Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU

Alex Markuze
Technion & VMware

Shay Vargaftik
VMware

Gil Kupfer
Technion

Boris Pismenny
Technion

Nadav Amit
VMware

Adam Morrison
Tel Aviv University

Dan Tsafrir
Technion & VMware

Abstract

Direct memory access (DMA) renders a system vulnerable to DMA attacks, in which I/O devices access memory regions not intended for their use. Hardware input–output memory management units (IOMMU) can be used to provide protection. However, an IOMMU cannot prevent all DMA attacks because it only restricts DMA at page-level granularity, leading to sub-page vulnerabilities.

Current DMA attacks rely on simple situations in which write access to a kernel pointer is obtained due to sub-page vulnerabilities and all other attack ingredients are available and reside on the same page. We show that DMA vulnerabilities are a deep-rooted issue and it is often the kernel design that enables complex and multistage DMA attacks. This work presents a structured top-down approach to characterize, exploit, and detect them.

To this end, we first categorize sub-page vulnerabilities into four types, providing insight into the structure of DMA vulnerabilities. We then identify a set of three vulnerability attributes that are sufficient to execute code injection attacks.

We built analysis tools that detect these sub-page vulnerabilities and analyze the Linux kernel. We found that 72% of the device drivers expose callback pointers, which may be overwritten by a device to hijack the kernel control flow.

Aided by our tools' output, we demonstrate novel code injection attacks on the Linux kernel; we refer to these as *compound* attacks. All previously reported attacks are *single-step*, with the vulnerability attributes present in a single page. In *compound* attacks, the vulnerability attributes are initially incomplete. However, we demonstrate that they can be obtained by carefully exploiting standard OS behavior.

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456249

ACM Reference Format:
Alex Markuze, Shay Vargaftik, Gil Kupfer, Boris Pismenny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447786.3456249

1 Introduction

Direct Memory Access (DMA) is a technology that allows input-output (I/O) devices to access memory without CPU involvement, thereby improving system performance. DMA-capable devices include internal devices, such as GPUs, Network Interface Cards (NICs), storage devices (e.g., NVMe), and other peripheral devices, including external devices such as FireWire and Thunderbolt.[1] However, in its basic form, DMA makes the system vulnerable to DMA attacks. These are cases where malicious DMA-capable devices, such as compromised firmware [7, 25], access sensitive memory regions not intended for their use.

Numerous DMA exploits are known [6, 21, 45], ranging from stealing and manipulating sensitive data to taking over the victim machine. Widespread attacks include: opening a locked computer [42, 64], executing arbitrary code on the victim machine [5, 24, 45, 67], stealing sensitive data items such as passwords [9, 13, 40, 63], and extracting a full memory dump of a victim machine [26, 42, 64, 65]. These threats are supposed to be mitigated by the Input-Output Memory Management Unit (IOMMU), which adds a layer of virtual memory to devices. The IOMMU brokers all I/O requests, translating their target I/O virtual addresses (IOVA) to physical addresses. In the process, the IOMMU provides address space isolation, allowing a device to access only permitted pages and rendering all other memory inaccessible.

Unlike processes that operate at page granularity, I/O buffers can be significantly smaller than a page. I/O buffers and other kernel buffers can co-reside on the same physical pages, inadvertently exposing these kernel buffers to the device. For this reason, known as the sub-page vulnerability [45, 47], the IOMMU cannot fully protect the kernel from unprivileged access. Consequently, sub-page vulnerabilities were the basis

[1] Currently, the Linux kernel (version 5.0) has as many as 700 such device drivers, of which one third are network device drivers.

for several recent DMA exploits [7, 8, 38, 45]. Nevertheless, these previously reported vulnerabilities have an ad-hoc nature rather than a structured top-down approach.

Accordingly, we conducted a systematic study of sub-page vulnerabilities. To provide insight into the structure of DMA vulnerabilities, we first break down sub-page vulnerabilities into four types (Section 3.2):

- Exposed driver metadata
- Exposed OS metadata
- Mapped by multiple IOVA due to multiple co-located buffers
- Randomly co-located

Next, we identify the ingredients that make it possible for a malicious device to exploit these four types of sub-page vulnerabilities and execute a viable DMA attack. Focusing on *code injection* attacks, we introduce (Section 3.3) a set of three vulnerability attributes that can be used to execute such attacks:

- A kernel virtual address (KVA) of a buffer filled with malicious executable code (i.e., malicious buffer).
- Write access to a function callback pointer, exposed in a data structure via one of the four sub-page vulnerability types.
- Existence of a time window such that the device can modify the callback pointer during that time window; the CPU will subsequently jump to the pointed code before the pointer gets overwritten, if it is ever overwritten.

With the characterization of the different sub-page vulnerabilities and the vulnerability attributes, we were able to build analysis tools that can detect potentially hazardous sub-page vulnerabilities:

- We built a static code analysis tool that performs a Sub-Page Analysis for DMA Exposure (SPADE). SPADE scans for potentially exposed callback pointers on DMA-mapped pages. We used SPADE on Linux kernel 5.0 and found that as many as 72% of device drivers are potentially vulnerable to code injection attacks (Section 4.1).
- Some sub-page vulnerabilities can only manifest dynamically at run-time, potentially exposing callback pointers and/or kernel addresses. Static analysis may not reveal vulnerabilities where a memory buffer is exposed randomly. For example, a random exposure can occur when a memory buffer is co-located on the same page as a mapped I/O buffer. Accordingly, we developed a run-time analysis tool that reports such vulnerabilities and demonstrate its use. Termed DMA-Kernel-Address-SANitizer (D-KASAN), this tool reports all cases where a kernel buffer is exposed, inadvertently or otherwise (Section 4.2).

We use our tools to find and demonstrate attacks on the Linux kernel. We focus on *compound* attacks, cases where a detected sub-page vulnerability alone is insufficient to execute a code injection attack since at least one of the three vulnerability attributes is initially missing, but can be attained via compound steps.

We observe that unlike compound attacks, previous work has explored *single-step* attacks, i.e., attacks in which the three vulnerability attributes are trivially provided. Namely, a mapped I/O buffer resides on a mapped page which, due to sub-page vulnerability, also exposes a callback pointer and a kernel virtual address, and the timing is such that the CPU will not overwrite the modifications.

Analysis of such *single-step* attacks, that can typically be blocked with localized fixes, may lead to a dangerous misconception. In particular, one may assume that buggy device drivers or poor but isolated design choices are to blame for DMA vulnerabilities [43, 44]. However, by introducing *compound* attacks, we demonstrate that it is often the kernel itself that supplements the missing pieces, showing that this is a deep-rooted issue rather than a collection of disjoint incidents. We identify multiple kernel APIs and data structure designs that facilitate the acquisition of the vulnerability attributes by a malicious device.

To summarize, we make the following contributions:

- Provide a categorization of the four sub-page vulnerability types.
- Introduce a set of three vulnerability attributes that are sufficient to execute code injection attacks.
- Develop a static code analysis tool (SPADE) to flag code paths that may expose callback pointers.
- Develop a run-time tool (D-KASAN) to identify sub-page vulnerabilities at run-time, including vulnerabilities caused by random exposure.
- Demonstrate novel DMA attacks on the Linux kernel, termed *compound* attacks.
- Make our tools publicly available [46, 48].

2 Background

In this section, we provide background on DMA-related attacks. First, we describe classic DMA attacks and the IOMMU protection against them. Then, we discuss well-established protection practices to prevent privilege escalation (i.e., code injection) attacks and methods for their circumvention.

2.1 DMA Attacks

DMA allows I/O devices direct access to memory [57] without CPU involvement. While DMA is essential for fast I/O, it also provides ample opportunity for unmonitored and malicious activity by DMA-capable devices, resulting in DMA attacks.

An attacker can access sensitive data, overwrite the OS code and data structures, and even gain full control of the

Start Addr	Offset	End Addr	Size	VM are description
ffff888000000000	-119.5 TB	ffffc87fffffffff	64 TB	direct map of phys memory (page_offset_base)
ffffc90000000000	-55 TB	ffffe8ffffffffff	32 TB	vmalloc/ioremap space (vmalloc_base)
ffffea0000000000	-22 TB	ffffeaffffffffff	1 TB	virtual memory map (vmemmap_base)
ffffec0000000000	-20 TB	fffffbffffffffff	16 TB	KASAN shadow memory
ffffffff80000000	-2 GB	ffffffff9fffffff	512 MB	kernel text mapping (physical address 0)
ffffffffa0000000	-1536 MB	fffffffffeffffff	1520 MB	module mapping space

Table 1. Linux kernel memory layout

victim system. DMA attacks can be carried out using an external or internal DMA-capable device.

Accessible expansion ports, e.g., FireWire or Thunderbolt, allow external devices to initiate DMA transactions merely by connecting a programmable accessory [21, 42, 45, 65]. Exploiting internal devices is more challenging, but enables persistent and stealthy attacks.

Many options are available to gain control of an internal device. For example, a resourceful attacker can exploit firmware bugs [63]. These can be well-known exploits, since end-users are often slow in deploying firmware updates [22]; they may even be newly discovered zero-day vulnerabilities [8]. Alternatively, certain attackers may be able to replace the device firmware altogether with a malicious one [55, 71]. It is also possible to manufacture devices that appear to be legitimate but are, in fact, malicious at the circuitry level [69].

Once an attacker gains control over a DMA device connected to a victim machine, various attacks are possible. These attacks can range from keyloggers [40, 63] to full control over commodity OS and hypervisor, including Windows [5, 45], Linux, OSX [24, 45], Android [8], and Xen [67].

Several software tools exist for perpetrating DMA attacks, with some of them being open source. Tools such as Volatility [65], Inception [42], GoldFish [26], and FinFireWire [64] can extract target machine memory and unlock victim machines by patching the OS code. These tools are reportedly used by government agencies, such as the NSA.

2.2 IOMMU

With the lack of software protection against DMA attacks, the common practice is to restrict DMA accesses through hardware protection. The most common mechanism for this purpose is the I/O memory management unit (IOMMU). The IOMMU adds a level of indirection for DMA addresses [54, 63, 66, 70], effectively providing peripheral devices with I/O virtual addresses (IOVA). This way, the device can access only those pages explicitly allowed by the OS. Inspired by the x86 MMU, the IOMMU uses a page table for address translation and an IOTLB for caching recent accesses. The page tables are managed by the OS, and as with the MMU, have a page granularity. The common page size is 4 KB, although there exist larger page sizes, up to GBs.

The IOMMU page table also holds page access rights for each IOVA. An access right can be either READ, WRITE, or BIDIRECTIONAL. Note that WRITE access does not grant a DMA device READ access, whereas BIDIRECTIONAL access is needed to both read and write from/to the page. It is also important to note that a single physical page can be mapped by multiple IOVAs, each with possibly different access rights.

IOMMUs were not designed primarily to provide security [19]. Instead, IOMMUs were used to allow devices that did not support vectored I/O, to access contiguous virtual memory that may map non-contiguous physical memory [12, 68]. IOMMUs also enabled legacy devices that only supported a limited address width (32-bit) to access high memory (64-bit). More recently, IOMMUs were used to assign I/O devices directly to virtual machines, while maintaining their isolation properties [1, 32].

2.3 DMA API

Device drivers must use the DMA API to manage the 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 an IOVA. The driver must configure the device to DMA for that specific IOVA; `dma_unmap` later takes this IOVA as its parameter. There are analogous methods to map and unmap for non-contiguous scatter/gather lists.

2.4 OS Defenses

Other than DMA attacks, OS developers have to worry about code injection originating from unprivileged users, such as buffer overflow attacks [23, 35]. We discuss the common mechanisms used to mitigate such attacks in the kernel. Subverting these countermeasures is essential to executing a successful DMA attack.

NX-BIT. A malicious device can gain write access to a function pointer and consequently gain the ability to inject malicious code. To protect against such threats, modern OSs make use of hardware support, namely the No-eXecute bit, to prevent code execution from data pages. This bit is defined for each page in the MMU's page tables. Whenever the CPU tries to fetch code from memory, this bit is validated. When set, the

CPU raises an exception to the OS instead of executing the code. The NX-bit method is also known as the W ⊕ X (Write ⊕ eXecute) or DEP (Data Execution Prevention). A DMA-capable device rarely has access to kernel text regions. Thus, the NX-bit is effective in preventing simple code injection attacks.

Subverting NX-BIT. Return Oriented Programming (ROP) is a common method used by malware to bypass DEP (i.e., NX-bit) defenses [60]. ROP exploits the fact that the CPU stack pointer may point to any data page. To set up an attack from a data page, the attacker builds a poisoned stack filled with required data and pointers to specific locations in the code section (a.k.a., ROP gadgets). Each gadget is a short piece of code, usually one or two instructions, and a return instruction. When the CPU executes a return instruction, the return address and thus the address of the next instruction to execute is taken from the stack. In the poisoned stack, each return address points to the next gadget, and so on.

By carefully selecting these gadgets, an attacker can execute any payload. To bootstrap a ROP attack, an attacker must modify the stack pointer register to the address of the poisoned stack. This is often achieved with another DEP circumventing technique, known as Jump Oriented Programming (JOP) [10]. JOP uses, `jump` instructions instead of `return` instructions and, thus, does not require a poisoned stack.

KASLR. Address Space Layout Randomization (ASLR) is a common mechanism for mitigating code injection attacks in the context of user-level processes. Systems that support ASLR, randomize the memory layout for each process on every execution. This way, ROP attacks built for a specific layout fail. Similar to ASLR, KASLR [23] randomizes the kernel memory layout.

Specifically, the Linux kernel has predetermined ranges for its virtual memory layout [36]. This layout defines the location of the kernel text mapping, and the direct mapping of physical memory and the virtual memory, as depicted in Table 1. At each boot, KASLR randomizes the offset of these segments in the corresponding range.

Subverting KASLR. To successfully execute a code injection attack, the attacker must know the memory layout. Specifically, the address of the code section is required for finding ROP gadgets (Section 2.4). Since KASLR randomizes only the base address of the kernel text mapping, text addresses always appear in the *kernel text mapping* range (Figure 1) and are therefore easy to detect. KASLR kernel text is aligned to 2 MB borders. This is the result of page table restrictions and is unlikely to change. This means that the lowest 21 bits are not modified by the KASLR randomization procedure. Hence, knowing even a single address of a known element is sufficient to deduce the base address and compromise KASLR. Once the base address is known, the attacker can use it to create a ROP stack.

To identify this first pointer, malicious devices can scan the pages mapped for reading, looking for kernel pointers leaked due to sub-page vulnerability. Once such a pointer is identified, all that remains is to infer the offset of the symbol in the binary from the pointer to get the base address.

In fact, during our investigation, we found that there is a symbol visible to both FireWire and NICs in Linux 5.0, that compromises KASLR. Specifically, as of version 2.6.24, Linux supports network namespaces and every network object, especially sockets, have a pointer to their namespace object. One such global namespace object, `init_net`, is always defined. By scanning leaked pages during I/O and utilizing the fact that the lower 21 bits of the text region are never modified, we can identify `init_net` with a high probability. The direct mapping base (`page_offset_base`) and virtual memory map (`vmemmap_base`) are also randomized (Figure 1), where each region's base pointer is randomized with respect to a 1 GB alignment. This means the lower 30 bits are unmodified and can leak both the Page Frame Number (PFN) and the randomized offset. This alignment is also due to page table considerations. That is, the page upper table (PUD) has a 30-bit shift. Once the random offsets *PAGE_OFFSET* and *vmemmap* for direct mapping base and virtual memory map are known, it becomes possible to translate between a KVA (kernel virtual addresses within the direct mapping region), its PFN, and its `struct page` address (virtual memory map region).

3 Categorizing DMA Risks

This section organizes and categorizes the risks associated with DMA operations, providing the building blocks for reasoning about code injection attacks. We first present our threat model and then organize the different sub-page vulnerabilities into four categories. Finally, we identify a set of three vulnerability attributes that make it possible for a malicious device to exploit a sub-page vulnerability and execute a viable code injection attack.

3.1 Threat Model

By organizing the DMA risks, we reveal that the Linux kernel is vulnerable to various high impact attacks. For example, a full memory dump is possible when an attacker can modify data pointers before they are mapped, causing the driver to map arbitrary kernel addresses. Alternatively, a malicious device can corrupt random memory regions [47], resulting in a denial of service attack (DOS).

The most significant potential consequence of these attacks is privilege escalation via code injection, allowing attackers to execute arbitrary code with kernel privileges. Indeed, this is the focus of our paper. Our attacks are designed with the following assumptions:

1. A malicious DMA-capable device is attached to the system.

2. The actual attack is performed solely by the DMA-capable malicious device.
3. Any hardware aside from the specific malicious device is working as expected.

3.2 Sub-Page Vulnerabilities

Anytime an I/O buffer smaller than a page is DMA-mapped, all additional information that resides on the same physical page becomes accessible to the device. Any such situation where a memory-region is exposed due to the IOMMU page granularity is called a sub-page vulnerability.

We classify the different types of sub-page vulnerabilities into four categories, as illustrated in Figure 1:

(a) The I/O buffer is part of a bigger data structure. This data structure may include function pointers, often caused by poor DMA hygiene in the driver. An isolated driver fix is usually sufficient to repair such vulnerabilities.

(b) The OS (e.g., memory allocator)—rather than the driver—saves metadata such as free-lists, on the same page as the I/O buffer [15]. Manipulating these data structures may also compromise the system [4]. Similar to (a), sensitive metadata is unwittingly shared. However, in this case, it is an OS subsystem that is at fault rather than the device driver.

(c) The same page is mapped multiple times due to co-located device driver buffers, resulting in multiple IOVAs indicating the same page. In this case, unmapping one IOVA will not prevent the device from accessing the page through a different IOVA. The device will retain access to the physical page as long as a single valid IOVA exists. This means that the device can tamper with memory regions that should no longer be accessible. We discuss the practical implications of multiple IOVAs indicating the same page in Section 5.2.

(d) The I/O buffer and a different, dynamically allocated memory buffer may coincidentally share a page. This common situation results in data leakage (e.g., kernel pointers). Currently, the Linux kernel uses the same memory allocation mechanism (e.g., kmalloc) for both I/O buffers and regular kernel buffers. Consequently, I/O buffers often share pages with other, potentially sensitive, kernel buffers. Since IOMMU works at page granularity, the respective I/O devices gain access to these kernel buffers. Such vulnerability is a subclass of (b), as it is caused by an OS subsystem; the main difference is that the exposed data structures are leaked randomly.

3.3 Vulnerability Attributes for Code Injection

We introduce a schema that allows for a systematic analysis of code injection attacks by DMA-capable devices. For a successful privilege escalation attack (i.e., code injection), a malicious device needs the following set of three vulnerability attributes:

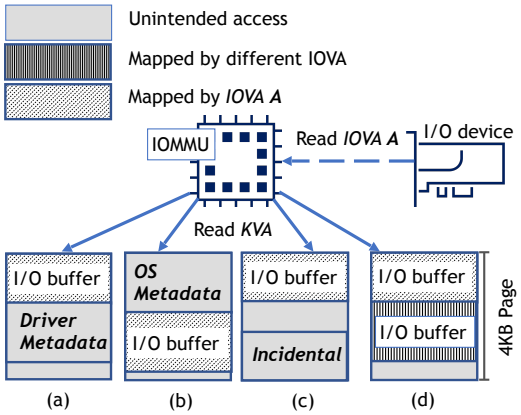

Figure 1. DMA sub-page vulnerabilities when the I/O buffer shares a page with other data: (a) I/O buffer metadata; (b) OS metadata; (c) a page mapped by multiple IOVA; (d) randomly co-located sensitive buffers.

1. The KVA of a kernel buffer filled with malicious code (e.g., ROP attack). Given that the device is using an IOVA, the attacker needs to obtain the buffer's KVA, for example, by observing leaked pointers.
2. Write access to a function callback pointer, which can alter the CPU control flow and cause it to execute the malicious code. For example, this might be write access to a data structure that holds a function callback pointer at a known offset.[2] The location on the page of the callback pointer must be known to the device.
3. A time window exists such that the device can modify the callback pointer during that window and the CPU will subsequently jump to the pointed code; this occurs before the pointer gets overwritten, if it is ever overwritten.

To further emphasize the significance of these attributes, we present a hypothetical scenario. Assume a NIC has write access to a page containing a received (RX) packet. Due to a sub-page vulnerability and a random allocation coincidence (Figure 1, type (d)), a structure with a callback pointer is write-accessible to a malicious device. Also, the device can create a valid malicious buffer in the aforementioned page. It may seem that the device has a valid attack, whereas it actually lacks the following:

- Without a valid KVA of the writable page, the device cannot modify the callback function pointer to indicate the malicious buffer.
- Although a callback function pointer is available for modifications, the device has no way of knowing that a

[2]The Linux kernel randomizes the layout of some data structures with __randomize_layout annotation [56].

callback function pointer is available for sabotage nor the correct offset of the callback function pointer.
- It is not known whether the modified callback pointer will be executed.

Under the hypothesized circumstance, and without additional information, a malicious device has no viable attack options. All three attributes are required for a code injection attack. While corrupting the random kernel memory is still a possibility and may even cause a kernel panic [47], it does not achieve the goal of privilege escalation.

4 Detecting Sub-Page Vulnerabilities

We now present the tools we developed to identify the sub-page vulnerabilities described in the previous section.

4.1 Sub-Page Analysis for DMA Exposure

We devised a static code analysis tool that performs Sub-Page Analysis for DMA Exposure (SPADE). With well over 1000 `dma_map*` function calls (i.e., the set of functions implementing the DMA API) in the Linux kernel, a manual process would be arduous. SPADE performs the following operations to detect the different sub-page vulnerability types (Figure 1) where a callback pointer may be exposed:

1. Type A: Looks for `dma_map*` functions and traces back the call stack to identify whether the mapped buffer is embedded inside a data structure.
2. Type B: Looks for kernel APIs that create a data structure inside a mapped buffer (e.g., `build_skb`).
3. Type C: Looks for functions that are used for fast allocation by slicing a contiguous memory buffer into segments (e.g., netdev_alloc_skb, napi_alloc_skb). These may result in multiple IOVA mapping the same page. These functions utilize the `page_frag` API, which we discuss in greater detail in Section 5.1.

4.1.1 High-Level Design Overview. SPADE operates recursively starting from calls to the dma_map* functions. From this initial set of calls, SPADE identifies the mapped variables and backtracks their declarations and assignments. When a data structure is identified as exposed, SPADE identifies the exposed callback pointers or mapped heap pointers.

SPADE is implemented in approximately 2000 lines of Perl 5 code. It uses `pahole` [17] to explore the compiled binaries for the layout of the exposed data structures. Pahole is a tool that uses the DWARF [28] standardized debugging data format to examine data structure layout. To navigate the kernel code, SPADE uses Cscope [37, 39] which is an open source tool for browsing C source code.

SPADE is applicable to any kernel code written in C. We intend to make the SPADE publicly available for the benefit of the research community.

4.1.2 Output. For each DMA-mapping call, SPADE outputs the line numbers of relevant declarations, function calls, and assignments, allowing human experts to trace back and validate the vulnerability. Figure 2 presents an example of output for a vulnerability found in the NVMe host driver.

The output starts from the impact evaluation, such as detected exposed callback pointers, and continues with pertinent code lines. Line 7 reveals that a single callback pointer is mapped in the mapped `nvme_fc_fcp_op` data structure (i.e., `fcp_req.done`) and line 8 reveals that it is possible to spoof another 931 callback pointers.[3] After finding the variable declaration in line 5, and looking at the mapped pointer `&op->rsp_iu` in line 4, SPADE concludes in line 6 that the entire `struct nvme_fc_fcp_op` is exposed to the device. Lines 1 through 3 repeat the same analysis process for the `dma_map_single` call that exposes the data structure to the device.

This example demonstrates the recursive nature of the analysis. SPADE first identifies the suspect function call, finds the mapped pointer's declaration, and then prints its type. This pattern repeats itself in lines 1 through 3 and 4 through 6 until a vulnerability is discovered. The findings are then displayed: line 7 counts the number of directly exposed callback pointers, and line 8 displays the number of callback pointers that may be potentially spoofed.

4.1.3 Analysis and Results. We used SPADE over Linux kernel 5.0 code, analyzing 1019 dma_map_single calls over 447 files. We present the results in Table 2. We found 156 cases in which device drivers expose callback pointers. Of these, 54 are cases in which the pointers are exposed directly, and the rest are cases in which callback pointers can be spoofed. We found that 13% (line 1 in Fig. 2) of the drivers expose data structures via type (a) vulnerabilities, whereas 60% (lines 2,7 in Fig. 2) expose data structures via type (b) vulnerabilities. Namely, 13% are vulnerable due to driver bugs and 60% of drivers are vulnerable due to OS design choices. In the case of the Linux kernel, the most common source of vulnerability caused by the OS design is struct `skb_shared_info`, which is used ubiquitously in Linux networking. This data structure is *always* located on the same page as the `skb->data`, and it also contains a callback pointer. We discuss the vulnerabilities related to `skb_shared_info` in Section 5. We found that more than 50% of the dma-map calls either directly map the `skb->data` or use the `build_skb` API (lines 2 and 7 in Table 2), which exposes `skb_shared_info`. The OS provides this data structure layout and API rather than it being an isolated driver bug. Additionally, we found 19 data structures that are exposed via APIs that store `private` data structures on the same page as vulnerable metadata, e.g., netdev_priv, aead_request_ctx, and scsi_cmd_priv.

In addition to type (a) and (b) vulnerabilities, SPADE also flagged 344 cases where a type (c) vulnerability is present.

[3] In this case, spoofing means replacing this pointer to indicate an instance of the structure created by the device, with its own callback pointers.

```
[8]/*** Spoofed Vulnerability:*/ 931 Callbacks reachable via struct nvme_fc_fcp_op : DMA_FROM_DEVICE
[7]/*** Direct Vulnerability: */ 1   Callback exposed in    struct nvme_fc_fcp_op : DMA_FROM_DEVICE
[6]/*mapped type:*/ struct nvme_fc_fcp_op
[5]/*DECLARATION*/["__nvme_fc_init_request:1698"]:__nvme_fc_init_request(struct nvme_fc_ctrl *ctrl,
                                                    struct nvme_fc_queue *queue, struct nvme_fc_fcp_op *op, ...)
[4]/*CALL*/["__nvme_fc_init_request:1731"]: fc_dma_map_single(ctrl->lport->dev, &op->rsp_iu,
                                                  sizeof(op->rsp_iu), DMA_FROM_DEVICE);
[3]/*mapped type:*/ void
[2]/*DECLARATION*/["fc_dma_map_single:935"]:fc_dma_map_single(struct device *dev, void *ptr, ...) {
[1]/*CALL*/["fc_dma_map_single:939"]: return dev ? dma_map_single(dev, ptr, size, dir) : (dma_addr_t)0L;
```

Figure 2. SPADE output example showing a path in the nvme_fc driver where a callback pointer is exposed with write access.

Stat	#API calls	#Files
1. Callbacks exposed	156 (15.3%)	57 (12.8%)
2. `skb_shared_info` mapped	464 (45.5%)	232 (51.9%)
3. Callbacks exposed directly	54	28
4. Private data mapped	19	7
5. Stack mapped	3	3
6. Type C vulnerability	344	227
7. `build_skb` used	46	40
Total dma-map calls	1019	447

Table 2. SPADE results summary

Our analysis found three instances where the stack pointer is mapped, potentially simplifying the execution of a ROP attack.

In total, we found 742 dma-map calls (i.e., 72.8% of all dma-map calls) with a potential vulnerability, of which 344 also admit a type (c) vulnerability.

Our code for SPADE is publicly available [46].

4.2 DMA Kernel Address SANitizer

In Section 4.1, we demonstrated that more than 70% of DMA-map operations result in exposed pointers. Most of the remaining 30% of DMA-map operations are executed on allocated objects that are presumably not co-located on the same page with vulnerable metadata. However, this is often not the case in practice. Indeed, objects allocated via the kmalloc API [15] may share a page with objects of similar size. As a result, vulnerable metadata may still be mapped. Such a vulnerability is not visible to SPADE as it is of a dynamic nature. Accordingly, we developed a run-time tool that reports such vulnerabilities.

Our solution is based on an existing kernel tool, KASAN [20], which is a dynamic memory error detector designed to detect out-of-bounds and use-after-free bugs. KASAN uses shadow memory to record whether a memory byte is safe to access. It also uses compile-time instrumentation to insert checks of shadow memory on each memory access. We modified KASAN to record DMA-map operations in addition to memory allocations. Our tool, referred to as DMA-KASAN (D-KASAN), reports the following:

1. alloc-after-map: kmalloc object is allocated from a mapped page.
2. map-after-alloc: the containing page is mapped after an object was allocated.

```
[1] size 512 [READ,WRITE] __alloc_skb+0xe0/0x3f0
[2] size 512 [WRITE] load_elf_phdrs+0xbf/0x130
[3] size 512 [WRITE] __do_execve_file.isra.0+0x287/0x1080
[4] size 64  [WRITE] sock_alloc_inode+0x4f/0x120
[5] size 328 [READ,WRITE] assoc_array_insert+0xa9/0x7e0
```

Figure 3. D-KASAN report example

3. access-after-map: the CPU accesses a DMA mapped page.
4. multiple-map: an object is mapped multiple times with possibly different permissions.

We tested D-KASAN using the setup described in Section 6). In our experiment we cloned a large project from a Git repository and compiled it concurrently with light network traffic (i.e., ICMP ping). This experiment identified numerous cases where a DMA-mapped page is used to hold network and file system metadata. Sample results are shown in Figure 3. Each line shows an allocation that results in a random exposure; namely, it shows the size of the allocated buffer, the DMA access type, and the allocating location (i.e., function name and offset).

Figure 3 also shows how random kernel data structures can be mapped for both READ and WRITE. Some of these, such as line 5, also contain callback pointers.

While we do not demonstrate random exposure exploits, these findings indicate that random exposure vulnerabilities should not be disregarded. Accordingly, in Section 5.5, we present a compound attack that exploits an I/O buffer that is mapped twice, once for read and once for write. Line 1 in Figure 3 shows how such double mapping innocently occurs.

4.3 Discussion and Limitations

D-KASAN is a run-time tool that has a large memory footprint and the obvious overhead of callbacks on each memory access. This tool is useful for testing specific systems for vulnerabilities. SPADE is a static analysis tool that may fail to follow a mapped variable due to complex code constructs such as function pointers, macros, and others, potentially resulting in a false-negative result. False positives may happen in the rare situation where the mapped data structure crosses a page boundary. In this case, SPADE may flag a callback

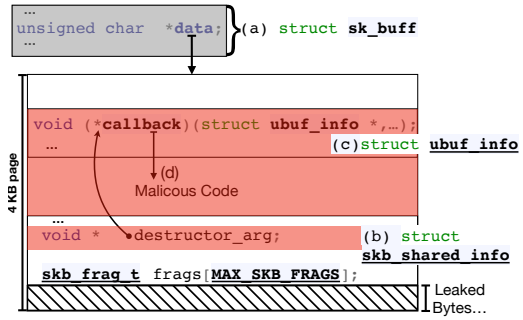

Figure 4. Using `skb_shared_info` to execute arbitrary code in a kernel context.

function that may not be exposed, since it resides on a different page. Only part of a data structure is accessible to the device due to the sub-page vulnerability at the mapped page, whereas the callback pointer resides on a different page that is not accessible to the device.

Our code for D-KASAN is publicly available [48].

5 Compound DMA Attacks

This section explores new attacks on the Linux network stack, where the vulnerability attributes are initially missing but are attainable via compound steps. We focus on the Linux network stack, which initially appears secure [45]. Nevertheless, as we demonstrate, the Linux network stack is actually responsible for 60% of the DMA vulnerabilities we found.

Recall that, once we have discovered a sub-page vulnerability, our goal is to obtain the three vulnerability attributes described in Section 3.3: (1) a KVA, (2) a callback pointer and, (3) timing. Accordingly, in Section 5.1, we first describe how to obtain (2) a callback function pointer. Then, in Section 5.2, we show that (3) a time window for exploiting this pointer is available. While these two steps for obtaining vulnerability attributes (2) and (3) are generic, there are different recipes for how to obtain the remaining vulnerability attribute (1), i.e., the KVA of the malicious buffer. We complete the vulnerability attributes in Section 5.3, Section 5.4, and Section 5.5, by showing different ways to obtain (1) the KVA.

5.1 Obtaining a Callback Pointer

Struct `sk_buff` is a data structure used by the Linux network stack to hold information representing a network packet. Struct `sk_buff` holds the metadata of a network packet (e.g., packet size, associated socket). One of these fields is a pointer to a data buffer. The data is allocated separately, and thus, does not share a page with its `sk_buff`, as shown in Figure 4.

This separation means that `sk_buff` is *never* intentionally mapped to the device. Indeed, it is a common belief (e.g., Markettos et al. [45]) that the Linux network stack is not susceptible to DMA attacks via the `data` pointer. In this work, we show that this belief is misplaced.

The Linux network stack supports packet cloning by merely copying `sk_buff` metadata. That is, the resulting `sk_buff` and the original one share the `data` buffer [16]. The payload in the `sk_buff` can be partially located on the *linear* part (i.e., in the buffer indicated by the `data` pointer) and partially on the *non-linear* fragments; that is, buffers that are described by their `struct page`, length, and offset in the `frags` array of `skb_shared_info` (Figure 4).

To support the sharing of these non-linear buffers, the embedded `skb_shared_info` metadata structure is used. Struct `skb_shared_info`, in contrast to `sk_buff`, is *always* allocated as part of the data buffer. Therefore it is *always* mapped to the device. `skb_shared_info` is unwittingly mapped with the permissions of the packet, i.e., WRITE for RX packets, READ for TX packets, and in some cases, such as XDP [53] with BIDIRECTIONAL.

Consequentially, `skb_shared_info` holds the potential callback pointer that the malicious device can exploit.[4] The sub-page vulnerability created by `skb_shared_info` represents a type (b) vulnerability (Figure 1 (b)): this is innate to Linux networking, as opposed to a driver security bug.

Figure 4 depicts how a malicious device can mount an attack using `skb_shared_info` in four steps:

(a) An RX `sk_buff` and its data buffer are allocated. The data buffer is mapped for the NIC with WRITE access to the whole 4 KB page.
(b) The NIC overwrites the `destructor_arg` field in `skb_shared_info` to point within the mapped page. As a result, the `destructor_arg` points to a struct `ubuf_info` that is created by the NIC.
(c) `ubuf_info` has a callback pointer that is now pointing to the malicious code residing on the same page. In the case of NX-bit, it is a poisoned ROP/JOP[10] stack (Section 2.4).
(d) When the `sk_buff` is released, the callback is invoked.

To expand this scenario into a complete attack, the attacker must obtain all three vulnerability attributes. Namely, the attacker needs the actual KVA of the malicious buffer and the NIC must have a timely window for WRITE access to the page. Next, we demonstrate how an attacker can leverage standard OS behavior to obtain both missing vulnerability attributes.

5.2 Existence of a Time Window

To reason about the existence of an appropriate time window for altering the callback pointer, we first discuss the Linux default mode for IOTLB invalidation, which is a known security vulnerability [47, 49]. In Section 5.2.1, we present the issue of

[4] In Fig. 4 the `destructor_arg`, which holds a callback pointer, is used for socket buffer accounting and facilitates custom handling when the buffer is freed.

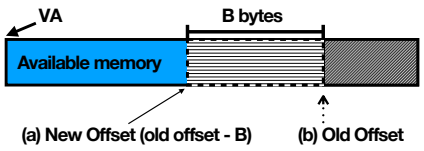

Figure 5. Allocation of B bytes from page_frag

deferred invalidation. Then, in Section 5.2.2, we discuss the multiple means by which an attacker can gain timely access to `skb_shared_info`.

5.2.1 Deferred Invalidation Vulnerability. The IOTLB is analogous to the MMU TLB. The IOMMU does not maintain consistency between the IOTLB and the IOMMU page tables. As a result, the OS has to explicitly invalidate the IOTLB to maintain consistency when a translation entry is removed. To ensure that the IOTLB never holds stale entries, the OS must invalidate the IOTLB entry immediately after removing a DMA mapping.

This scheme, called *strict* mode in Linux, can degrade performance due to the overhead of IOTLB invalidations following each I/O operation [47, 49, 59]. In I/O intensive workloads, the combined cost of IOTLB invalidations can be prohibitively high. The overhead of each IOTLB invalidation can be as high as 2000 cycles [2]. This overhead is considerably higher than a TLB invalidation, which takes roughly 100 cycles [29].

To reduce this overhead, Linux uses deferred mode as a default. Linux defers specific IOTLB invalidations and instead performs periodic global IOTLB invalidations. While this *deferred* mode improves I/O performance, it also breaks the guarantee that after unmapping (e.g., `dma_unmap_page`), the physical page should no longer be accessible by the device. This *deferred* time frame, shown in Figure 6), may be as high as 10 milliseconds [49].

The repercussions of *deferred* mode are that a malicious device can take advantage of this time window, where it has access to memory pages unbeknownst to the CPU. The *deferred* mode opens up two distinct attack options:

1. A device can alter data structures that the CPU has modified *after* unmapping (e.g., calling `dma_unmap_page`). IOVA mappings, as a rule, are short-lived as they are meant be used only for the duration of the I/O, usually for a few microseconds. The additional milliseconds provide the attacker with a time window wide enough to conduct its attack.
2. The page can be freed and then immediately reused by the OS. Fast reuse is a common scenario since Linux reuses *hot* pages (i.e., recently used pages) as they are likely to reside in the CPU caches [14]. However, this also leaves the kernel open to additional random exposure attacks.

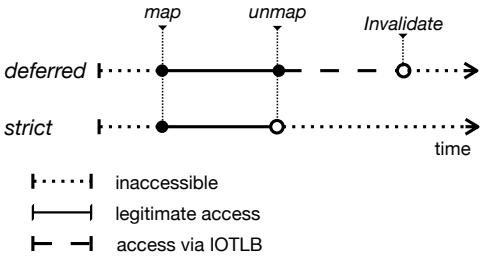

Figure 6. Strict vs. deferred IOTLB invalidation. In *deferred* mode, there is a time window in which the data is accessible to the device, but the mapping no longer appears in the page table.

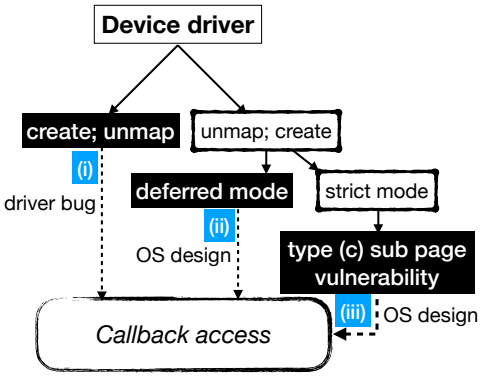

Figure 7. Different ways in which the callback pointer in `skb_shared_info` can be successfully exploited.

5.2.2 Time Window. When a packet arrives on a receive path, an `skb_shared_info` struct is initialized after the packet is received i.e., after the DMA operation was completed and the DMA access is potentially revoked. In such a case, correct use of the DMA API should thwart the attack outlined in Section 5.1 (Figure 4). First unmapping the buffer and only then initializing the `skb_shared_info` should allow the CPU to undo any malicious changes made by the NIC. But, as we next demonstrate, DMA access is often easily achieved even after the CPU has made its changes.

We now describe how the time window is attainable via three different paths, as illustrated in Figure 7:

(i) Apparently, prevalent device drivers (e.g., Intel 40GbE driver, i40e) first create an `sk_buff` and only then unmap the buffer. This order of execution allows the *device* to undo legitimate changes to `skb_shared_info` by the CPU.
(ii) Even when the order is correct and the unmapping of the buffer occurs before the creation of the `sk_buff`, `skb_shared_info` is still not secure from later modifications. Because the default IOMMU mode in Linux is *deferred* protection (Section 5.2.1), the unmap order is made irrelevant. Even though the unmap function is invoked in the correct order, the device can still corrupt `skb_shared_info` due to the IOTLB.

(iii) In response, a security-conscious admin may change the default setting to *strict* mode, where the IOTLB is flushed at every unmap. However, this severely degrades networking performance [47, 49] and does not alleviate the security threats on the system. Presumably, with *strict* mode enabled, the IOVA that is used by the NIC to access that `skb_shared_info` is no longer valid. This initially sounds promising. The problem is that in many cases the device still has legitimate WRITE access to the physical page of the `skb_shared_info`. The vulnerability stems from the way *data* is allocated. An RX `sk_buff` is almost exclusively allocated via an API (e.g., `netdev_alloc_skb`) that creates a type (c) sub-page vulnerability (Figure 1(c)). The device can use the IOVA of a co-located buffer to access the `skb_shared_info` it requires. Specifically, the buffers of the driver RX ring are allocated sequentially, resulting in pairs of successive RX descriptors that map the same page. Obviously, this holds as long as the buffer sizes are smaller than 4 KB. This is a reasonable assumption since the default MTU size is 1500 B. These allocation functions, use a `page_frag` mechanism to allocate the *data* buffers, which in turn contain `skb_shared_info`. The `page_frag` is an efficient method for allocating small buffers, and is often used by the Linux network stack. In fact, it is used 344 times by network drivers in Linux kernel 5.0. A `page_frag` is initialized by allocating a contiguous memory region (usually 32 KB), setting a *va* pointer to the beginning of the region, and setting an `offset` to the end. An allocation request for `B` bytes subtracts `B` bytes from the `offset` pointer and returns the new value of the `offset`. In multi-core environments, the `page_frag` uses a different buffer for each CPU and each CPU has a single RX ring. As a result, each RX ring is served by its own (per-CPU) contiguous buffer, as show in Figure 5). This mechanism for memory allocation results in consecutive *data* buffers often residing on the same memory page. Due to this type (c) sub-page vulnerability, the NIC does not require the invalidated IOVA to modify the `skb_shared_info`. Instead, it can use the IOVA for the next data buffer.[5] The device still has write access due to the valid IOVA of the next buffer (i.e., the striped area at the end of the page in Figure 4).

From this point on, we assume that the attacker can always modify the callback pointer. In the next subsections, we demonstrate various compound DMA attacks in which an attacker can exploit the OS design to obtain the kernel virtual address of buffers containing malicious code; this completes the trifecta of vulnerabilities.

5.3 RingFlood *Compound* Attack

A malicious device can generate a poisoned ROP stack in each RX buffer. However, this is not sufficient to execute a successful code injection attack since the device has all the IOVA for the RX buffers, but not the KVA.

In this attack, we take advantage of the fact that the boot process is *deterministic*. At every reboot, the same set of commands is executed in the same order, initiating the same kernel modules and starting the same processes. While the pages each module receives may vary in a multi-core environment due to timing issues, we do not expect the drift to be too large.

We evaluated this assumption on our setup, running 256 reboots on Ubuntu 18.04 with Linux kernels 5.0 and 4.15. With the mlx5_core driver, many PFNs repeat in more than 50% of reboots on kernel 5.0 and more than 95% on kernel 4.15. The 4.15 driver version allocates much more memory, allocating 64 KB per RX buffer to facilitate the HW LRO feature. We assume an attacker can gain access to an identical setup and identify the most common PFN. Therefore, an adversary with knowledge about the physical setup can deduce a valid KVA for one of the RX pages containing a malicious buffer. This provides the needed KVA. Thus, the device can execute the attack as shown in Figure 4.

The chances of success for the RingFlood attack increase with the memory footprint of the device driver. The memory footprint, in turn, depends on the NIC capabilities and the number of cores (number of RX rings) on the server. This means such attacks have a higher chance of success on larger machines. For example, some NICs have a HW LRO capability[50], where a NIC can aggregate multiple TCP packets into a single TCP packet that is larger than the MTU (e.g., bnx2x, mlx5_core). On drivers configured with these options, each RX buffer is 64 KB, regardless of the MTU. As a result, these drivers have a much larger memory footprint. The Mellanox mlx5_core driver on kernel 4.15 enables HW LRO and, as a result, allocates 2 GB of memory per physical device port on a 32-core machine. On kernel 5.0, HW LRO is disabled, and the driver allocates 2 KB per entry, thus only using 64 MB per port.

5.4 Poisoned TX *Compound* Attack

The RingFlood attack, described in Section 5.3, allows a NIC to execute arbitrary code, provided it has enough information regarding the server's physical layout and a sufficiently high driver memory footprint. When deducing a valid PFN is not an option (e.g., due to a low memory footprint), another way of acquiring a valid KVA is needed.

In this next attack, the KVA is acquired by spoofing a malicious transmitted (TX) packet. The attacker gains the needed KVA by *reading* it from the `skb_shared_info` of the sent packet. There are multiple ways in which a malicious NIC device can initiate a TX flow on the server. We list a few examples below:

[5] Note that the lower 12 bits (i.e., the offset on the page) of the IOVA are identical to the corresponding KVA bits.

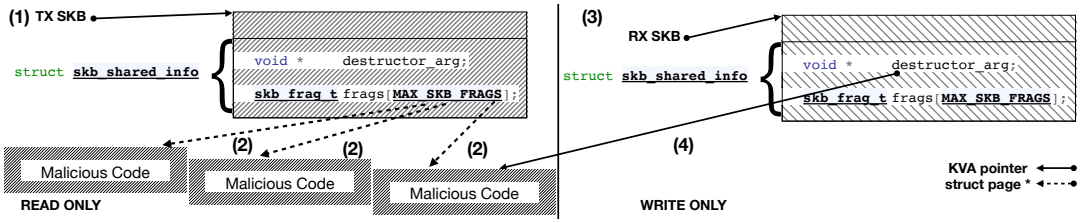

Figure 8. A TX sk_buff filled with malicious code provides the KVA for a DMA attack.

1. A userspace process can be coerced into echoing a malicious buffer's contents in various ways, including a proxy server, a key/value store, and a streaming service.
2. A cloud VM (e.g., on GCP, AWS, or Azure) or publicly accessible VM may be used to compromise the *host* in the presence of a malicious device.[6]
3. Packet forwarding is enabled on the server.

Since a NIC has READ access to the `skb_shared_info` of a TX packet, this also provides the NIC with READ access to the `frags` array of `skb_shared_info`, as shown in Figure 8. This array contains `struct page` pointers and thus, leaks kernel pointers that allow the attacker to compromise KASLR in addition to providing the PFNs of specific pages containing the data (i.e., pages the device can read).

Once the content of the malicious buffer is echoed via one of the methods mentioned above, the device can execute a code injection attack in four steps:

1. The TX *data* and the fragments are mapped for the NIC to read.
2. The NIC spoofs an RX packet and delays the completion notification of the TX packets so the malicious buffer is not released prematurely.
3. The NIC identifies the poisoned buffer and translates `struct page` to KVA (Section 2.4).
4. The NIC overwrites `skb_shared_info` with the KVA retrieved during step 3.

In this scenario, the attacker does not require prior knowledge regarding the physical setup since the echoed buffer provides the KVA.

Note that an attacker will need to delay the TX completion of the echoed buffer to ensure the contents are unchanged until the ROP/JOP attack is executed. Moreover, a TX completion event that fails to appear in due time will trigger a TX T/O error that flushes all buffers and resets the driver. The T/O is set by the driver, usually to a few seconds, which is sufficient to complete the attack.

5.5 Forward Thinking *Compound* Attack

Packet forwarding is a standard Linux feature that allows a Linux machine to serve as a router or a load balancer. Packet forwarding functionality is usually disabled by default on Linux servers.

When this functionality is enabled, the NIC can independently generate an RX packet to a legitimate destination. This packet will then be forwarded to become a TX packet. However, unlike the TCP layer, which usually creates `sk_buff` packets with fragments, device drivers often create a linear `sk_buff`. Namely, the drivers do not fill the `frags`, which the attacker uses to obtain a KVA. Device drivers, use the `napi_gro_receive` function to pass the `sk_buff` to the upper layer. This is the standard for most NIC drivers[7]).

In this case, the upper layer is the Generic Receive Offload (GRO) layer [30]. The GRO attempts to aggregate multiple TCP segments into a single large packet. Specifically, the GRO converts multiple linear `sk_buff` buffers belonging to a single TCP stream, into a single `sk_buff` with multiple fragments. This `sk_buff` then traverses the Linux network stack and becomes a TX packet. The attacker can use this TX packet as described in the previous attack shown in Figure 8).

Packet forwarding, also opens up an additional attack option. An attacker may be interested in persistent surveillance rather than overtaking the machine. Packet forwarding allows the NIC to inspect arbitrary pages at will. Instead of sending a TCP packet and letting the GRO layer fill in the `frags` information, the NIC can generate a small UDP packet and fill in the `frags` array with any arbitrary `struct page` addresses within the system. As a result, the driver maps these pages, providing READ access to the NIC for any page in the system.

To avoid detection and preserve OS stability, the device must undo the changes to `skb_shared_info` before creating a TX completion. That is, before letting the CPU know that the packet was sent and its buffer can now be freed. Otherwise, the OS will try freeing the pages, indicated by `skb_shared_info`.

6 Attack Demonstrations

We implemented and demonstrated *compound* attacks against the Linux kernel network stack. In order to demonstrate an attack by a malicious NIC, we used a FireWire device similar to Sang et al. [62]. We created an IOVA page table that is shared between the FireWire and the actual NIC. Because

[6] Indeed, Google's OpenTitan [27] exemplifies that cloud providers actively worry about the root of trust for their servers.

[7] It is used by 98 NIC drivers in Linux 5.0

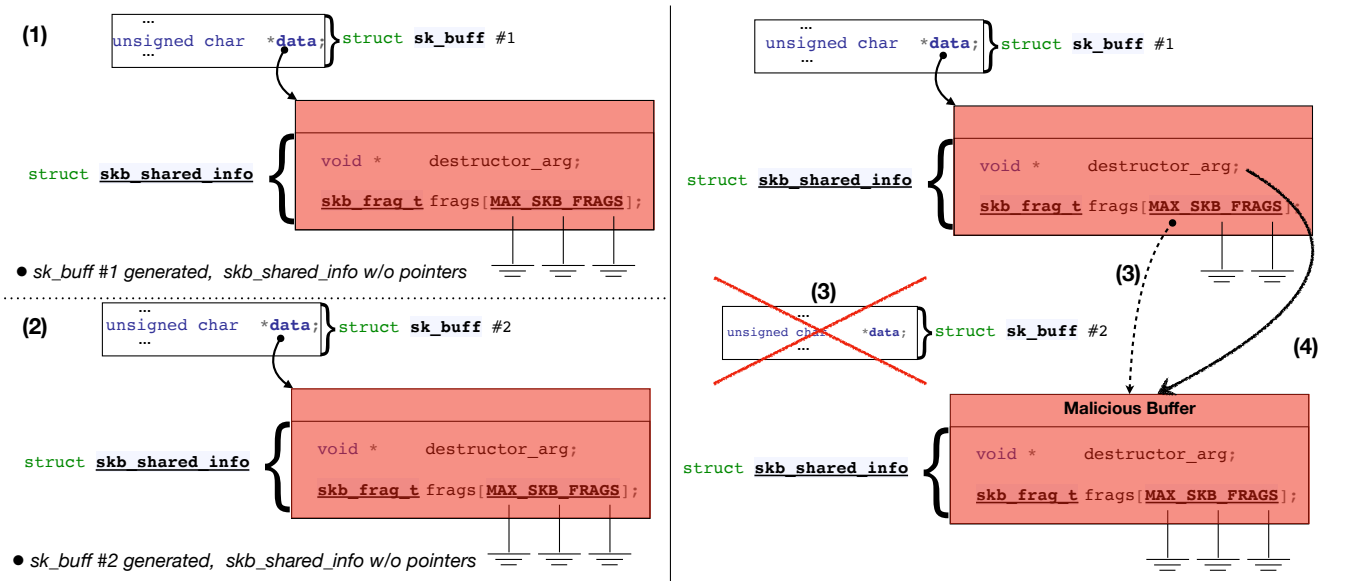

Figure 9. An RX `sk_buff` after GRO provides the KVA for a DMA attack.

the attacker machine can access the same pages as the NIC, this allowed us to execute an attack using a programmable interface, emulating a malicious NIC.

We created a malicious FireWire device by modifying the Linux-IO Target (LIO) subsystem on the attacker machine. The LIO subsystem supports hard disk emulation for remote computers via the SPB2 protocol.

Test Setup. We used a 28-core Dell PowerEdge R730 server, with Ubuntu 18.04 (kernel version 5.0), as our victim machine. This server is equipped with an Intel VT-d IOMMU, a Broadcom NetXtreme BCM5720 Gigabit Ethernet NIC, a Mellanox Technologies ConnectX-4 Ethernet NIC, and VIA Technologies, Inc. VT6315 Series Firewire Controller. We connected an identical machine to the victim via a FireWire cable, to act as the attacker.

Executed Attacks. We executed the *RingFlood* attack on the `skb_shared_info` structure to inject and run malicious code in the kernel. Our exploit places a ROP gadget on the DMA buffer page. To execute this ROP gadget, the device points the struct's callback pointer to a JOP gadget in the kernel. The kernel then passes the callback in the `%rdi` register to its containing struct. Thus, this pointer contains the DMA buffer's address. To complete the attack, we needed a JOP [10] gadget that performs `%rsp = %rdi + const`. We located such a gadget using the ROPgadget tool [61].

7 Applicability to Other OSs

The current state of IOMMU adaptation varies among different OS vendors. We briefly discuss other OSs below.

Windows. Until recently, Windows had no IOMMU support, exposing it to *single-step* DMA attacks. In 2019, with build 1803, Microsoft introduced Kernel DMA Protection [51], which provides IOMMU protection by default with a dedicated I/O page table per device. In addition, network buffers are allocated from dedicated pools of memory, limiting the possible exposure of sensitive data. However, a brief exploration of the Windows Networking drivers' API reveals functions such as *NdisAllocateNetBufferMdlAndData* [52] that allocates a NET_BUFFER structure and data in a single memory buffer, exposing the OS to *single-step* attacks. The NET_BUFFER vulnerability was previously described by Markettos et al.[45].

MacOS. IOMMU protection is enabled by default. It also uses dedicated memory for network I/O. MacOS, however, does expose the *mbuf* data structure to the device, though with some precautions such as blinding the exposed callback pointer *ext_free* by XORing it with a secret cookie. Indeed, this is sufficient to defend against *single-step* attacks. However, such an exposure of metadata opens up the MacOS to potential *compound* attacks. Although the value of the secret cookie is random, *ext_free* can receive only one of two possible values. As a result, once an attacker compromises MacOS KASLR (as demonstrated in [45]), the random cookie is revealed by a single XOR operation.

FreeBSD. An *mbuf* struct that is used for networking exposes the *ext_free* callback pointer. An attack on FreeBSD via this callback pointer was demonstrated by Markettos et al. [45]. To the best of our knowledge, as of October 2020, this vulnerability still exists in the FreeBSD kernel.

8 Related Work

In this section, we cover DMA attacks in the presence of IOMMU, defenses, and emerging ROP mitigation techniques.

DMA Attacks in the Presence of IOMMU. Beniamini demonstrated attacks on cellular devices (e.g., iPhone 7, Nexus 5/6/6P), through their WiFi chips [7, 8]. The attack exploited a Time of Check To Time of Use (TOCTTOU) vulnerability in the NIC driver. Kupfer [38] demonstrated *single-step* attacks exploiting weaknesses in the Linux FireWire driver. In both cases, all the DMA writes were legal, made only to buffers currently mapped to the device.

Thunderclap [45]. This work also considers sub-page vulnerabilities and *single-step* attacks. Markettos et al. developed a security evaluation platform built on FPGA hardware. By mimicking a legitimate peripheral device's functionality, the platform can convince a target operating system to grant it access to regions of memory. They used this platform to demonstrate *single-step* DMA attacks on Windows, macOS, and FreeBSD. Our work takes a step forward in characterizing, exploiting, and detecting DMA vulnerabilities. In particular:

- Thunderclap provides a taxonomy that differentiates between data leakage and kernel pointer attacks. We extend this taxonomy by characterizing the different types of sub-page vulnerabilities (Section 3.2).
- We explicitly characterize the attributes required for a successful code injection DMA attack. This allows us to better reason about a DMA attack focusing separately on each of its constituting parts.
- We introduce *compound* attacks and propose techniques to identify the buffer's KVA (Section 5.3, 5.4, 5.5), which enables their execution.
- We demonstrate (Section 6) that Linux is not safe from DMA attacks on the network data structures.
- We introduce new static [46] and dynamic [48] analysis tools that identify sub-page vulnerabilities, run them on Linux, and report many previously unknown DMA vulnerabilities.

Adressing IOMMU Vulnerabilities. Boyd-Wickizer and Zeldovich [11] and LeVasseur et al. [41] suggested isolating unmodified device drivers in user space programs and virtual machines, respectively. Similarly, Cinch [3] used an isolated red virtual machine to intercept bus traffic. These methods could be applied to limit the damage of potential attacks in addition to other protection mechanisms. They do not, however, prevent code execution in an isolated environment. By attacking the isolation mechanism, attackers can still compromise the entire system.

Markuze et al. suggested that the IOMMU driver should use bounce buffers [47]. Typically, device drivers invoke map/unmap requests for desired buffers through the DMA API. According to their suggestion, instead of dynamically mapping/unmapping pages, the DMA backend would copy the buffer to/from designated pages with fixed mapping. By keeping separate data pages for each device, they avoid data co-location and, as a result, eliminate the sub-page granularity vulnerability. Since the mappings are static, the issue of deferred invalidation is eliminated as well. Nevertheless, this solution imposes a large overhead of data copying and memory waste. In a later work, Markuze et al. suggested reducing these overheads by implementing the DMA-Aware Malloc for Networking (DAMN) [49]. The security of the system still depends on developers avoiding mistakes (e.g., not using `build_skb`) and does not provide a solution for packet forwarding or zero-copy I/O (e.g., `sendfile`, XDP [53]).

Intel's sub-page security technology suggests protecting fixed-sized buffers smaller than a page [34]. Since the buffers are still fixed in size, the same vulnerability remains, albeit for buffers smaller than a page. Intel's Memory Protection Extensions (MPX) lets the user define boundaries for buffers and later explicitly checks that the corresponding pointers are between these boundaries [31]. Oracle's Silicon Secured Memory (SSM) lets the user *color* buffers and associative pointers [58]. The color is implicitly checked for a match at each memory access. MPX, SSM, and other similar approaches may be used to build a secure alternative to IOMMU.

Emerging ROP Mitigation Techniques. Intel Control-Flow Enforcement Technology (CET) is a new instruction set for mitigating ROP attacks [33]. Processors that support CET use two stacks simultaneously instead of the regular one; the new shadow stack has only return addresses rather than a full copy of the data. During each RET command, the shadow stack address is checked, and the code continues running only if the stacks agree on the address. Even if an attacker manages to control the regular stack, the shadow stack prevents the attack. Moreover, each legitimate indirect jump target is marked with a special instruction. Thus, it is impossible to jump to arbitrary locations in the code, and JOP attacks are also prevented. Similarly, each legitimate call target is also marked. De Raadt [18] recently announced the Kernel Address Randomized Link (KARL) for OpenBSD. Each time the system is booted, it links a new, randomized kernel binary. As opposed to the Linux KASLR, this strong randomization makes it harder to patch the payload during run-time.

9 Discussion

By introducing and demonstrating *compound* attacks on the Linux kernel, we have shown that IOMMU, as it is used today, leaves the OS vulnerable to DMA attacks. Such vulnerabilities have been considered to be caused by buggy device drivers or poor, but isolated, driver design choices. We find that it is often the OS design choices that compromise the system security.

9.1 API

Existing APIs used for I/O operations and associated performance considerations make it is extremely difficult not to create a sub-page vulnerability that can later be exploited. Thus, even well-written drivers can be subverted by the OS

(e.g., bnx2 by deferred protection), as in the following examples.

- The *dma_map_single* call accepts a pointer and the buffer length. This API insinuates that only the mapped bytes are exposed, when, in fact, the whole page is accessible.
- *dma_unmap_single*, insinuates that the buffer is not accessible to the device after the call. This does not hold due to both the deferred protection and type (c) sub-page vulnerabilities.
- *build_skb* facilitates building an `sk_buff` around an arbitrary I/O buffer, in turn, embedding critical data structures inside the I/O buffer.
- While `page_frag` (Section 5.2.2) is an efficient allocator, it inherently creates a type (c) sub-page vulnerability.
- By design, `skb_shared_info` is built within an I/O buffer. Avoiding type (b) sub-page vulnerabilities imposes a challenge.

9.2 Conclusion

The success of a DMA attack relies on the exposure of restricted metadata fields, caused by sub-page vulnerability. To prevent such exposure, previous works proposed separating the I/O memory from CPU memory [49], by providing a separate allocator for networking. Nevertheless, this API can be easily thwarted by device drivers via functions, such as `build_skb`, that add a vulnerable `skb_shared_info` into an I/O region. Moreover, these solutions are focused solely on network devices, leaving the system unprotected against other DMA-capable devices such as FireWire, USB C, NVMe, and more.

To achieve better DMA security in future OSs, one possible direction is to consider the segregation of I/O memory from OS memory. Alternatively, to prevent the existence of sensitive metadata on I/O pages, we propose to open-source and offer SPADE [46] and D-KASAN [48] to validate the security of the system in the development and deployment stages.

Acknowledgments

This work was funded in part by ISF under grant 2005/17, Blavatnik ICRC at TAU, Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate, and VMware Research.

References

[1] AMD. AMD IOMMU architectural specification, rev 3.00. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf, Dec 2016. Accessed: June 2020.

[2] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *ATC 11*, 2011. https://www.usenix.org/legacy/events/atc11/tech/final_files/Amit.pdf Accessed: June 2020.

[3] Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against malicious peripherals with Cinch. In *USENIXSEC*, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel Accessed: June 2020.

[4] argp and karl. Exploiting UMA : FreeBSD kernel heap exploits. *PHRACK*, 13(66), November 2009. http://phrack.org/issues/66/8.html Accessed: June 2020.

[5] Damien Aumaitre and Christophe Devine. Subverting windows 7 x64 kernel with dma attacks. *HITBSecConf Amsterdam*, 2010.

[6] Michael Becher, Maximillian Dornseif, and Christian N Klein. FireWire: all your memory are belong to us. https://cansecwest.com/core05/2005-firewire-cansecwest.pdf, 2010. CanSecWest Presentation. Accessed: Jun 2020.

[7] Gal Beniamini. Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices. https://googleprojectzero.blogspot.co.il/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html, 2017. Accessed: June 2020.

[8] Gal Beniamini. Over the air: Exploiting broadcom's wi-fi stack. *Last retrieved*, 7, 2017.

[9] Erik-Oliver Blass and William Robertson. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78. ACM, 2012.

[10] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS*, 2011.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *ATC*, 2010.

[12] Hsiao-keng Jerry Chu. Zero-copy tcp in solaris. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 21–21. Usenix Association, 1996.

[13] Catalin Cimpanu. $300 device can steal Mac FileVault2 passwords. https://www.bleepingcomputer.com/news/security/-300-device-can-steal-mac-filevault2-passwords/, December 2016. Accessed: June 2020.

[14] Jonathan Corbet. Hot and cold pages. https://lwn.net/Articles/14768/. Accessed: Jun 2020.

[15] Jonathan Corbet. The SLUB allocator. https://lwn.net/Articles/229984/, April 2007. Accessed: June 2020.

[16] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O'Reilly Media, Inc.", 2005.

[17] Arnaldo Carvalho de Melo. The 7 dwarves: debugging information beyond gdb. In *Proceedings of the Linux Symposium*. Citeseer, 2007.

[18] Theo de Raadt. KARL - kernel address randomized link. https://marc.info/?l=openbsd-tech&m=149732026405941, October 2017. Accessed: Jun 2020.

[19] Robert C De Ward and Kenneth J Thurber. Method for providing virtual addressing for externally specified addressed input/output operations, May 15 1979. US Patent 4,155,119.

[20] Linux Kernel Developers. The kernel address sanitizer (kasan)—the linux kernel documentation, 2017. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[21] Maximillian Dornseif. 0wned by an ipod. *Presentation, PacSec*, 2004.

[22] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.

[23] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/. Accessed Jun 2020.

[24] Ulf Frisk. Direct memory attack the kernel. *Proceedings of DEFCON*, 24, 2016.

[25] Sean Gallagher. Photos of an NSA "upgrade" factory show Cisco router getting implant. *Ars Technica*, 14, 2014. http://arstechnica.com/tech-policy/2014/05/\photos-of-an-nsa-upgrade-factory-show-cisco\-router-getting-implant/ Accessed: June 2020.

[26] Pavel Gladyshev and Afrah Almansoori. Reliable acquisition of ram dumps from intel-based apple mac computers over firewire. In *International Conference on Digital Forensics and Cyber Crime*, pages 55–64.

Springer, 2010.
[27] Google. Opentitan. https://opentitan.org/. Accessed Jun 2020.
[28] Free Standards Group. Dwarf. http://dwarfstd.org/.
[29] Dave Hansen. x86: mm: set TLB flush tunable to sane value (33). linux-mm mailing list https://patchwork.kernel.org/patch/4066561/, 2014. Accessed: Jun 2020.
[30] Red Hat. Performance tuning guide 8.10. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-nic-offloads. Accessed: June 2020.
[31] INTEL. Intel-64 and IA-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm, December 2016. Accessed: Jun 2020.
[32] INTEL. Intel virtualization technology for directed I/O - architecture specification - Rev. 2.4. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, June 2016. Accessed: June 2020.
[33] INTEL. Intel control-flow enforcement technology preview - Rev. 2.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, June 2017. Accessed: Jun 2020.
[34] INTEL. Intel architecture instruction set extensions and future features programming reference - may 2018. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, May 2018. Accessed: June 2020.
[35] Corbet Jonathan. x86 nx support. https://lwn.net/Articles/87814/, June 2004. Accessed: June 2020.
[36] Linux Kernel. X86 kernel memory layout. https://elixir.bootlin.com/linux/latest/source/Documentation/x86/x86_64/mm.rst. Accessed: June 2020.
[37] Eduardo Kortright and David Cordes. Cnest and cscope: Tools for the literate programming environment. In *Proceedings IEEE Southeastcon'92*, pages 604–609. IEEE, 1992.
[38] Gil Kupfer, Dan Tsafrir, and Nadav Amit. Iommu-resistant dma attacks. Master's thesis, Computer Science Department, Technion, 2018.
[39] Bell Labs. Cscope. http://cscope.sourceforge.net/.
[40] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.
[41] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.
[42] Carsten Maartmann-Moe. Inception. *Break & Enter:*, 2011. Accessed: June 2020.
[43] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. riommu: Efficient iommu for i/o devices that employ ring buffers. *ACM SIGPLAN Notices*, 50(4):355–368, 2015.
[44] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient intra-operating system protection against harmful dmas. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 29–44, 2015.
[45] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.
[46] Alex Markuze. Sub-Page Analysis for DMA Exposure (SPADE). https://github.com/Markuze/mmo-static.git, 2021.
[47] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ASPLOS*, pages 249–262, 2016.
[48] Alex Markuze and Boris Pismeny. DMA Kernel Address SANitizer (DMA-KASAN). https://github.com/Markuze/dma-kasan.git, 2021.
[49] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-free iommu protection for networking. In *ASPLOS*, 2018.
[50] Mellanox. Mellanox adapters programmer's reference manual. http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf. Accessed: June 2020.
[51] Microsoft. Kernel dma protection. https://docs.microsoft.com/en-us/windows/security/information-protection/kernel-dma-protection-for-thunderbolt.
[52] Microsoft. Ndisallocatenetbuffermdlanddata. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ndis/nf-ndis-ndisallocatenetbuffermdlanddata.
[53] David Miller. Xdp mythbusters. https://netdevconf.info/2.1/slides/apr7/miller-XDP-MythBusters.pdf. Accessed: June 2020.
[54] Tilo Müller, Benjamin Taubmann, and Felix C Freiling. Trevisor. In *International Conference on Applied Cryptography and Network Security*, pages 66–83. Springer, 2012.
[55] Karsten Nohl and Jakob Lell. Badusb-on accessories that turn evil. *Black Hat USA*, 1:9, 2014.
[56] Hussein Nur. Randomizing structure layout. https://lwn.net/Articles/722293/, May 2017. Accessed: June 2020.
[57] United States. National Bureau of Standards. *Computer Development, SEAC and DYSEAC, at the National Bureau of Standards, Washington*. US Government Printing Office, 1954.
[58] ORACLE. Inoculating software, boosting quality. http://www.oracle.com/technetwork/database/bi-datawarehousing/sas/con8216-final-2760803.pdf, 2015. CON8216. Accessed: Jun 2020.
[59] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the {IOMMU} scalably. In *2015 {USENIX} Annual Technical Conference (USENIX ATC)*, pages 549–562, 2015.
[60] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *TISSEC*, 15(1):2:1–2:34, March 2012.
[61] Jonathan Salwan. Ropgadget. https://github.com/JonathanSalwan/ROPgadget. Accessed Mar 2021.
[62] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *MALWARE*, pages 7–14, 2010.
[63] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.
[64] FinFisher surveillance. Finfirewire, 2014. (WikiLeaks).
[65] Aaron Walters. The volatility framework: Volatile memory artifact extraction utility framework, 2007.
[66] Paul Willmann, Scott Rixner, and Alan L Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX Annual Technical Conference*, pages 15–28, 2008.
[67] Rafal Wojtczuk et al. Subverting the xen hypervisor. *Black Hat USA*, 2008:2, 2008.
[68] Lucas Womack, Ronald Mraz, and Abraham Mendelson. A study of virtual memory mtu reassembly within the powerpc architecture. In *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 81–90. IEEE, 1997.
[69] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *2016 IEEE symposium on security and privacy (SP)*, pages 18–37. IEEE, 2016.
[70] Jiewen Yao and Vincent Zimmer. A tour beyond bios using intel vt-d for dma protection in uefi bios, 2015.
[71] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th annual computer security applications conference*, pages 279–288. ACM, 2013.