explore multi-core TCP throughput. We run 16 netperf client/server instances (one per core) on each machine. We report aggregated TCP throughput and CPU utilization, for RX in Figure 6 and for TX in Figure 7. (Here CPU utilization is over all cores, i.e., eight 100% busy cores translate to a 50% utilization.)

We observe a striking difference between *identity+* and the other designs. For RX, *identity+* obtains 5× worse throughput than the other designs (which obtain comparable throughput among each other) across all message sizes. For TX, *identity+* is 5× worse for small message sizes, but closes the gap as message size increases. Moreover, *identity+* is the only design that always has 100% CPU utilization, both for RX and TX. The reason for this is a dramatic increase in overhead compared to the single-core case. The average packet processing time breakdown in Figure 8 depicts this: First, IOTLB invalidation time in *identity+* in-

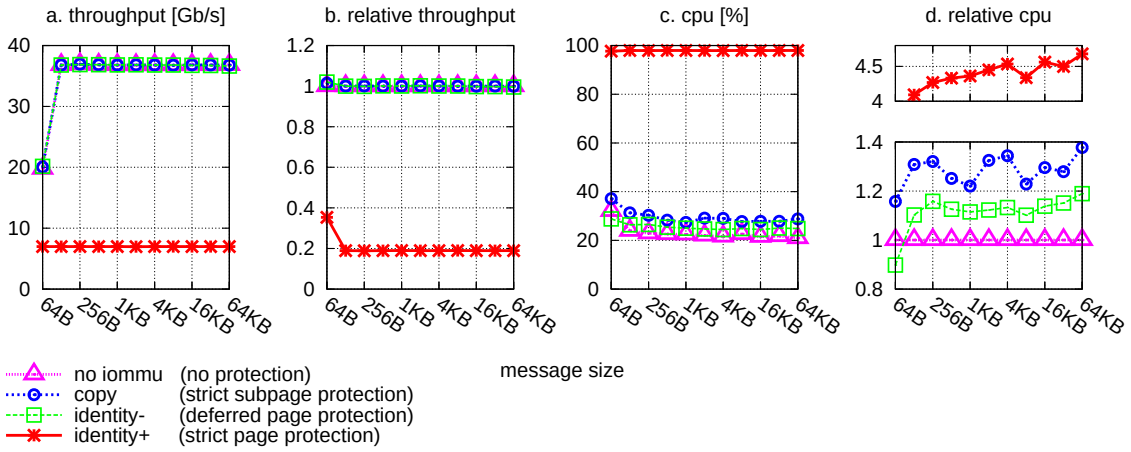


Figure 6: *16-core TCP receive (RX) throughput and CPU utilization (netperf TCP STREAM).*

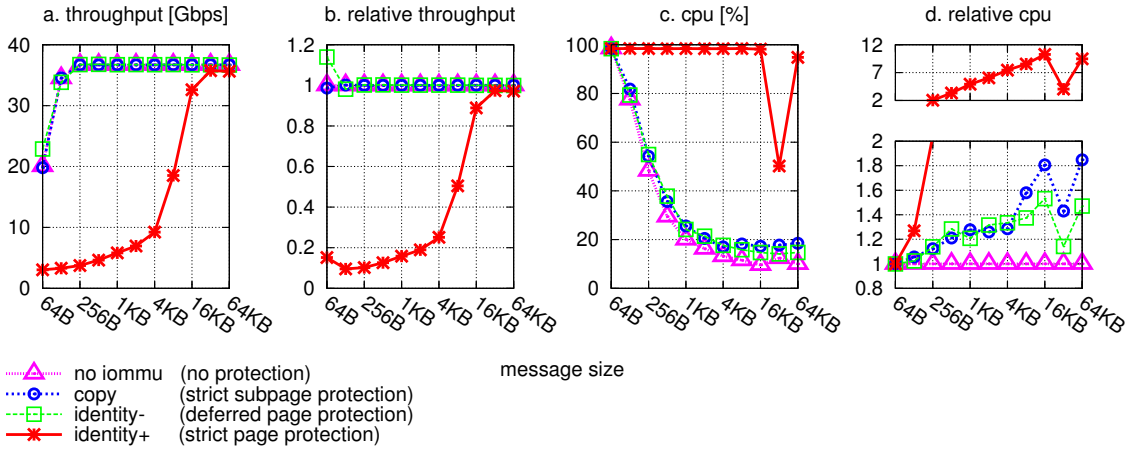


Figure 7: *16-core TCP transmit (TX) throughput and CPU utilization (netperf TCP STREAM).*

creases to 2.7 μs (however, it remains unchanged for TX). More significantly, *identity+* now suffers from contention on the IOTLB lock (§ 2.2.1). In RX, which requires more MTU-sized packets/second to sustain line rate, this contention is more severe—$\approx$ 70 μs per packet—than in TX, where TSO causes packet rate to decrease as message size increases. This is why on TX, *identity+* eventually manages to drive 40 Gb/s, whereas for RX its throughput remains constant. In short, Figure 8b shows that IOTLB invalidation lock overhead *is even more expensive* than the 64 KB buffer `memcpy` done by *copy* in TX, including the resulting cache pollution.

The *copy* design achieves comparable throughput to *no iommu*, with up to 60% CPU overhead, for both RX and TX. While *identity-* also achieves this throughput, with lower CPU overheads of up to 20%, it trades off strict OS protection in order to do so.

TCP latency We measure TCP latency with a single-core netperf request/response benchmark. This benchmark measures the latency of sending a TCP message of a certain size (which we vary) and receiving a response of the same size. Figure 9 shows the resulting latency and CPU utilization. In this benchmark, per-byte costs are not the dominating factor: observe that although the message size increases by 1024$\times$ from 64 B to 64 KB, the latency only increases by $\approx 4\times$. Indeed, the protection-related overheads in the *copy* and the *identity* designs do not make a noticeable impact on the overall latency, and all designs obtain comparable latency to *no iommu*. The overheads are more observable through the CPU utilization, which is broken down (for 64 KB messages) in Figure 10. We find that *identity+* spends almost half its time on IOMMU-related tasks, whereas the *copy* overheads of shadow buffer management and copying constitute 20% of its time, and less than 10% of the overall CPU time.

memcached Our memcached benchmark consists of multiple memslap instances generating load on 16 memcached instances running on the evaluation machine. The results, depicted in Figure **??**, exhibit a similar trend to the multi-core TCP throughput benchmark. Except for *identity+*, all designs obtain comparable memcached transactional throughput, which is 6.6$\times$ that of the throughput *identity+* ob-

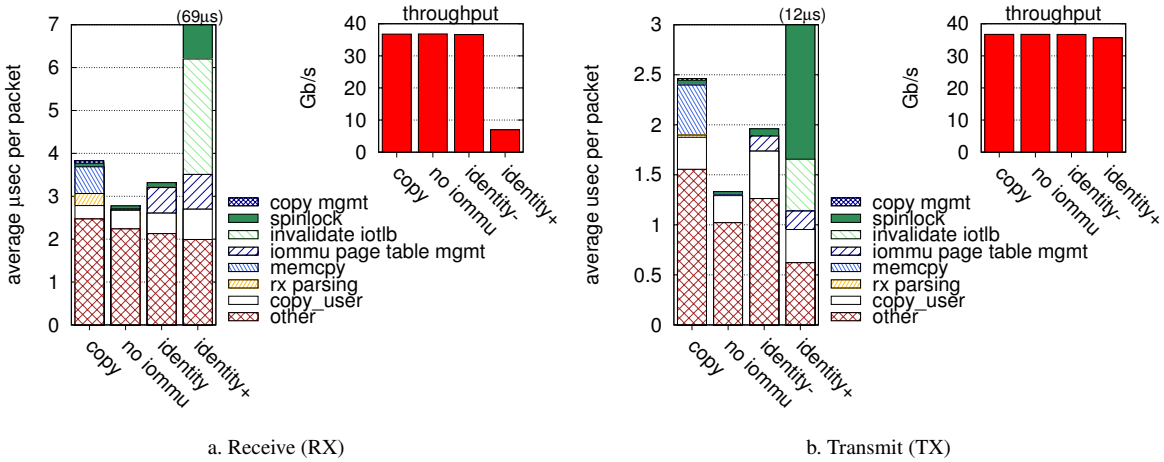


Figure 8: *Average packet processing time breakdown in 16-core TCP throughput tests (64 KB message size).*

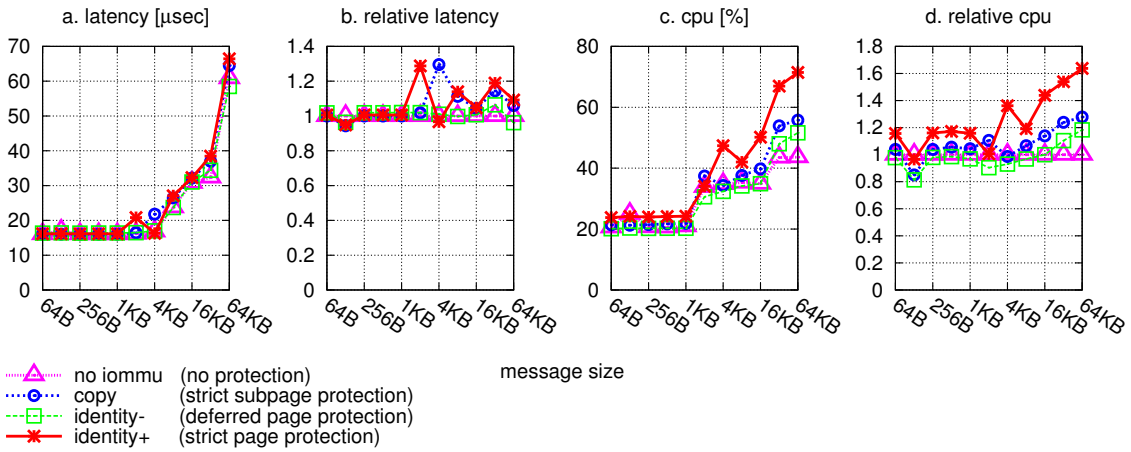


Figure 9: *TCP latency (single-core netperf TCP request/response).*

tains. We thus see that for a realistic demanding workload, *copy* provides full DMA attack protection at essentially the same throughput and CPU utilization ($< 2\%$ overhead) as *no iommu*.

Memory consumption Our prototype implementation of *copy* supports shadow buffers of two size classes, 4 KB and 64 KB. Although a size class of C bytes can have at most $2^{37-\lceil \log_2 C \rceil}$ shadow buffers (§ 5.3), we use a more practical bound of 16 K buffers. Thus, in the worst case, *copy* could consume about 2.1 GB (4 KB $\times$ 16 K = 64 MB plus 64 KB $\times$ 16 K = 1 GB, for each of the two NUMA domains).

In practice, however, we expect memory consumption to be dramatically less. This is because shadow DMA buffer allocations correspond to *in flight* DMAs. To test this, we measured shadow DMA buffer memory consumption during our benchmarks. We observe 64 MB of shadow buffers being allocated to shadow TX buffers, and 96 MB to shadow RX buffers—i.e., $\approx 13\times$ less than the worst case bound.

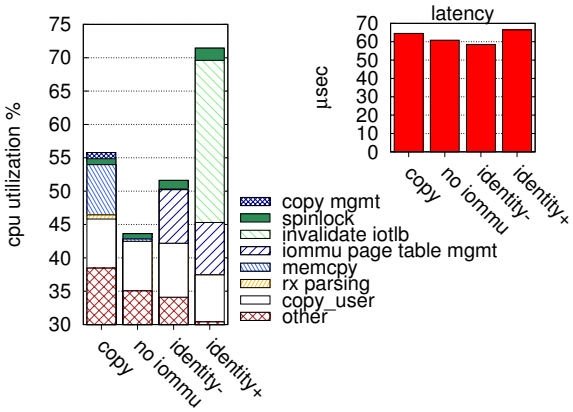


Figure 10: *Single-core TCP request/response CPU utilization breakdown (64 KB message size).*

Summary Our results show that *copy* provides full protection from DMA attacks with comparable or better per-

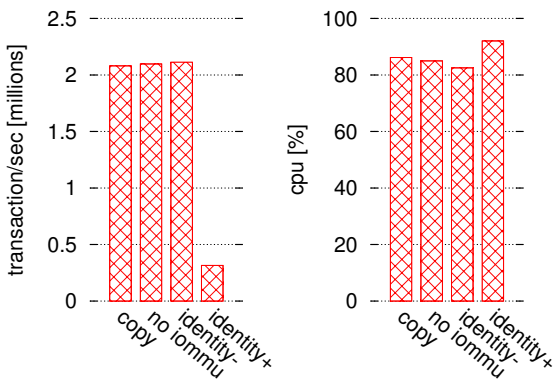


Figure 11: *memcached: aggregated throughput (16 instances).*

formance than the existing, less secure designs. Compared to *identity+*, which does not provide byte-level protection, *copy* provides comparable (< 10% less) or far better throughput (2×–5×), comparable latency, and better or comparable CPU utilization. Compared to *identity-*, which lacks byte-level protection and may allow devices to access unmapped memory, *copy* obtains comparable throughput and latency with 1×–1.4× CPU overhead. Finally, *copy* obtains 0.76×–1× the throughput of *no iommu*, which is defenseless against DMA attacks, with a 20% increase in CPU utilization (which translates to an overhead of 1.8×).

7. Related work

IOMMU-based OS protection Willmann et al. [50] describe five IOMMU-based protection strategies, each of which lacks one or more of the properties provided by our design—strict intra-OS protection (§ 2.2.1) at byte granularity with efficiency and scalability that support high-throughput I/O such as 10–40 Gb/s networking. EiovaR [38] provides strict protection but at page granularity, and it shares the scalability bottlenecks of Linux. Peleg et al. [42] describe scalable protection schemes, but these require deferred protection for high-throughput and are at page granularity. In SUD [15], usermode drivers communicate with the OS through shared IOMMU-mapped buffers, which act similarly to our shadow buffers. While SUD also protects from malicious drivers, it does not appear compatible with high-throughput I/O requirements. Even for 1 Gb/s networking, SUD incurs up to 2× CPU overhead due to context switches, and must use batching—hurting latency—to obtain acceptable throughput [15]. Arrakis [43] provides direct application-level access to devices for efficient I/O. To protect themselves from DMA attacks, Arrakis applications must thus use a DMA API-like scheme, making our shadow buffers relevant there.

Copying-based protection Linux supports an SWIOTLB mode [2] in which the DMA API is implemented by copying DMA buffers to/from dedicated *bounce buffers*. This mode makes no use of the hardware IOMMU and thus provides no protection from DMA attacks. Instead, its goal is to allow systems without an IOMMU to work with devices, such as 32-bit devices, that cannot address the entire physical address space. Horovitz et al. describe a protection scheme similar to shadow DMA buffers [24] in the context of a software cryptoprocessor. However, they do not discuss implementation details and do not provide a performance evaluation.

Hardware solutions Basu et al. [10] propose a hardware IOMMU design in which mappings self-destruct after a threshold of time or DMAs, thereby obviating the need to destroy the mapping in software. However, this hardware is not currently available.

8. Conclusion

Due to lack of byte-granularity protection and trading off security to obtain acceptable performance when using IOMMUs, OSes remain vulnerable to DMA attacks. We propose a new way of using IOMMUs, which fully defends against DMA attacks. Our new usage model restricts device access to a set of shadow DMA buffers that are never unmapped, and copies DMAed data to/from these buffers. It thus provides sub-page byte-granularity protection while closing the current vulnerability window in which devices can access in-use memory. Our key insight is that the gains of zero-copy for IOMMU protection are negated by the cost of interacting with the slow IOMMU hardware and the synchronization it entails, making copying preferable to zero-copying in many cases.

We show that, despite being more secure than the safest preexisting usage model, our DMA shadowing approach provides up to 5× higher throughput. Moreover, while DMA shadowing cannot indefinitely scale with I/O throughput due to the overhead of copying, we show that it *does scale* to 40 Gb/s rates and incurs only 0%–25% throughput degradation as compared to when the IOMMU is disabled.

With malicious devices being used in the wild [3, 52], defending against DMA attacks becomes an increasingly pressing problem. DMA shadowing provides full protection from DMA attacks at reasonable overheads, and we hope to see it deployed in practice.

Acknowledgments

This research was funded in part by the Israeli Ministry of Science and Technology grant #3-9779, by the Israel Science Foundation (grants 1227/10 and 1749/14), by Yad HaNadiv foundation, and by Mellanox, which additionally graciously donated NICs.

References

[1] Intel TXT Overview. `https://www.kernel.org/doc/Documentation/intel_txt.txt`. Linux kernel documentation.

[2] Dma issues, part 2. https://lwn.net/Articles/91870/. (Accessed: January 2016).

[3] Inside TAO: Documents Reveal Top NSA Hacking Unit. *Der Spiegel*, Dec 2013. http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html. (Accessed: January 2016).

[4] B. Aker. Memslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html. libmemcached 1.1.0 documentation.

[5] AMD Inc. AMD IOMMU architectural specification, rev 2.00. http://developer.amd.com/wordpress/media/2012/10/488821.pdf, Mar 2011.

[6] Apple Inc. Thunderbolt device driver programming guide: Debugging VT-d I/O MMU virtualization. https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/DebuggingThunderboltDrivers/DebuggingThunderboltDrivers.html, 2013. (Accessed: January 2016).

[7] ARM Holdings. ARM system memory management unit architecture specification — SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062c/IHI0062C_system_mmu_architecture_specification.pdf, 2013.

[8] D. Aumaitre and C. Devine. Subverting Windows 7 x64 Kernel with DMA attacks. In *Hack In The Box Security Conference (HITB)*, 2010. http://esec-lab.sogeti.com/static/publications/10-hitbamsterdam-dmaattacks.pdf. (Accessed: January 2016).

[9] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM EuroSys*, pages 73–85, 2006.

[10] A. Basu, M. D. Hill, and M. M. Swift. I/O memory management unit providing self invalidated mapping. https://www.google.com/patents/US20150067296, 2015. US Patent App. 14/012,261.

[11] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. In *CanSecWest Applied Security Conference*, 2005.

[12] A. Boileau. Hit by a Bus: Physical Access Attacks with Firewire. In *Ruxcon*, 2006. http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf. (Accessed: January 2016).

[13] J. Bonwick. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Annual Technical Conference*, pages 87–98, 1994.

[14] J. E. Bottomley. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt. Linux kernel documentation.

[15] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX Annual Technical Conference (ATC)*, pages 117–130, 2010.

[16] J. Brossard. Hardware bakdooring is pratical. In *Black Hat*, 2012. http://www.toucan-system.com/research/blackhat2012_brossard_hardware_backdooring.pdf. (Accessed: January 2016).

[17] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, Feb 2014.

[18] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[19] M. Dornseif. 0wned by an iPod. In *PacSec*, 2004. https://pacsec.jp/psj04/psj04-dornseif-e.ppt. (Accessed: January 2016).

[20] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain. Can you still trust your network card? Technical report, French Network and Information Security Agency (FNISA), Mar 2010. http://www.ssi.gouv.fr/uploads/IMG/pdf/csw-trustnetworkcard.pdf. (Accessed: January 2016).

[21] L. Duflot, Y.-A. Perez, and B. Morin. What if You Can't Trust Your Network Card? In *Conference on Recent Advances in Intrusion Detection*, pages 378–397, 2011.

[22] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), Aug 2004.

[23] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)*, pages 41–50, 2007.

[24] O. Horovitz, S. Rihan, S. A. Weis, and C. A. Waldspurger. Secure support for I/O in software cryptoprocessor. https://patents.google.com/patent/US20150269091A1, 2015. US Patent App. 14/663,217.

[25] IBM Corporation. PowerLinux servers — 64-bit DMA concepts. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm. (Accessed: January 2016).

[26] IBM Corporation. AIX kernel extensions and device support programming concepts. http://public.dhe.ibm.com/systems/power/docs/aix/71/kernextc_pdf.pdf, 2013. (Accssed: July 2015).

[27] Intel Corporation. Intel SSD Data Center Family Product Brief. http://www.intel.com/content/www/us/en/solid-state-drives/data-center-family.html.

[28] Intel Corporation. Intel Trusted Execution Technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2012.

[29] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide. http://download.intel.com/products/processor/manual/325384.pdf, 2013.

[30] Intel Corporation. Intel Virtualization Technology for Directed I/O, Architecture Specification - Architecture Speci-

fication - Rev. 2.3. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Oct 2014.

[31] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *ACM International Symposium on Memory Management (ISMM)*, pages 26–36, 1998.

[32] R. A. Jones. A network performance benchmark (revision 2.0). Technical report, Hewlett Packard, 1995. http://www.netperf.org/netperf/training/Netperf.html.

[33] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2009.

[34] B. H. Leitao. Tuning 10Gb network cards on Linux. In *Ottawa Linux Symposium (OLS)*, pages 169–189, 2009.

[35] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2004.

[36] Linux. Documentation/intel-iommu.txt, Linux 3.18 documentation file. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt. (Accessed: January 2016).

[37] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 355–368, 2015.

[38] M. Malka, N. Amit, and D. Tsafrir. Efficient Intra-Operating System Protection Against Harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 29–44, 2015.

[39] V. Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007. (Accessed: January 2016).

[40] D. S. Miller, R. Henderson, and J. Jelinek. Dynamic DMA mapping guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. Linux kernel documentation.

[41] PCI-SIG. PCI Express 2.0 Base Specification Revision 0.9. https://www.pcisig.com/specifications/iov/ats, Sep 2006.

[42] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir. Utilizing the IOMMU Scalably. In *USENIX Annual Technical Conference (ATC)*, pages 549–562, 2015.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2014.

[44] F. L. Sang, Éric Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability . In *International Conference on Malicious and Unwanted Software (MALWARE)*, pages 7–14, 2010.

[45] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 21–41, 2012.

[46] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, Feb 2005.

[47] A. Triulzi. I 0wn the NIC, now I want a shell! In *PacSec*, 2008. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf. (Accessed: January 2016).

[48] C. Waldspurger and M. Rosenblum. I/O virtualization. *Communications of the ACM (CACM)*, 55(1):66–73, Jan 2012.

[49] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 241–254, 2008.

[50] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008.

[51] R. Wojtczuk and J. Rutkowska. Following the White Rabbit: Software attacks against Intel VT-d technology. http://invisiblethingslab.com/resources/2011/SoftwareAttacksonIntelVT-d.pdf, Apr 2011.

[52] K. Zetter. How the NSA's Firmware Hacking Works and Why It's So Unsettling. *Wired*, Feb 2015. http://www.wired.com/2015/02/nsa-firmware-hacking/. (Accessed: July 2015).

[53] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE Symposium on Security and Privacy (S&P)*, pages 616–630, 2012.