DAMN: Overhead-Free IOMMU Protection for Networking

Alex Markuze[†,◇] Igor Smolyar[†] Adam Morrison[‡] Dan Tsafrir[†,◇]

[†]Technion – Israel Institute of Technology [‡]Tel Aviv University [◇]VMware Research Group

Abstract

DMA operations can access memory buffers only if they are "mapped" in the IOMMU, so operating systems protect themselves against malicious/errant network DMAs by mapping and unmapping each packet immediately before/after it is DMAed. This approach was recently found to be riskier and less performant than keeping packets non-DMAable and instead copying their content to/from permanently-mapped buffers. Still, the extra copy hampers performance of multi-gigabit networking.

We observe that achieving protection at the DMA (un)map boundary is needlessly constraining, as devices must be prevented from changing the data only after the kernel reads it. So there is no real need to switch ownership of buffers between kernel and device at the DMA (un)mapping layer, as opposed to the approach taken by all existing IOMMU protection schemes. We thus eliminate the extra copy by (1) implementing a new allocator called DMA-Aware Malloc for Networking (DAMN), which (de)allocates packet buffers from a memory pool permanently mapped in the IOMMU; (2) modifying the network stack to use this allocator; and (3) copying packet data only when the kernel needs it, which usually morphs the aforementioned extra copy into the kernel's standard copy operation performed at the user-kernel boundary. DAMN thus provides full IOMMU protection with performance comparable to that of an unprotected system.

CCS Concepts • **Security and privacy → Operating systems security**; • **Software and its engineering → Memory management**;

Keywords IOMMU, DMA attacks

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173175

ACM Reference Format:
Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-Free IOMMU Protection for Networking. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173175

1 Introduction

Bandwidth demands of servers in modern data centers scale with the number of cores per server, requiring support for ever-growing networking speeds both at the endpoints and at the data center fabric. Google's Jupiter fabric [36], for example, assumed 40 Gb/s endpoints, and 100 Gb/s network cards (NICs) have since been made available. Data center servers must also offer strong security guarantees to their clients, one aspect of which includes protection from *compromised or malicious devices*, a threat that is explicitly considered in Google's security design [18, Secure Data Storage].

Presently, however, an operating system (OS) cannot defend against malicious NICs without paying a significant performance penalty. This limitation results from the use of I/O memory management units (IOMMUs) [2, 4, 19, 23] to restrict device direct memory access (DMA), as unrestricted DMAs allow malicious devices to steal secret data or compromise the OS [5, 7, 11, 14, 16, 37–39]. The IOMMU views DMA addresses as *I/O virtual addresses* (IOVAs) [25], and it blocks DMAs whose target IOVA does not have a valid mapping to a physical address.

OSes deploy their IOMMU-based DMA attack protection model in the existing DMA APIs [3, 9, 20, 28]. The DMA API seems a perfect fit for such a protection model, as it requires drivers to `dma_map` and `dma_unmap` memory buffers before/after they are DMAed (e.g., to resolve coherency issues and support bounce buffering [10]), and so OSes naturally adopted a protection model in which a device may DMA to/from a buffer if and only if the buffer is DMA mapped.

The straightforward implementation of this protection model—where the OS creates IOMMU mappings in `dma_map` and destroys them in `dma_unmap`—proved crippling to multi-gigabit NICs, because the overhead of destroying IOMMU mappings on each DMA becomes prohibitive for millions of DMAs per second [27, 34]. Consequently, several protection schemes with improved performance have been proposed, by practitioners [24, 41] and researchers [27, 29, 34] alike, with some [27, 34] adopted by Linux.

Still, all existing work compromises on either performance or security, as it adheres to the DMA API-based protection model. On one hand, it seems impossible to enforce a protection boundary that makes buffers inaccessible to the device on `dma_unmap` without imposing overhead on each DMA. On the other hand, relaxing the protection boundary [34] opens a vulnerability window in which the device can access unmapped buffers. The device can then mount "time of check to time of use" (TOCTTOU) attacks such as modifying a packet after it passes firewall checks, or access sensitive data if the unmapped buffer is reused by the OS.

Shadow buffers [29] attempt to alleviate the map-unmap overhead by copying packets to non-DMAable buffers instead of (un)mapping them in the IOMMU, but this approach does not scale with growing networking speeds, cannibalizing the system's CPU and memory bandwidth (§ 4).

We show that the above security/performance dichotomy can be made false by changing the DMA attack protection model. We propose a conceptually different model, whose DMA protection boundary lies at the user/kernel boundary, where packet data is *already copied* and thus made inaccessible to devices without imposing *extra* overhead relative to unprotected networking. Our model uses the IOMMU to prevent arbitrary DMAs—particularly to data copied to/from user-level applications—and not to restrict DMAs only to buffers mapped by the DMA API. In fact, we explicitly allow devices to access packet buffers as they are processed by the OS; we protect OS processing by dynamically copying only the data accessed by the OS—typically, just the headers—when it is first accessed.

We implement our new DMA attack protection model with DAMN, a DMA-Aware Malloc for Networking. DAMN shifts IOMMU management from the DMA API to a higher OS level—the memory allocator. It satisfies packet buffer allocations from a set of buffers permanently mapped in the IOMMU, obviating the need to set up and destroy IOMMU mappings on each DMA. Shadow buffers [29] also restrict the NIC to such permanently-mapped buffers, but since they implement the DMA API-based protection model, the OS cannot directly work with the permanently-mapped buffers. Instead, data must be copied to/from these buffers on each DMA. DAMN's new protection model eliminates this extra copy, and thus DAMN is essentially overhead-free, while still providing DMA attack protection, due to the following.

DAMN-allocated buffers cannot be reused to contain sensitive data, because DAMN is distinct from the kernel `malloc`. Every page that DAMN maps is exclusively used to satisfy packet buffer allocations, so such pages always contain data that the OS allowed the device to access at some point, and nothing else. Separation from the kernel allocator also guarantees that DMA buffers never get co-located on the same page with unrelated kernel data, and so DAMN provides *byte-granularity* IOMMU protection instead of the page-granularity protection of most prior schemes.

DAMN efficiently defends against TOCTTOU attacks by dynamically copying OS-accessed bytes, removing them from the device's reach. To intercept *all* OS accesses to packet buffers, DAMN interposes on the special accessor methods that OS packet accesses must already go through. We observe that only packet headers are typically security sensitive and thus accessed by the OS, whereas packet *data* remains opaque to the OS until it gets copied at the kernel/user boundary and removed from the device's reach. (Any malicious DMA performed before this copy is indistinguishable from a valid DMA performed while the packet was mapped.) Therefore, in the common case, only header data will require copying; this copy is virtually free due to the small size of headers.

Effectiveness of DAMN DAMN is the first system that provides full DMA attack protection with comparable network performance to an *unprotected* system at 100 Gb/s speeds. With a 100 Gb/s NIC, Linux/DAMN achieves 90% of the TCP throughput of an unprotected multi-core system, and only 3% less than the default, partially-secured protection model. Relative to shadow buffers—the only IOMMU protection system with equivalent security guarantees—DAMN provides 7% more throughput using half the CPU cycles; when using only a single core, DAMN's throughput is 2.7× higher.

Non-networking devices DAMN co-exists with prior DMA API-based IOMMU protection schemes (§ 5.3), so all devices remain IOMMU-protected. Most non-networking device classes have low DMA rates, i.e., they perform few DMAs per second. These devices thus suffer negligible overhead even from protection schemes that (un)map DMA buffers from the IOMMU on each DMA. NVMe storage media supports high DMA rates, but the DAMN approach is incompatible with storage devices, as we discuss in detail in § 2.2. Fortunately, however, the DMA rate of NVMe storage media is not as high as that of a multi-gigabit NIC, and we find that shadow buffers can protect NVMe storage media without noticeable overhead (§ 6.5). Overall, DAMN's compatibility with prior protection schemes allows the OS to fully protect itself from DMA attacks by any of the attached devices.

Contributions To summarize, we make three contributions: (1) a conceptually different DMA attack protection model, which relaxes the protection boundary enforced by the DMA API; (2) design and implementation of DAMN, which realizes our new model and offers nearly overhead-free protection; and (3) experimental evaluation of DAMN with 100 Gb/s networking workloads.

2 Scope: attack model, goals & limitations

2.1 Attack Model

The attacker controls some DMA-capable hardware devices but cannot access OS constructs, e.g., to reconfigure the IOMMU. Our focus is protection from *unauthorized* DMAs to target buffers not mapped with the DMA API; attacks

carried out with authorized DMAs, such as tampering with contents of packets, are out of scope. In the case of malicious NICs, such out-of-scope attacks are essentially man-in-the-middle attacks that could also be performed elsewhere on the network. We therefore assume that the system uses other means to protect against such attacks (e.g., encryption and authentication).

The attacker's target is the OS. Peer-to-peer attacks by one device against another [42] are also out of scope; they can be prevented using PCIe Access Control Services (ACS) [42].

Assumptions The OS, device drivers, and the IOMMU are trusted. We assume that the IOMMU is secure, that DMAs cannot be "spoofed" to appear to come from another device, and that the IOMMU blocks all DMAs during OS boot [21, 22], before its page tables are configured.

2.2 Goals and limitations

Goal DAMN addresses the main networking use case of commodity OSes: serving as an endpoint for applications that use a POSIX-like kernel interface, which creates a copy of I/O buffers on the kernel-user boundary.

Zero-copy The DAMN approach does not apply to end-to-end zero-copy paths, such as `sendfile()` or zero-copy IP forwarding. In these cases, the OS falls back to DMA API-based protection.

Kernel bypasses Similarly, we target applications that rely on the OS for managing the networking data plane. Applications that implement their own data plane, bypassing the OS and interacting directly with the device (e.g., with newer OS designs like Arrakis [35], or with mainstream kernel-bypasses mechanisms such RDMA and DPDK [15]) are out of scope. Applications that bypass the kernel's data plane management inherently cannot rely on the kernel for DMA attack protection and must implement their own protection model. It seems, however, that the techniques used in DAMN can be ported to such applications.

Storage media The DAMN approach is incompatible with storage media. Unlike received packets, data received from storage media is cached in the OS page cache. It is impossible to protect a DMA buffer stored in the page cache from TOCTTOU attacks without copying it all. Some applications bypass the page cache by directly accessing the storage media (via the `O_DIRECT` flag to `open()`). In such cases, the OS uses the user's buffers as DMA buffers. This approach is essentially a kernel bypass and thus DAMN cannot be used in this case, for the reasons discussed above.

3 Background

IOMMUs IOMMUs virtualize the address space of devices, from using physical addresses (PAs) to using I/O virtual addresses (IOVAs). The OS maintains per-device page tables that define IOVA-to-PA mappings at page granularity. Mappings also specify the allowed access types—read, write, or both. The IOMMU routes a DMA to the PA that corresponds to the IOVA or blocks the DMA if no valid mapping exists. IOMMUs cache IOVA-to-PA mappings in an *IOTLB*. To destroy a mapping, the OS must remove it from the page tables and invalidate the IOTLB entries associated with the mapping. The OS controls the IOTLB via an *invalidation queue*, a cyclic buffer from which the IOMMU reads commands.[1]

DMA API Device drivers must use the DMA API to manage DMA buffers. Drivers `dma_map` a buffer before initiating a DMA to that buffer, thereby passing ownership of the buffer to the device. Drivers `dma_unmap` the buffer upon DMA completion, thereby regaining ownership of the buffer. The `dma_map` call returns a *DMA address* (let us denote it as α). The driver must configure the device to DMA to α; `dma_unmap` later takes α as its parameter. There are analogous methods to (un)map non-contiguous scatter/gather lists.

4 Motivation: IOMMU protection tradeoffs

Here, we make the case for abandoning the low-level DMA API as the OS layer implementing IOMMU protection of high-speed networking cards. The generality of the DMA API dictates that (1) it provides an overly strict security guarantee, namely that a buffer must be inaccessible to the device once unmapped; and (2) it protects arbitrary buffers, which may reside on the same page as unrelated sensitive data. Both properties inherently create a tension between security and performance, because achieving them requires performing work on each DMA.

We demonstrate this tension by analyzing existing IOMMU protection schemes. We perform all experiments on a machine with two 14-cores Intel Xeon E5-2660 v4 CPUs (28 cores overall), equipped with a dual-port Mellanox ConnectX-4 100 Gb/s NIC. Each port is connected back-to-back to a dedicated traffic generation machine equipped with the same NIC. (§ 6 describes our setup in detail.)

Table 1 summarizes the tradeoffs associated with different protection schemes (further discussed below). The only scheme with negligible overhead is Linux's default scheme, *deferred* protection, which trades off security for performance by delaying (batching) IOMMU unmappings, creating a vulnerability window in which a device can access `dma_unmap()`ed buffers.

The remaining schemes trade off significant performance (e.g., network throughput, CPU utilization, or memory traffic) to provide protection. Despite making these tradeoffs, only one protection scheme, DMA shadow buffers [29], offers *sub-page* protection of DMA buffers, which prevents sensitive data co-located on the buffer's page from being accessible to the device. Sub-page protection in the DMA API, however,

[1] This discussion describes Intel's x86 IOMMU architecture [23], but other architectures have similar designs [2, 4, 19].

iommu protection	*secure subpage*	*secure window*	*perform multi-Gbps*	*perform zero-copy*
partial				
deferred Linux≥4.7 (ATC'15 [34])	✗	✗	✓	✓
strict Linux≥4.7 (ATC'15 [34])	✗	✓	✗	✓
full				
shadow buffers (ASPLOS'16 [29])	✓	✓	✓	✗
DAMN	✓	✓	✓	✓

Table 1. *IOMMU protection-performance tradeoffs. (*Deferred *is the default on Linux.)*

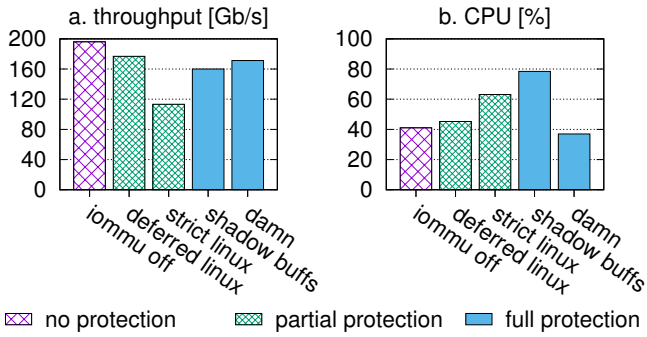

Figure 1. *Protection-performance tradeoffs. Bars show aggregated TCP throughput and CPU consumption of multiple netperf instances.*

comes at the price of additional per-DMA overhead, as it requires copying the entire DMA buffer.

4.1 Linux IOMMU protection

Linux offers two IOMMU protection models with different tradeoffs, *strict* protection and *deferred* protection, which is the default.[2] In both models, `dma_map` allocates an IOVA range for the target buffer, maps the two in the IOMMU page tables, and returns the IOVA to the driver. Likewise, in both models, `dma_unmap` removes these mappings from the IOMMU page tables. Because IOMMU protection works at the granularity of pages, both models provide only *partial* protection, as they allow devices to access sensitive data that might be co-located on pages IOMMU-mapped for a device; this can happen because the mapped buffers are allocated with the standard kernel `malloc`, which can satisfy unrelated allocations from the same page.

Linux's protection models differ in how they invalidate the IOTLB. *Strict* protection synchronously invalidates the IOTLB on `dma_unmap`, guaranteeing that the device can no longer DMA to the buffer. *Deferred* protection batches invalidation operations, invalidating the IOTLB after sufficiently many requests accumulate or every 10 milliseconds, whichever comes first. In the window between a `dma_unmap` and the IOTLB invalidation, the device can thus still access unmapped buffers. Such access enables, e.g., "time of check to time of use" (TOCTTOU) attacks for DMA-writable buffers, such as modifying a packet after it passes firewall checks. DMA-readable buffers access allows devices to steal data placed in unmapped buffers after the OS reuses them.

Deferred protections makes the above security compromises to improve performance. We illustrate these tradeoffs by comparing the performance of the IOMMU protection models using netperf [33], a standard TCP/IP benchmarking tool. We maximize network load by running several netperfs, some transmitting and some receiving traffic. We report the sum of receive and transmit throughputs. The maximum throughput achievable is 200 Gb/s, as the NICs are full-duplex.

[2]We use Linux as a representative study vehicle; other OSes have similar IOMMU protection designs [34].

Figure 1 shows the aggregate throughput and CPU utilization (one core at 100% would be reported as 100% / 28 cores = 3.57% utilization). *Deferred* protection achieves 90% of the throughput obtained without IOMMU protection, with similar CPU utilization. *Strict* protection, on the other hand, achieves only 58% of the unprotected throughput due to the cost of (1) the contended lock protecting the invalidation queue and (2) waiting for the invalidation to complete.

4.2 Shadow buffers

DMA shadow buffers provide full IOMMU protection at byte-granularity. But they trade off extra CPU and memory bandwidth to obtain it. With shadow buffers, the OS restricts device access to a set of permanently-mapped pages, and the DMA API *copies* DMAed data to/from these shadow pages. The DMA API thus never needs to invalidate the IOTLB. However, the overhead of IOTLB invalidation is replaced by the overhead of copying.

Shadow buffers have been shown to sustain 40 Gb/s rates, but with a $\approx$ 2× increase of CPU consumption [29]. The overhead of copying, however, is proportional to the I/O throughput. With faster NICs, the extra CPU cycles and memory bandwidth demands of shadow buffer grow. Therefore, unless the system is over-provisioned in terms of CPU and memory bandwidth, shadow buffers eventually fail to drive the NIC at maximum rate. Figure 1 shows this limitation. Shadow buffers achieve 80% of the unprotected system's throughput, while consuming twice the CPU time (80%). Here, shadow buffers actually exhaust memory bandwidth; see § 6 for details.

The overhead of shadow buffers additionally leaves less CPU and memory bandwidth available to the rest of the system, potentially leading to severe degradation in the execution efficiency of programs co-running on the same machine. Figure 2 depicts an example of this problem. The figure shows the results of running netperf concurrently with a Graph500 BFS loop on different cores. (We describe the experiment in detail in § 6.4.) The *no graph* bar shows the netperf throughput obtained without the Graph500 co-runner on

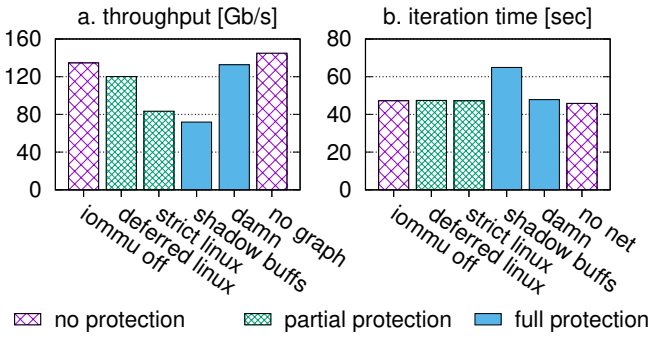

Figure 2. *Netperf throughput (a) and Graph500 BFS iteration time (b), when each benchmark runs alone (rightmost bar) or concurrently (the rest of the bars). Shadow buffers cannibalize CPU and memory bandwidth.*

Linux when IOMMU protection is disabled. Similarly, the *no net* bar shows the speed of the Graph500 program without the netperf tests running. Shadow buffers' resource cannibalization causes a 1.44× increase in Graph500 iteration execution time, and it halves the netperf throughput compared to most other protection schemes.

5 DMA-Aware Malloc for Networking

This section describes the design and implementation of DAMN, our DMA-Aware Malloc for the Networking subsystem. We describe DAMN in the context of Linux, but the same design applies to other monolithic OSes, which have similar network I/O subsystem designs.

DAMN eliminates the overhead of IOMMU mappings maintenance with a similar technique to shadow DMA buffers. Namely, it restricts device access to a set of permanently-mapped pages. But whereas shadow buffers require copying of data to/from these pages, DAMN offers a conceptually different model that allows the network stack to store data directly in these pages.

To the networking stack and drivers, DAMN appears as a memory allocation API, which additionally allows specifying the device permitted to access the allocated buffer and the type of access allowed. DAMN leaves protection of buffers not allocated through it to the DMA API, and is thus compatible with any DMA API-based protection scheme. Deploying DAMN can therefore be done gradually, focusing on performance-critical flows first. Moreover, non-network devices maintain their DMA API-based IOMMU protection.

5.1 Interface

DAMN's memory allocation interface adheres to the kernel memory allocation API. The kernel provides two allocation functions relevant for networking: `kmalloc` and `alloc_pages`. The `kmalloc` function returns the virtual address of a newly allocated, physically contiguous, 8-bytes aligned object that is smaller than a page size. Larger allocations are usually done with `alloc_pages`, which allocates a sequence of 2^k physically contiguous pages ($k \geq 0$ is supplied by the caller) and returns a pointer to the `page` structure of the first page in the sequence. A `page` structure holds the OS metadata for a physical page, and each page is associated with its own structure. The structures are housed in an array, allowing constant-time conversions between physical addresses and their `page` structures.

The DAMN interface is specified in Table 2. It consists of the functions `damn_alloc` and `damn_alloc_pages` (and their `damn_free` and `damn_free_pages` counterparts). They are analogous to the kernel allocation API, but DAMN adds to them two arguments: a `device` pointer that specifies which device is given permission to DMA to/from the buffer being allocated, and a `rights` bitmask that specifies the permitted access rights for that device (read, write, or both). If the `device` pointer is NULL, DAMN falls back to the standard kernel API. (We explain in § 5.7 how this can occur.)

Callers of `damn_free` and `damn_free_pages` do not always know the device and access rights associated with the buffer they are freeing, in contrast to allocating callers. Therefore, the DAMN API does not specify passing it, and DAMN must internally look up this information (§ 5.5).

5.2 TOCTTOU protection

Having network receive buffers that are permanently writable to the device raises concern that the device might exploit this access to launch TOCTTOU attacks. We observe, however, that most received bytes flow opaquely through the network stack until being copied to a user-level buffer and out of the device's reach. We thus need to protect packet headers, and handle corner cases in which the OS does access packet data (e.g., a firewall rule that inspects HTTP headers). Any writes the device makes to the other bytes have the same effect as valid writes performed while the packet was `dma_map()`ed.

When a driver processes a received packet, it allocates a *socket buffer*, or `skbuff` in Linux terminology.[3] OS code must use a special API to access `skbuff` data, because this data might be held with a non-contiguous set of buffers (e.g., for avoiding copying of scatter/gather lists). We interpose on this API, and whenever the OS first accesses a byte range, we copy it into a new buffer—inaccessible to the device—and adjust the `skbuff` accordingly. We expect, as explained above, to copy only header bytes in the common case, which incurs negligible overhead. In fact, several Linux drivers already copy packet headers when constructing receive `skbuffs`.

5.3 DMA API compatibility

Drivers currently translate buffer addresses to IOVAs using the DMA API, with `dma_map` returning a buffer's IOVA. To facilitate easy adoption of DAMN in commodity OSes, we

[3] Other OSes have similar structures, e.g., BSD mbufs [31].

operation	*description*
`damn_alloc` (*dev*, *rights*, *s*)	Allocates an *s*-byte buffer that is DMA-accessible to *dev* as specified in *rights* (read/write). Returns the virtual address of the buffer.
`damn_alloc_pages` (*dev*, *rights*, *k*)	Allocates a sequence of 2^k physically contiguous pages that are DMA-accessible to *dev* as specified in *rights* (read/write). Returns the `struct page` of the first page in the sequence.
`damn_free` (*addr*)	Deallocate the DMA buffer whose virtual address is *addr*.
`damn_free_pages` (*page*, *k*)	Deallocate the 2^k contiguous physical pages that start at *page*.

Table 2. *DAMN's memory allocation API (§ 5.1).*

do not wish to modify the DMA API, nor require driver changes for DAMN-allocated buffers. Such buffers thus go through the standard flows in the networking stack, and the OS will eventually attempt to `dma_map`/`dma_unmap` them. We interpose on `dma_map`/`dma_unmap` calls and check if they target a DAMN-allocated buffer (§ 5.5 details how this is done); if not, we fall back onto the standard DMA API (and the IOMMU protection scheme it implements).

For DAMN buffers, we perform the following actions. In `dma_map`, DAMN-allocated buffers have long-lived IOMMU mappings, and so there is no need to create a new IOMMU mapping for them. Instead, we look up the buffer's IOVA (§ 5.5 explains how this is done) and return it. In `dma_unmap`, we determine whether the argument IOVA is associated with a DAMN-allocated buffer by examining the MSB of the IOVA (see § 5.4). If so, there is no need to tear down any IOMMU mappings and we immediately return from the `dma_unmap` call; the buffer itself will be freed later by the networking subsystem. Otherwise, the legacy `dma_unmap` code precedes as usual.

5.4 Allocator

DAMN's allocation interface is backed by a set of memory allocators, one for each device and DMA access rights (read-only or read/write) combination, called *DMA caches*. A DMA cache is named thus because it is essentially a caching layer on top of the kernel page allocator. Each page that a DMA cache allocates is mapped in the IOMMU to a specific device and access rights, and it is used to satisfy networking allocation requests for that type of DMA buffer. The DMA cache design is based on *magazines* [8], a general method for highly scalable OS resource allocation. Magazines add a per-core scheme for caching objects allocated by an arbitrary resource allocator—the kernel's page allocator, in our case.

The DMA cache subsystem supports allocations of up to 64 KiB. This maximal value is determined by the implementation of the network stack and the network device driver. This parameter can be tuned as required for other OS and device driver combinations.

Next, we describe the logical organization of the DMA cache, the process of satisfying an allocation request, and the implementation of the per-core physical page caches.

DMA cache organization For every device and access rights combination (read/write), the DMA cache is organized as follows. The cache has a two-level hierarchy. The bottom level maintains a cache of *chunks*, which are sets of C physically contiguous pages, where C is big enough to satisfy the maximal supported buffer size. (Our Linux implementation thus uses $C = 16$, for 64 KiB.) The chunks are mapped in the IOMMU with the appropriate access rights for the device The top level consists of a per-core allocator that maintains a chunk, acquired from the bottom level, and carves it up in order to satisfy DAMN allocation requests.

Top-level allocation I/O allocation patterns typically consist of a producer/consumer flow, in which one core allocates a DMA buffer and hands it to the device, and the buffer is later deallocated by whichever core handles the interrupt raised by the device to notify that the DMA has completed. We thus specialize our allocation scheme to this use case; in particular, unlike standard `malloc` implementations, we do not optimize for rapid successions of allocating and deallocating the same buffer.

Our allocation scheme uses a simple "bump pointer" allocator, which satisfies allocation requests from the underlying chunk by "bumping" a pointer, until reaching the end of the chunk; at this point, a new underlying chunk must be obtained from the bottom level of the DMA cache. Each allocation increments a reference count on the chunk, and each deallocation decrements the reference count, returning the chunk to the bottom level of the DMA cache if its reference count reaches zero. The reference count is manipulated with atomic instructions, using the existing OS page reference-count interface, which stores the count in the `page` structure of the first page in the chunk.

Our Linux implementation of DAMN uses the OS-provided *page frag* API which essentially implements the scheme described above; on other OSes, this scheme can be implemented from scratch (it requires less than 100 LOC).

The DMA cache contains two of these bump pointer allocators for each core, one for non-page-aligned allocations requests (`damn_alloc`), and one for page-aligned allocation requests (`damn_alloc_pages`). Finally, for each device and access rights, we maintain a DMA cache for each NUMA

domain. This design leverages the NUMA-awareness capabilities of the OS page allocator, allowing DMA caches to be populated with pages allocated from the calling core's NUMA domain's memory.

Top-level deallocation The common case of `damn_free` and `damn_free_pages` is to decrease the reference count of the appropriate chunk, as described above, and return. If, however, the reference count reaches zero, the chunk must be placed into a magazine; this requires looking up the appropriate DMA cache (i.e., the device, access rights, and NUMA domain combination). Looking up this information requires additional data structures, which are discussed in § 5.5.

Bottom-level chunk cache The bottom level of the DMA cache, which caches physical page chunks, is implemented using *magazines* [8], proposed as a generic method for highly scalable OS resource allocation. Magazines add a per-core scheme for caching *objects* allocated by an arbitrary *resource allocator*. In our case, the objects managed by the magazines are the physical page chunks and the underlying resources allocator is the kernel's page allocator. Next, we provide background on magazines, and then describe the extensions we performed to obtain high performance on modern NUMA architectures available today.

A *magazine* is an M-element per-core cache of objects, maintained as a LIFO stack. Being per-core, magazine manipulation requires no synchronization and is extremely fast. Conceptually, a core attempts to allocate and deallocate from its magazine first. If it fails—namely, it tries to allocate from an empty magazine or to deallocate into a full one—then it must access a global shared *depot* of magazines, protected by a global lock, to exchange a full magazine with an empty one (or an empty magazine with a full one). The depot falls back onto the underlying resource allocator if it cannot return a full magazine; it fills a new magazine with freshly allocated objects and returns it. The actual magazine replenishment policy is more sophisticated, and guarantees that a core can satisfy at least M allocations and at least M deallocations without accessing the depot.

Physical DMA cache organization For performance reasons, we maintain *two copies* of the above described structures (i.e., the two bump allocators and per-core magazines). The two copies are needed to avoid the cost of disabling interrupts. Because networking allocations can come from regular context or interrupt context, it is possible for a DMA cache operation (called from a regular context) to be interrupted, only to have the interrupt handler also invoke a DMA cache operation. The standard solution to handle this race would be to disable interrupts when a DMA cache operation is invoked. We find, however, that interrupt disabling has measurable negative impact on I/O throughput. Therefore, our solution is to maintain the logical DMA cache hierarchy as two physical copies, one for standard context and one for interrupt context. Having two per-context DMA caches does not double the amount of memory consumed compared to having a single cache, as the buffers allocated in each context are typically disjoint: receive allocations occur in interrupt context and transmit allocations occur in standard context.

Initialization A DMA cache is initially empty. Each time the depot allocates a chunk, it zeroes the allocated chunk and maps it in the IOMMU. We partition the 48-bit IOVA address space, using the MSB of the IOVAs, to distinguish between IOVAs used by DAMN-allocated buffers and IOVAs used by the standard DMA API, as explained in § 5.5.

Responding to OS memory pressure Many OS subsystems maintain caches of objects, and modern OSes therefore provide a standard interface for the OS to request a cache to release memory back to the system if memory pressure occurs (e.g., the Linux *shrinker* interface [12]). While our research prototype does not implement this functionality, it is straightforward to support it: when memory pressure occurs, DAMN simply has to release chunks that reside in magazines or in the depot back to the OS. Such chunks can safely be released, since they do not contain allocated or DMA-mapped buffers.

5.5 Mapping from addresses to DAMN metadata

Let B be an I/O buffer with virtual address B_{va} and IOVA B_{iova}. With the help of its metadata, DAMN needs to be able to perform the following operations: (i) in `dma_unmap`, given B_{iova}, it needs to be able to determine to do nothing if B was allocated by DAMN, or to fall back on the standard DMA API otherwise; (ii) in `dma_map`, given B_{va}, it likewise needs to either decide to fall back, or (if DAMN allocated B) to find B_{iova} and return it to the caller; and (iii) in `damn_free` or `damn_free_pages`, given B_{va}, it needs to be able to find B's allocator and decrease the corresponding reference count.

A natural solution for the tasks that get B_{va} as input, (ii) and (iii), would be to store this information in the `page` structure of the pages allocated by DAMN. But enlarging the `page` structure is considered a major OS change, which might have unintended performance regressions. So we avoid modifying the `page` structure by leveraging the *compound pages* [13] functionality of the OS. A compound page is an OS-recognized grouping of at least two contiguous physical pages. The `page` structure of the first page in the sequence (the *head page*) represents the rest of the pages, making several fields in the `page` structures of the remaining *tail pages* unused, which provides us with room to store DAMN's data, notably, B_{iova}.

A remaining question is how to identify a page as a DAMN page whose `page` structure holds a valid B_{iova}. All tail pages point to the compound's head `page`, which has predetermined semantics that we are not allowed to change. The same applies to the second `page`. We therefore add a flag F to the third `page` structure, indicating that this is a DAMN compound. Thus, given B_{va}, DAMN checks its `page` to see

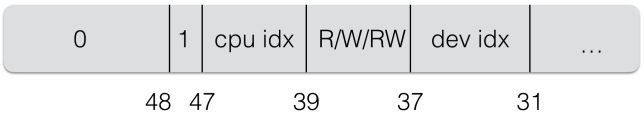

Figure 3. *DAMN's IOVA encoding (bit 47 is always 1).*

if its a compound whose third page has F set, in which case the corresponding B_{iova} is valid.

To accomplish task (iii), when B_{iova} is generated, DAMN encodes in its top bits: B's allocating core, the access rights, and the device, as depicted in Figure 3. This encoding thus allows DAMN to identify B_{iova}'s allocator and deallocate B.

5.6 Security

We argue the security of DAMN by considering two cases: transmitted (TX) and received (RX) data.

TX Transmitted data is mapped with read access rights. The only concern, therefore, is whether the device can read sensitive information. DAMN zeroes memory it obtains from the OS page allocator. We also assume that once allocated, a packet buffer is used to hold only packet data. It follows that any page managed by DAMN will contain zeroes or data that the OS has allowed the device to read at some point, and thus cannot contain sensitive information.

RX Receive security follows from the property that any packet bytes accessed by the OS are copied. The OS thus observes a consistent view of a packet that the device cannot change under its feet. Bytes that are not copied in this way are copied to the user's buffer, where they cannot be modified by the device. It follows that any byte read from a packet could have well been written while the buffer was `dma_map()`ed.

5.7 Deploying DAMN in Linux

Our DAMN implementation consists of ≈500 LOC: ≈250 LOC for the DMA cache implementation and ≈250 LOC for the magazines implementation.

We are able to deploy DAMN in the Linux TCP stack without making intrusive changes to the entire stack. Rather, we extend the internal `__alloc_skb` function, which allocates `skbuff` structures, by adding a device argument. The `__alloc_skb` function already receives a `flags` argument that specifies, among other things, whether the `skbuff` will be used for received or transmitted data. We rely on this flag to define the access rights (read for TX, write for RX).

The `__alloc_skb` function has three classes of callers. First, it is invoked by the `skbuff` code itself, when `skbuff` structures are copied or cloned. In such case, the device argument can be extracted from the source `skbuff`. Second, it is invoked by device-API code for allocating of packet buffers. There, too, a device pointer is readily available. Finally, `__alloc_skb` is invoked indirectly by the networking stack through an `alloc_skb` method. We modify `alloc_skb` to pass a NULL device to `__alloc_skb`, and create a new method, `dma_alloc_skb`, for use by DAMN-aware flows.

Having done this, deploying DAMN in the TCP code only requires modifying it to invoke `dma_alloc_skb` and passing a device pointer, which is available in the TCP socket. To complete the deployment, we additionally modify the NIC driver to use DAMN for its allocations (2 LOC change). The entire effort took a graduate student who was not familiar with the networking code 4 days to complete, and added ≈300 LOC to the kernel: 150 LOC in the `skbuff` code and 150 in protocol and socket code.

6 Evaluation

Our evaluation seeks to analyze the following issues:

- § 6.1 Can strict, byte-granularity protection be achieved without hampering performance? To answer this question, we quantify the performance/security tradeoffs in DAMN and prior DMA attack protection schemes. We extend the scope of the evaluation to more challenging workloads than those evaluated in prior work [29, 34], including >100 Gb/s streaming data.
- § 6.2 What are the overheads imposed by DAMN's TOCTTOU protection, which copies packet bytes that are accessed by the OS?
- § 6.3 What is the memory overhead imposed by DAMN?
- § 6.4 What is the effect that the different protection schemes have on the performance interaction between the networking subsystem and unrelated programs running on the same machine?
- § 6.5 Is the relegation of IOMMU protection of non-networking device classes to prior schemes justified, or do we lose performance as a result of this choice?

Evaluated systems We implement DAMN in Linux 4.7. We compare DAMN (*damn*) to Linux 4.7 without IOMMU protection (*iommu-off*), and to Linux 4.7 in *deferred* and *strict* IOMMU protection modes. (Linux 4.7 contains the IOMMU protection scalability improvements of Peleg et al. [34].) We further compare to Linux 4.7 with shadow DMA buffers [29] (*shadow buffs*), the only other DMA attack protection scheme that offers strict, byte-granularity, protection.

Experimental setup We run the evaluated kernel on a 28-core Dell PowerEdge R730 machine equipped with a dual-port Mellanox ConnectX-4 100 Gb/s Ethernet NIC. The machine is a dual-socket server with 2 GHz Intel Xeon E5-2660 v4 (Broadwell) CPUs, each of which has 14 cores. The machine has 128 GiB of memory, with four 16 GiB 2400 MHz DDR4 DIMMs per socket. For traffic generation, we use two 16-core Dell PowerEdge R430 machines, also equipped with ConnectX-4 100 Gb/s NICs. Each such client is connected back-to-back to one of the ports on the evaluated server NIC. However, the PCIe 3.0 bus on the server limits the server's receive/transmit bandwidths to 128 Gb/s [30, Section 7.4] (In

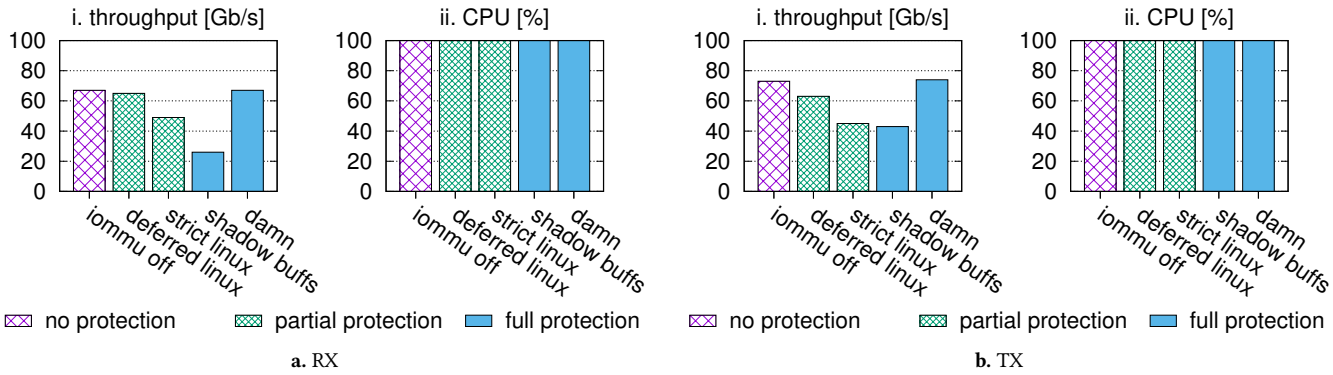

Figure 4. *Single-core TCP throughput and CPU utilization of netperf TCP-STREAM (100% CPU is one core).*

practice, we do not manage to obtain more than 106 Gb/s.) The clients are dual-socket machines with 2.4 GHz Intel Xeon E5-2630 v3 (Haswell) 8-core CPUs and 32 GiB of memory. We disable hyperthreading on all machines, and configure all machines for maximum performance, with Intel Turbo Boost (dynamic clock rate control) disabled. The traffic generators run with the IOMMU disabled.

6.1 Performance vs. security tradeoffs

Methodology We follow the methodology we used in our previous IOMMU study [29]. To even the load between cores, we configure the NIC drivers to use one receive ring per core with even distribution of interrupts between cores. We run the benchmarks for 60 seconds on an otherwise idle system and report averages of 10 runs.

Benchmarks We use netperf [33], a standard networking benchmarking tool, to evaluate TCP/IP throughput. We evaluate both receive (RX) and transmit (TX) throughput, by having the evaluation machine play the role of the netperf receiver/transmitter. Finally, we study an application workload using the memcached [17] key-value store. To avoid lock contention in the multi-threaded version of memcached, we run a separate memcached instance on each core. We provide further details on each experiment as we present it.

Single-core TCP throughput We measure the throughput obtained by netperf's TCP-STREAM, such that all activity in the system (including interrupt processing) is restricted to run on a single core. In this test, the measured application repeatedly transmits or receives a 16 KiB buffer to/from a TCP socket without waiting for a response, and it reports the average throughput at the end of the execution. We concurrently run four netperf instances on a single core, using both NIC ports, to bring the tested core's utilization to 100%. The loader machine at the other end is not similarly constrained and is free to use as many cores as it pleases.

When transmitting, by default, Linux uses the NIC's TCP Segmentation Offload (TSO) hardware, allowing the network stack to aggregate buffers that the user sends into segments as big as 64 KiB, before handing them to the NIC. The NIC then breaks these segments into MTU-sized Ethernet frames for transmission on the wire. We configure our machines to also use the NIC's Large Receive Offload (LRO) hardware, so the NIC at the receiving machine symmetrically aggregates the frames into larger TCP segments of up to 64 KiB, before handing them to the receiving kernel. We set the MTU size to be 9000 bytes ("jumbo frames"), and we turn Ethernet flow control on to make it lossless.[4]

Figure 4a shows the RX throughput and CPU utilization, in which the evaluation system acts as the receiver. All protection modes are bottlenecked by the CPU as intended. The throughput obtained by *iommu-off*, *deferred* and *damn* is 65–67 Gb/s, whereas *strict* obtains only 50 Gb/s, because it spends many CPU cycles invalidating the IOTLB, which is a costly hardware operation.[5] *Shadow buffs* obtain the lowest throughput, 26 Gb/s (*damn*'s throughput is 2.7× higher), because the CPU overhead of this mode eats into the cycles required to drive the NIC. It appears that in our workload, *shadow buffs* hit their scalability limit—compared to the other modes, they do much worse than at 40 Gb/s rates [29].

Figure 4b shows the throughput and CPU utilization in the TX test, in which the evaluation system acts as the sender. The throughput of *iommu-off* and *damn* increases by about 10% to 73–74 Gb/s, whereas the throughput of *deferred* and *strict* slightly decreases. The throughput of the *shadow buffs* mode improves by 1.7×, but it is still the worse-performing setup. We verified that the throughput difference between RX

[4]This hardware configuration, coupled with running multiple netperf instances that utilize multiple NIC ports, makes our setup *much* more performant than in previous work [26, 27, 29], leading to higher single-core throughput than is usually reported.

[5]In comparison, a single RX netperf instance using only one port and running with the default Linux configuration (1500 bytes MTU and LRO off) will approach 20 Gb/s if the IOMMU is turned off. This throughput will further drop to around 5 Gb/s if the IOMMU is turned on and *deferred* is used (or half that much if *strict* is used) because the core will have to handle many more incoming packets that are much smaller.

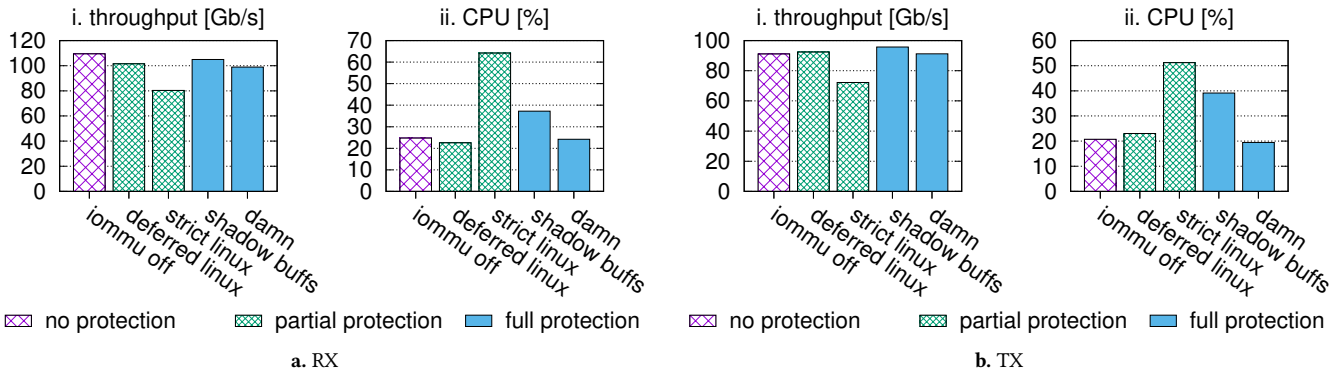

Figure 5. *Multi-core TCP throughput and CPU utilization of netperf TCP-STREAM (100% CPU is 28 cores).*

and TX for this mode occurs because RX has a much bigger memory footprint, since it must post many more buffers for DMA in expectation of incoming traffic. When artificially enlarging TX's memory footprint to be as big as that of RX, the performance of the two becomes identical.

Multi-core TCP throughput Here, we perform netperf RX/TX tests, only this time with 28 netperf client/server instances (one per core) on the server machine. We report the aggregate throughput and CPU utilization over all cores (i.e., one core at 100% CPU would be reported as 100%/28 = 3.57% CPU utilization).

Figure 5a shows the results of the RX test. Here, the NIC becomes the bottleneck, and all variants but *strict* achieve at least 100 Gb/s. *Damn*, *iommu-off* and *deferred* have comparable CPU utilization. The only variant failing to sustain 100 Gb/s is *strict*, whose throughput throttles at 80 Gb/s and its CPU use spikes to 64%. In multi-core workloads, *strict* suffers not only due to the high cost of IOTLB invalidations, but also from the bottleneck created by the IOTLB invalidation lock [29, 34]. *Deferred* alleviates these bottlenecks, but trades off security to do so. *Shadow buffs* use 37% overall CPU time, 1.5× more than *damn*, *deferred*, and *iommu-off*. The results of the TX test (Figure 5b) show similar trends.

Beyond 100 Gb/s Here, we stress the system—and the protection schemes—further, by running RX and TX netperf TCP-STREAM tests simultaneously, for a peak theoretical bidirectional bandwidth of 200 Gb/s. (This corresponds to the experiment discussed in § 4.1.) Figure 6 shows the results. *IOMMU-off* obtains 196 Gb/s, *deferred* 176 Gb/s, and *damn* 171 Gb/s (3% worse than *deferred*). The *strict* protection model obtains 113 Gb/s, which is 44% worse than *damn*.

Bidirectional traffic has a detrimental effect on TCP, as ACK segments compete with data segments for resources. Therefore, compared to previous experiments, we need more flows, and so more cycles, to drive 100 Gb/s in each direction. For example, RX/*iommu-off* requires 100% CPU usage of a single core to sustain 67 Gb/s (Figure 4a), whereas the corresponding bidirectional experiment requires 41% of a 28-core machine to sustain 196,Gb/s (Figure 1), namely, the equivalent of 11.5 cores.

Shadow buffs, which implement the only protection scheme with comparable security to *damn*, lag behind with 160 Gb/s, namely, 0.8× that of *iommu-off*. As shown in Figure 1, *shadow buffs* consume only 80% of the CPU resources, which is about twice that of *iommu-off*, *damn*, and *deferred*, but still not 100%. The reason that *shadow buffs* do not maximize throughput is that the OS throttles its network I/O rate because the NIC does not empty its rings sufficiently fast. Throttling occurs because of memory bandwidth exhaustion, as shown in Figure 6 (measured using Intel's performance counter monitor tool [40]). With *shadow buffs*, the system's memory bandwidth use is ≈80 GB/s, which is the advertised limit of the processor's memory controller, through which the NIC accesses memory. The resulting memory bottleneck prevents the NIC from operating its rings at a fast enough rate.

In contrast to the previous TCP throughput experiments, here *damn* fails to achieve the same throughput as *iommu-off*. We evaluate variants of *damn* in which we modify parts of the design, in an attempt to tease apart the sources of the throughput difference. Table 3 shows the results. We first evaluate a variant that increases the effectiveness of the IOTLB. In this variant, we map IOVAs using "huge" IOVA pages, which enables a single IOTLB entry to cache a 2 MiB IOVA range. However, *damn*'s policy of encoding metadata in the high bits of the IOVA (§ 5.5) causes IOVAs of buffers allocated from different DMA caches to fall into different huge pages. As a result, DAMN uses more huge IOVA pages than are necessary for mapping the allocated DMA buffers, which makes the IOTLB less effective. Therefore, in this variant we allocate IOVAs densely, using the minimum number of huge pages (which means that we do not encode metadata in the IOVAs and thus cannot free them, and so this variant is useful for analysis only).

Making the IOTLB more effective increases *damn*'s throughput by 13 Gb/s (6.5% of *iommu-off*). Next, we test *damn* with

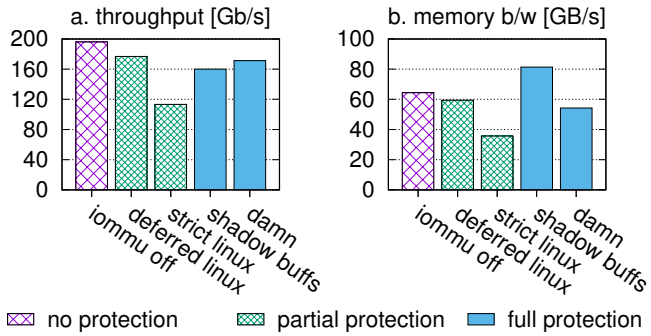

Figure 6. *Throughput and memory bandwidth in multi-core netperf TCP-STREAM test using bidirectional traffic. (Same experiment as depicted in Figure 1; the left sides of both figures are identical.)*

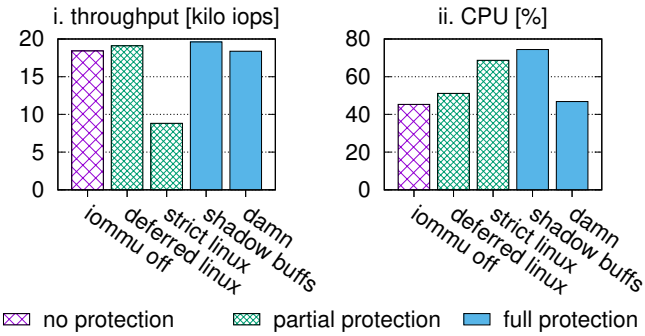

Figure 7. *Memcached: aggregated throughput and CPU utilization (of 28 concurrent instances).*

configuration	*Gb/s*
damn	170 (86.3%)
damn + huge iova pages + dense iova range	183 (92.9%)
damn without iommu	192 (97.5%)
iommu-off	197 (100%)

Table 3. *Factors affecting the throughput difference between* damn *and* iommu-off *in the multi-core bidirectional netperf TCP-STREAM test (Figure 6). The values in the parentheses show the throughput relative to the* iommu-off *configuration.*

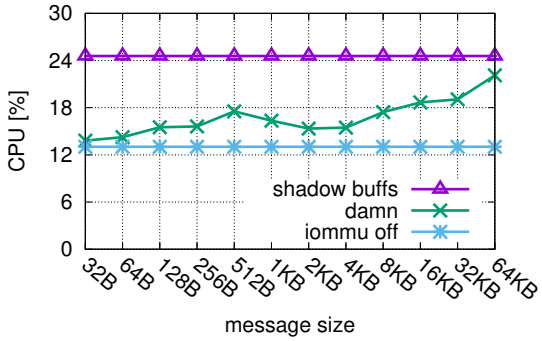

Figure 8. *Impact on CPU utilization of accessing a varying fraction of packet data bytes under* damn *when running XOR netfilter.*

the IOMMU disabled, which eliminates all IOMMU hardware-related overheads and boosts throughput by an additional 9 Gb/s (4.5% of *iommu-off*). This leaves a 5 Gb/s (2.5%) gap relative to *iommu-off* that we are unable to explain. We speculate that the source of the remaining throughput difference is implementation artifacts that could be addressed by profiling and further optimizing *damn*.

memcached The memcached benchmark consists of several memslap [1] instances generating load on 28 memcached instances that run on the evaluation machine. We use a workload with 50%/50% GET/SET operations of 512 KiB keys and values. We use non-default key/value sizes to obtain significant network traffic that stresses the protection schemes. Figure 7 shows the aggregated throughput (memcached operations) obtained. *Damn*, *shadow buffs* and *deferred* obtain comparable throughput to *iommu-off*, but *shadow buffs* use about 1.6× the CPU as compared to *damn* and *iommu-off*. Due to the IOTLB invalidation rate caused by TX traffic, *strict* obtains half the throughput of the other protection schemes, with *strict* requiring 70% CPU time to drive only 8816 TPS.

6.2 Impact of DAMN TOCTTOU protection

DAMN defends against device TOCTTOU attacks on RX packets, whose buffers are writable by the device, by dynamically copying packet bytes that the OS accesses to remove them from the device's reach. In the standard networking workloads evaluated thus far, DAMN only copies packet header bytes. Here, we use a custom benchmark to evaluate the performance impact of TOCTTOU-defense copying.

We use the Linux `netfilter` [32] framework to register a callback function with the network stack that gets passed each incoming TCP segment destined to the host. Our callback accesses a configurable fraction of the packet's data bytes; it is invoked after segment reassembly, and so can access up to 64 KiB of data. We model efficient segment processing by simply XORing the accessed bytes with a constant value. Data-accessing workloads that are more computationally heavy, such as encrypting tunnels, incur overheads high enough to make the resulting throughput an order of magnitude lower than line rate, so all protection schemes have negligible overhead and thus become indistinguishable.

Figure 8 compares the CPU consumption of *iommu-off*, *shadow buffs*, and *damn* when running concurrent netperf RX TCP-STREAM throughput tests on all 14 cores of a CPU populating one socket of the test machine. (We show only CPU use because all variants achieve the same throughput.) The lightweight XOR processing does not add measurable overhead to *iommu-off* and *shadow buffs*, as indicated by their CPU consumption, which remains fixed.

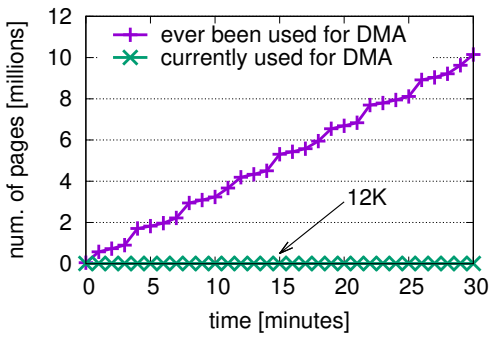

Figure 9. *Pages used to hold DMA I/O buffers in stock Linux. Over time, many pages hold DMA buffers, potentially increasing the vulnerability of partial protection models.*

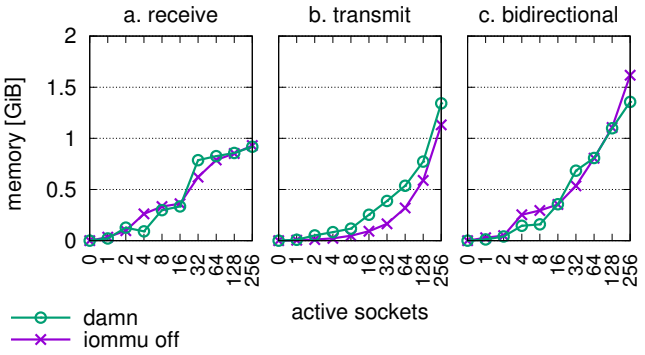

Figure 10. *Kernel memory usage during multi-core netperf TCP-STREAM tests.*

As expected, *Damn* initially resembles *iommu-off*, and then gradually, as more data is being accessed (and thus copied), *damn*'s CPU use grows until it approaches *shadow buffs*. It remains below 16% (1.2× of *iommu-off* and 0.6× of *shadow buffs*) up to 1 KiB of copying, approaching 18% at 16 KiB (1.5× of *iommu-off* and 0.75× of *shadow buffs*). Even when copying all the segment, *damn*'s CPU consumption is 10% lower than that of *shadow buffs*, which copy to arbitrary `kmalloc()`ed kernel buffers that are colder in the cache.

Given that typical packet inspection workloads (e.g., firewalls or virtual switches) are unlikely to access more than 1 KiB of data, it seems *damn*'s TOCTTOU protection imposes only modest overheads compared to an unprotected system.

6.3 Memory consumption

With DAMN, the network stack uses an allocator that permanently binds buffers to the I/O subsystem, to be exclusively used for networking until global memory pressure prompts the OS to invoke the shrinker and reclaim unused memory (§ 5.4). It is therefore important for DAMN to reuse its I/O buffers effectively and allocate more of them only when the aggregated size of networked data exceeds the memory capacity currently owned by DAMN.

Conversely, as illustrated in Figure 9, stock Linux need not—and in fact does not—systematically reuse I/O buffers for the purpose of DMA. Rather, it removes their mappings from the associated I/O page tables at `dma_unmap`, thus making them safe to use for other purposes. In the figure, we show the number of pages ever mapped for DMA vs. the number of pages currently mapped for DMA. The measurements are taken over a period of 30 minutes while running four netperf instances alongside an iterative kernel compile job, which stresses the kernel allocator. Memory used for DMA buffers remains constant, requiring less than 50 MiB throughput, whereas the cumulative number of pages that ever held network DMA buffers monotonically increases.

The memory capacity required to house all networking data is dictated by the running applications. Roughly speaking, it is the sum of all the receive buffers populating RX rings ready to store incoming data, plus all socket buffers ready to absorb application write operations. As long as a socket buffer is not exhausted, the application that writes into it will continue to execute. But if the buffer becomes full, the OS will suspend the writing application until enough bytes are sent and sufficient room becomes available.

To evaluate the effectiveness of DAMN in terms of memory consumption, we run multiple netperf TCP STREAM instances on multiple cores for a period of 25 seconds. During the run, after each second, we record the overall system memory usage (using Linux's `/proc/meminfo`), and, at the end of the run, we compute the average consumption using the 25 measurements. Figure 10 compares the average memory usage of *iommu-off* and *damn*, using a set of experiments that are configured to run with an increasing number of concurrent netperf instances. We report results for RX-only, TX-only, and bidirectional streams (i.e., with half of the netperfs receiving and the other half transmitting). Because *damn*'s DMA cache recycles allocated memory, *damn* essentially consumes only the memory required to hold the workload's networking data. Consequently, *damn*'s memory usage is comparable to that of *iommu-off*. The difference is at most 270 MiB, and, except for the TX workload, neither system is consistently better.

Memory usage with *damn* and *iommu-off* differs once the experiments complete (not shown). With *iommu-off*, system memory usage drops back to near its starting level before the experiments, with a ≈300 MiB increase. With *damn*, memory usage remains at its level during the experiment, as the networking buffers remain in the DMA cache, mapped in the IOMMU. Memory will be unmapped and released back to the system only when the kernel invokes DAMN's shrinker, if memory pressure occurs.

6.4 Interaction with other applications

We evaluate how the network stack affects an unrelated, non-networking program running on the same machine, and vice versa. The results are discussed in § 4.2 and depicted in Figure 2. Here, we detail the setup used for the experiment, and highlight *damn*'s performance.

We run the bidirectional netperf TCP-STREAM test on 4 cores, 2 per CPU. The remaining cores run 3 instances of the Graph500 BFS loop, each using 8 cores (again divided equally between the two CPUs), on a graph problem with 2^{20} vertices and average vertex degree of 256. We measure the throughput obtained by the TCP-STREAM test and the average execution time of the BFS algorithm, measured in seconds/iteration.

The main takeaway from Figure 2 is *damn*'s advantage over *shadow buffs*. As it avoids the extra copy induced by *shadow buffs*, it curbs the completion of netperf and Graph500 over memory bandwidth, allowing one to perform as if the other is not present. In contrast, *shadow buffs* generate memory pressure that adversely affects both, reducing netperf's throughput and increasing Graph500's runtime.

6.5 Protection against NVMe DMA attacks

The DAMN approach is incompatible with storage devices (§ 2.2). However, DAMN is compatible with prior DMA attack protection schemes, which can therefore be used for storage devices, complementing DAMN. To determine the effectiveness of this approach, we evaluate the prior schemes on a storage device with high DMA rates. We use an Intel DC P3700 400 GiB NVMe SSD installed in a Dell R430 server. The server is equipped with two Intel Xeon E5-2650 v4 (Broadwell) 12-core CPUs and 64 GiB RAM. To maximize the DMA rate, we use the `fio` [6] benchmark (version 2.2.10). We run 12 `fio` threads that each perform asynchronous direct sequential reads from the device. Using direct I/O bypasses the page cache, so that the benchmark interacts directly with the device. Each test runs for 60 seconds on an otherwise idle system, and we report an average of three runs.

Figure 11 depicts the CPU utilization and throughput in terms of I/O operations per second (IOPS), as we vary the block size used for reads. With a 512-byte block size, we reach the maximal possible IOPS rate of the disk (≈900 K IOPS). In all tests the NVMe disk is the bottleneck. Tests with larger block sizes are limited by the maximal disk throughput (≈3.2 GiB/sec). Thus, no protection scheme throttles the device speed, even when hitting the maximal IOPS limit. Moreover, all protection schemes but *strict* execute the benchmark with roughly the same CPU utilization. In the 512-byte block test, *strict* consumes 2× the CPU as the other schemes, but its CPU consumption is comparable for larger block sizes. Importantly, *shadow buffs*' performance is essentially identical to *iommu-off*, both in IOPS and CPU utilization. This supports our premise that *shadow buffs* can be used for DMA attack protection from high-speed storage media, but that networking requires a specialized solution.

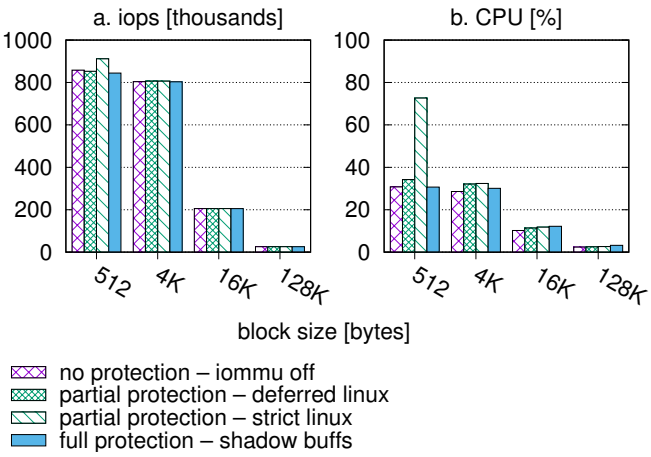

Figure 11. *Multi-core fio/NVMe I/O rate and CPU usage.*

Interestingly, with 512-byte blocks, *strict* achieves 6% higher IOPS than the other schemes. As *strict* I/O submission is slower, there are more opportunities for the device's completion queue to fill up and then be processed at the end of I/O submissions. The faster I/O submissions in the other protection schemes leave less time for completions to accumulate, leading to more completion interrupts being generated.

7 Conclusion

Defending against NIC DMA attacks at the DMA API level requires systems to compromise on either performance or security. We instead propose to provide protection by: (1) modifying the network stack to use a new allocator, denoted DAMN, which manages buffers that are permanently mapped in the IOMMU (until memory pressure occurs); (2) copying DMAed data from these buffers into memory inaccessible to the NIC when the data is first accessed; and (3) avoiding the overhead of this extra copy by leveraging the fact that standard network stacks copy data at the user-kernel boundary in any case. DAMN thus provides full, byte-granularity IOMMU protection with performance rivaling that of an unprotected system. Compared to DMA shadow buffers (the only IOMMU protection scheme with equivalent security guarantees), DAMN provides 7% higher throughput using half the CPU. When constrained to use a single core, DAMN's throughput becomes 2.7× higher than that of shadow buffers.

Acknowledgments

We thank Nadav Amit for many helpful discussions. This research was funded in part by the Israel Science Foundation (grant 2005/17) and by Mellanox, which additionally graciously donated NICs. Adam Morrison is supported by Len Blavatnik and the Blavatnik Family foundation. Igor Smolyar is partially supported by the Technion Hasso Plattner Center.

References

[1] Brian Aker. Memslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html. libmemcached 1.1.0 documentation. Accessed: Jan 2018.

[2] AMD Inc. AMD IOMMU architectural specification, rev 2.00. http://developer.amd.com/wordpress/media/2012/10/488821.pdf, Mar 2011. Accessed: Jan 2018.

[3] 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: Jan 2018.

[4] ARM Holdings. ARM system memory management unit architecture specification — SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, 2013. Accessed: Jan 2018.

[5] Damien Aumaitre and Christophe 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: Jan 2018.

[6] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio. Accessed: Jan 2018.

[7] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest Applied Security Conference*, 2005. https://cansecwest.com/core05/2005-firewire-cansecwest.pdf. Accessed: Jan 2018.

[8] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference (ATC)*, pages 15–44, 2001. https://www.usenix.org/legacy/publications/library/proceedings/usenix01/full_papers/bonwick/bonwick.pdf.

[9] James E.J. Bottomley. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt. Linux kernel documentation. Accessed: Jan 2018.

[10] James E.J. Bottomley. Integrating DMA into the generic device mode. In *Ottawa Linux Symposium (OLS)*, pages 63–75, 2003. https://www.kernel.org/doc/ols/2003/ols2003-pages-63-75.pdf. Accessed: Jan 2018.

[11] Jonathan Brossard. Hardware backdooring is pratical. In *Black Hat*, 2012. http://www.toucan-system.com/research/blackhat2012_brossard_hardware_backdooring.pdf. Accessed: Jan 2018.

[12] Jonathan Corbet. Smarter shrinkers. https://lwn.net/Articles/550463/, May 2013. Accessed: Jan 2018.

[13] Jonathan Corbet. An introduction to compound pages. https://lwn.net/Articles/619514/, Nov 2014. Accessed: Jan 2018.

[14] Maximillian Dornseif. 0wned by an iPod. In *PACific SECurity – applied security conferences and training in Pacific Asia (PacSec)*, 2004. https://pacsec.jp/psj04/psj04-dornseif-e.ppt. Accessed: Jan 2018.

[15] DPDK. http://dpdk.org/. Accessed: Jan 2018.

[16] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier 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: Jan 2018.

[17] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), Aug 2004. http://www.linuxjournal.com/article/7451. Accessed: Jan 2018.

[18] Google LLC. Google infrastructure security design overview. https://cloud.google.com/security/security-design, Jan 2017. Google Cloud Whitepaper. Accessed: Jan 2018.

[19] IBM Corporation. PowerLinux servers — 64-bit DMA concepts. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm. Accessed: Jan 2018.

[20] 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: Jan 2018.

[21] Intel TXT Overview. https://www.kernel.org/doc/Documentation/intel_txt.txt. Linux kernel documentation. Accessed: Jan 2018.

[22] 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. Accessed: Jan 2018.

[23] Intel Corporation. Intel virtualization technology for directed I/O, architecture specification - architecture specification - Rev. 2.5. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Nov 2017. Accessed: Jan 2018.

[24] Joerg Roedel. AMD IOMMU DMA-API scalability improvements, Linux patch. https://lists.linuxfoundation.org/pipermail/iommu/2015-December/015245.html, Dec 2015. Accessed: Jan 2018.

[25] Intel-IOMMU.txt – Linux IOMMU support. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt. Linux kernel documentation. Accessed: Jan 2018.

[26] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan 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. http://dx.doi.org/10.1145/2872362.2872379.

[27] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient intra-operating system protection against harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 29–44, 2015. https://www.usenix.org/system/files/conference/fast15/fast15-paper-malka.pdf.

[28] Vinod Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007. Accessed: Jan 2018.

[29] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–262, 2016. https://doi.org/10.1145/2872362.2872379.

[30] Mellanox Technologies. ConnectX-5 Ex 100 Gb/s Ethernet Single and Dual QSFP28 Port Adapter Cards User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-5_Ethernet_Single_and_Dual_QSFP28_Port_Adapter_Card_User_Manual.pdf, 2018. Accessed: Jan 2018.

[31] Bosko Milekic. Network buffer allocation in the FreeBSD operating system. In *The Technical BSD Conference (BSDCan)*, 2004. https://www.bsdcan.org/2004/papers/NetworkBufferAllocation.pdf. Accessed: Jan 2018.

[32] The netfilter.org project. http://www.netfilter.org/. Accessed: Jan 2018.

[33] Netperf – a network performance benchmark. https://github.com/HewlettPackard/netperf. Accessed: Jan 2018.

[34] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU Scalably. In *USENIX Annual Technical Conference (ATC)*, pages 549–562, 2015. https://www.usenix.org/system/files/conference/atc15/atc15-paper-peleg.pdf.

[35] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[36] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of Clos topologies and centralized

control in Google's datacenter network. *Communications of the ACM (CACM)*, 59(9):88–97, Aug 2016. https://doi.org/10.1145/2975159.

[37] SPIEGEL Staff. 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: Jan 2018.

[38] Patrick Stewin and Iurii Bystrov. Understanding DMA malware. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 21–41, 2012. https://doi.org/10.1007/978-3-642-37300-8_2.

[39] Arrigo Triulzi. I 0wn the NIC, now I want a shell! In *PACific SECurity – applied security conferences and training in Pacific Asia (PacSec)*, 2008. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf. Accessed: Jan 2018.

[40] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel performance counter monitor. http://www.intel.com/software/pcm, Jan 2017. Intel Developer Zone. Accessed: Jan 2018.

[41] Mitch Williams. i40e: enable packet split only when IOMMU present, Linux commit. https://github.com/torvalds/linux/commit/2bc7ee8ac5439efec66fa20a8dc01c0a2b5af739. Accessed: Aug 2018.

[42] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy (S&P)*, pages 616–630, 2012. https://doi.org/10.1109/SP.2012.42.