Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals

A. Theodore Markettos†, Colin Rothwell†, Brett F. Gutstein†*,

Allison Pearce†, Peter G. Neumann‡, Simon W. Moore†, Robert N. M. Watson†

†University of Cambridge
Dept. Computer Science and Technology

‡SRI International

*Rice University

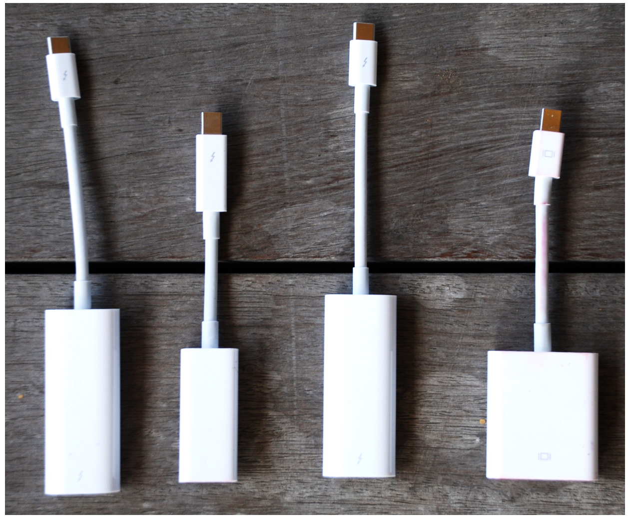
Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 and HR0011-18-C-0016. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

SRI International

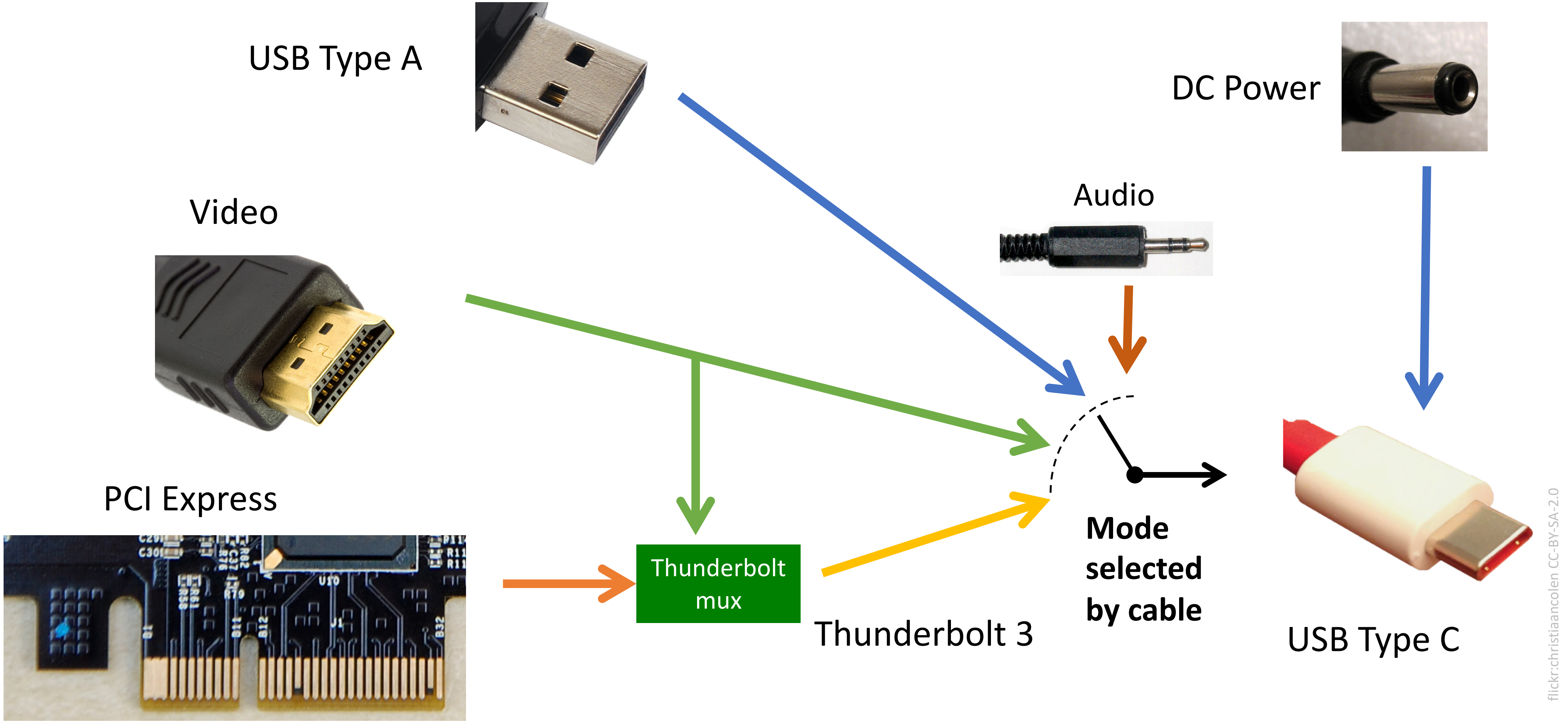
UNIVERSITY OF CAMBRIDGE

Smaller laptops, more external peripherals

- Laptops getting smaller, more devices are going external
  - Chargers, dongles, docking stations
  - Common to borrow external peripherals (power, dongles, displays) from others
- Performance is increasingly more of a constraint
- Security?



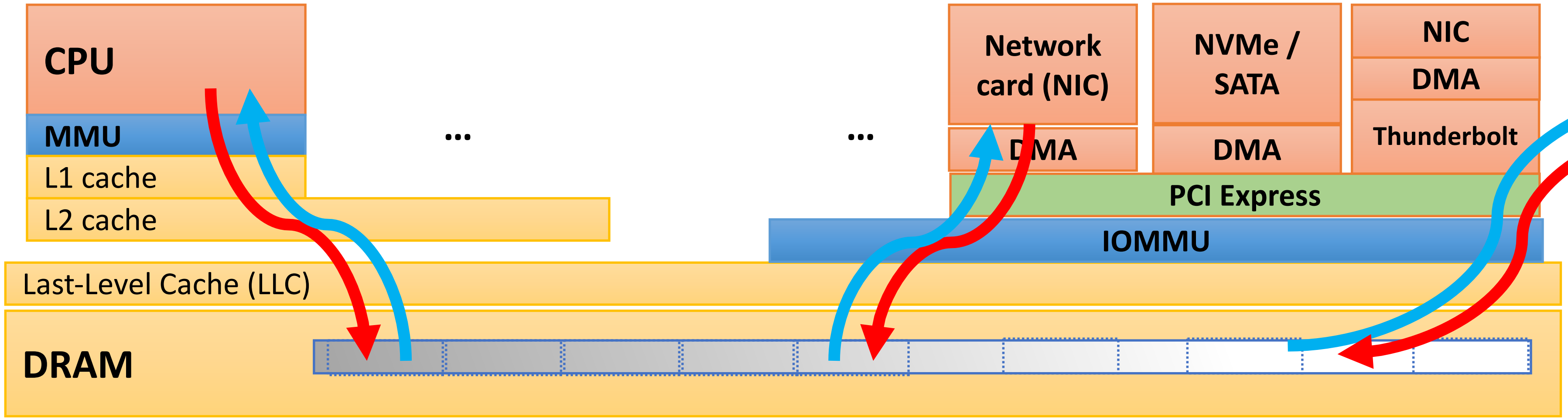
USB-C convergence: can’t tell protocol from the connector



The security story...

- USB is a message-based protocol
  - people craft bad messages (eg BadUSB)
  - attack the device driver stack (buffer overflow, etc)
- PCI Express is a shared-memory protocol
  - 'DMA' means devices Directly Access Memory, not via CPU
  - used when performance is more critical
  - wide exposure of system memory, all data on the system is accessible
  - PCIe threat model: existing chips/cards with bad firmware update/compromise (eg in servers)
- Thunderbolt is a multiplex of PCI Express and DisplayPort video over USB-C (or miniDisplayPort)
  - Thunderbolt threat model: can now hotplug DMA-capable devices into running systems
  - Do everything PCIe devices can do and more
  - Even more scope for user confusion
- Surely there are defenses?

IOMMU = an MMU for device memory accesses



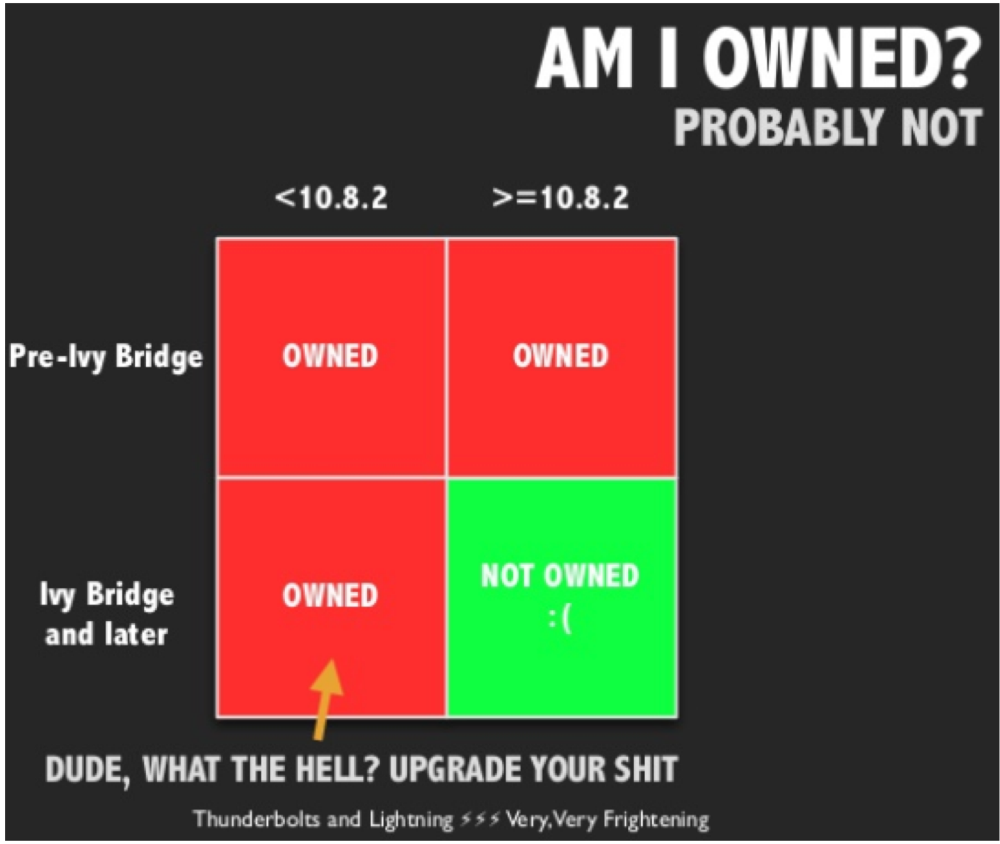
- MMU used to protect & virtualize memory access from *processes*
- IOMMU used to protect & virtualize memory access from *peripheral devices*
- One IOMMU page table per device
  - mapping 4KiB / 2MiB / 1GiB pages into I/O Virtual Address space
- IOTLB as a cache of recently-used translations (*c.f.* TLB in MMU)
- Implementations: Intel VT-d, AMD-Vi, Arm System MMU

Use of the IOMMU to protect from I/O devices?

- ✗ Windows 7 / 8 : don't use the IOMMU, all memory exposed
- ✗ Windows 10 Home/Pro : didn't use the IOMMU
- MacOS ≥10.8.2 : IOMMU enabled by default
- ✗ Linux : supported, but IOMMU rarely enabled by default
- ✗ FreeBSD : supported, but not enabled by default
- ✗ IOMMU often disabled in default firmware settings (BIOS, UEFI)
- **Current state of the world is not good**
  - Our work assumes that the OS vendor is at least vaguely trying...
  - What is the attack surface if they turned on IOMMU protection?

The attack surface from a real device

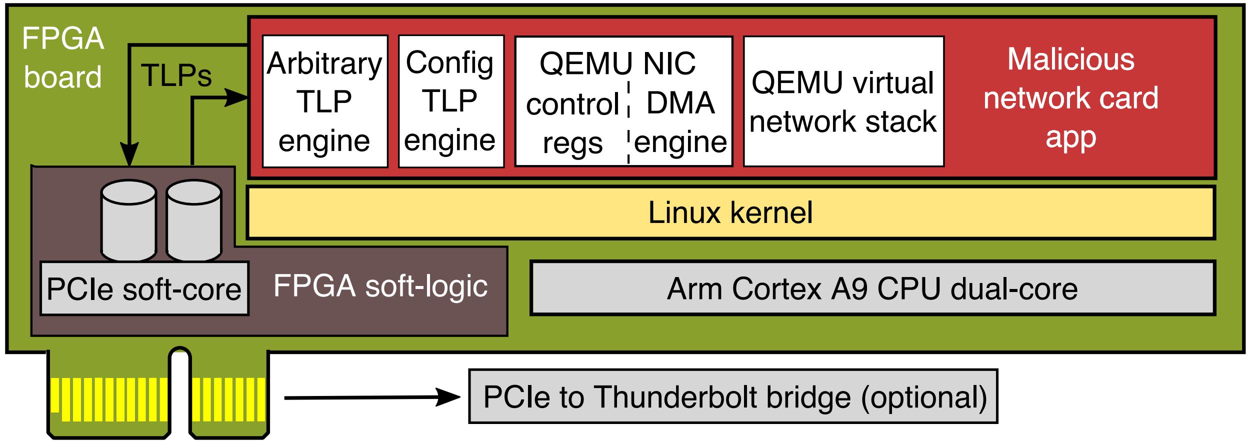
- prior work: "when the IOMMU is enabled, attacks are foiled"
  - these are simple memory-probing attacks
  - no interactions with driver or kernel
- actually, the attack surface is much more nuanced
- what attack surface does a real I/O device have?
  - what accesses can it make?
  - how does it interact with the device driver stack?
  - as the OS increasingly trusts it, what extra vulnerabilities does it open up?



snare and rzn, *Thunderbolts and Lightning – Very Very Frightening* (2014)

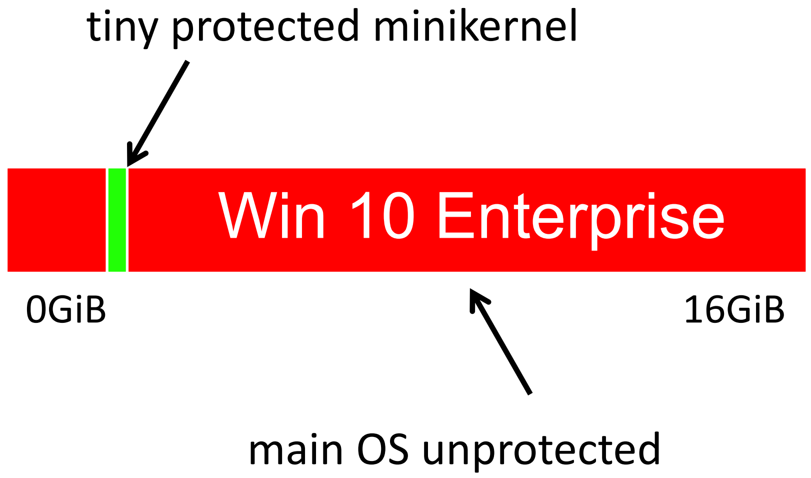
Thunderclap: a research platform for I/O security

- We built a fake network card:
  - software device model of an Intel E1000 PCIe ethernet card from QEMU
    - software = easy to change, add malicious behavior
  - run it on a CPU on an FPGA (Arm Cortex A9 on Intel Arria 10, running Ubuntu)
    - FPGA logic can send and receive arbitrary PCIe packets
    - QEMU model responds to PCIe packets and generates 'DMA' like a real NIC
  - runs on FPGA dev boards, attached via PCIe or Thunderbolt dock
  - hardware/software open sourced
  - designed physical embodiments
    - Thunderbolt dock implant
    - malicious projector, charger
    - not fully engineered/productized
    - not released at this time

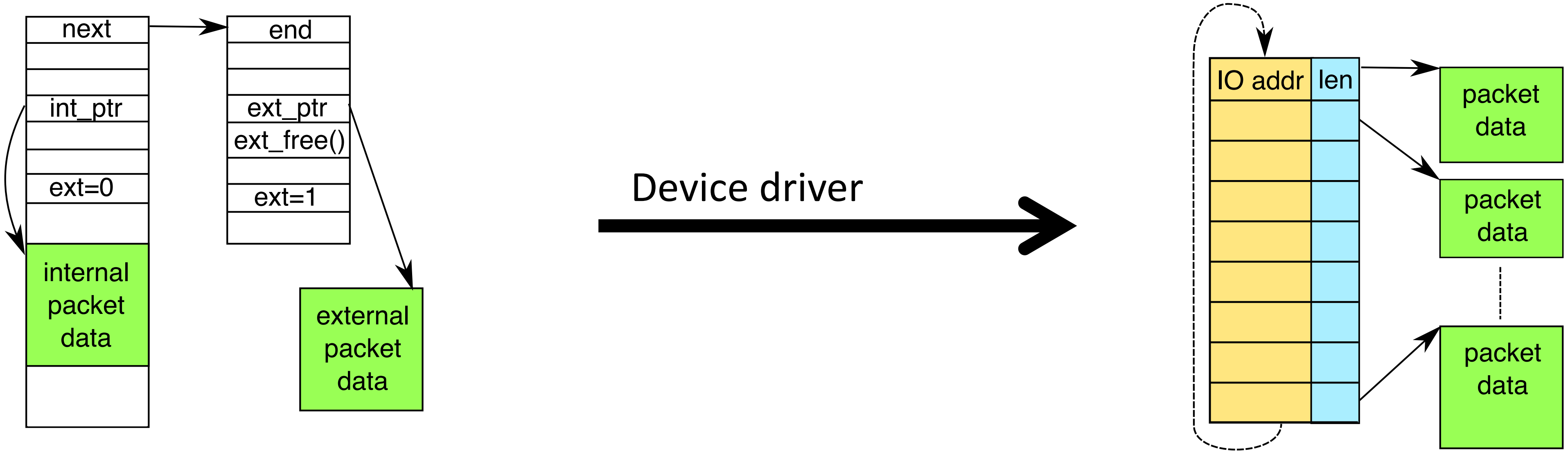


Attack: Windows 10

- Windows 10 Home/Pro don't use the IOMMU
- Windows 10 Enterprise doesn't by default
- Enterprise can enable Virtualization Based Security: runs the main OS in a HyperV VM
  - second minikernel for key storage, etc
- Under VBS: I/O device has full access to all system memory except the few pages of minikernel are protected
- Attacker can get everything except the disk encryption keys
  - keyloggers
  - filesystem plaintext
  - run arbitrary code
  - screen capture
  - network traffic
  - much more...



Network card (NIC): common design patterns



OS packet structure with internal or external data

structure contains external data free() function pointer

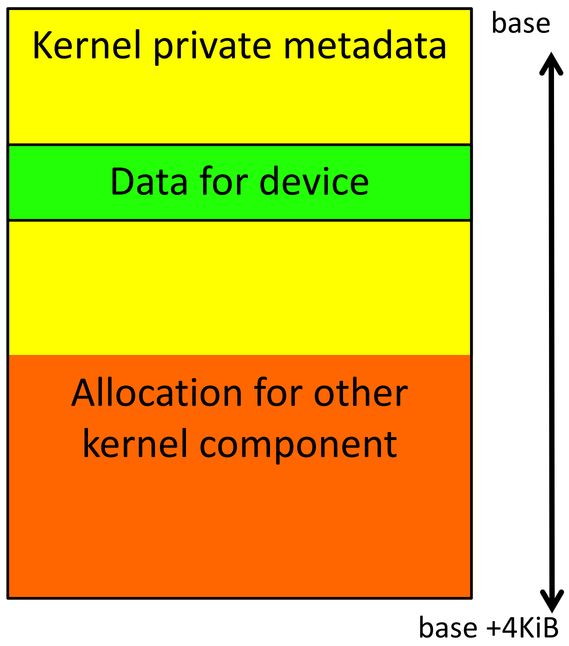
mbuf (MacOS/FreeBSD)
skbuff (Linux)
NET_BUFFER_LIST (Windows)

NIC-specific ring buffer

1. NIC reads table by DMA
2. follows pointers in its I/O virtual address space
3. reads/writes data blocks by DMA to send/receive

IOMMU vulnerability taxonomy

- IOMMU *windows* = regions of memory exposed to a device, sized in pages
- *Spatial vulnerability*
  - 4KiB page granularity isn't fine enough to distinguish data fields in complex data structures like *mbufs*
  - Read or write memory we aren't supposed to access
- *Temporal vulnerability*
  - Exploit the time gap between asking for a window to be closed and closure taking place
  - Memory gets reused for something else in the interim



Attack: MacOS data leakage and root shell

- MacOS
  - all devices share one page map
    - NIC can't read/write kernel or apps memory, but can access USB buffers, framebuffer
  - mbufs are allocated in a single block and exposed to all devices at boot time
    - access all of the network data all of the time – traffic for other NICs, VPN plaintext, etc
  - Kernel-Address Space Layout Randomization (KASLR) can be broken due to leaked USB symbol
  - free() function pointer and 3 parameters from mbuf allow launching a root shell

```
struct mbuf {
  ...
  struct m_ext;
  ...
  // internal buffer
  char M_databuf[224];
};

struct m_ext {
  // external buffer pointer
  caddr_t ext_buf;
  // free() function pointer
  void (*ext_free)(caddr_t,
          u_int, caddr_t);
  u_int   ext_size;
  ...
  struct ext_ref {
    u_int32_t refcnt;
    // buffer is external flag
    u_int32_t flags;
  } *ext_refflags;
};
```

iMac (victim) ---- **PCIe** ---- Thunderclap (Intel e1000 model)

Device discovery and driver attachment

Hi! What are you?

I'm an … Intel e1000 NIC .. I promise!

Oh cool, I've got the perfect large, buggy, and highly vulnerable vendor-provided device driver just for you!

Attacker selects their device driver of choice via the returned PCI device ID

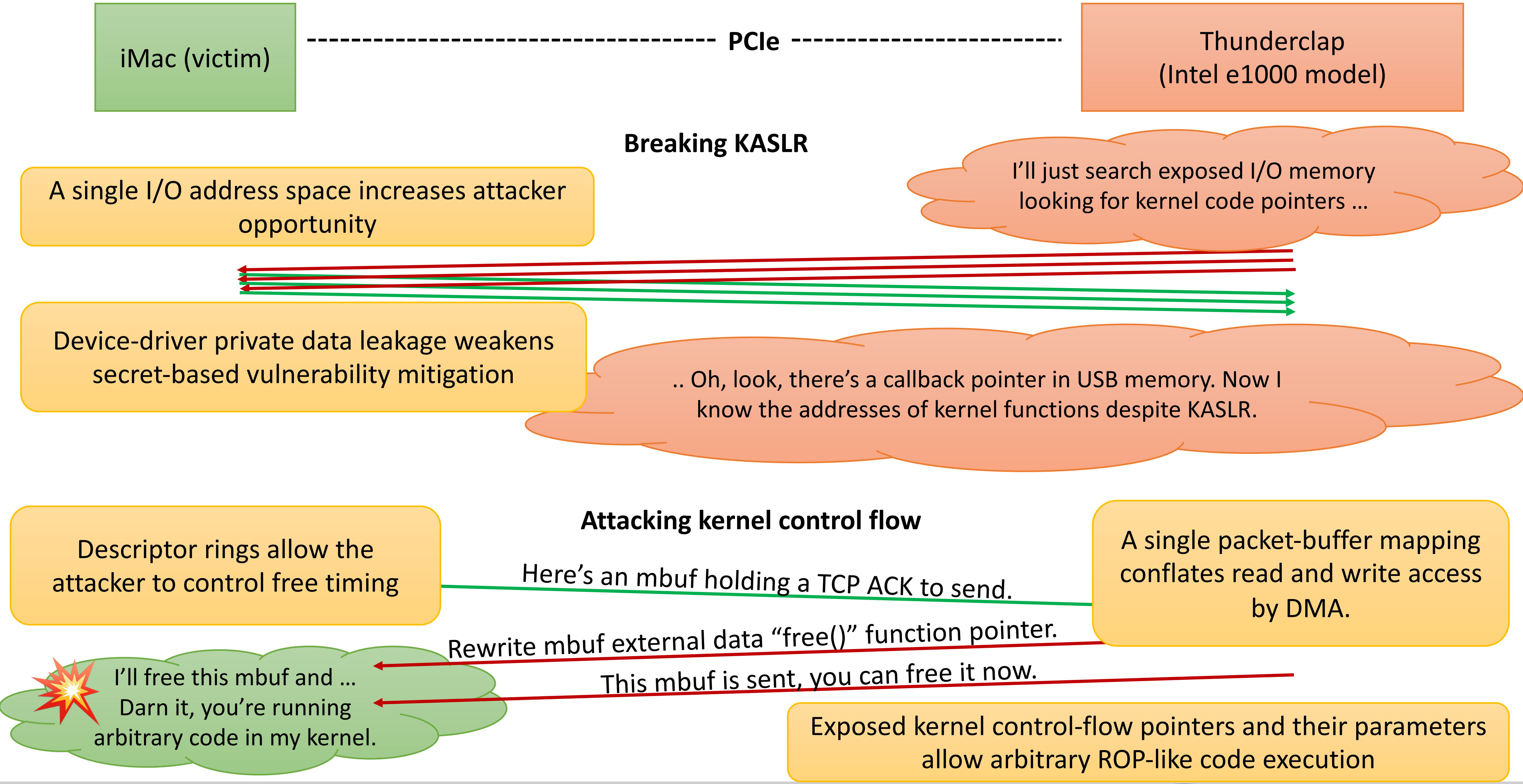
Device-driver/NIC protocol enters steady state.

Here are the descriptor rings, other parameters.

Great, because I'm a NIC.

The attacker can source and sink packets, allowing it to interact with OS state: respond to DHCP, make and accept TCP connections, trigger OS services launching, etc.

Use spatial vulnerability to look in IOMMU windows for sensitive leaked data, change it

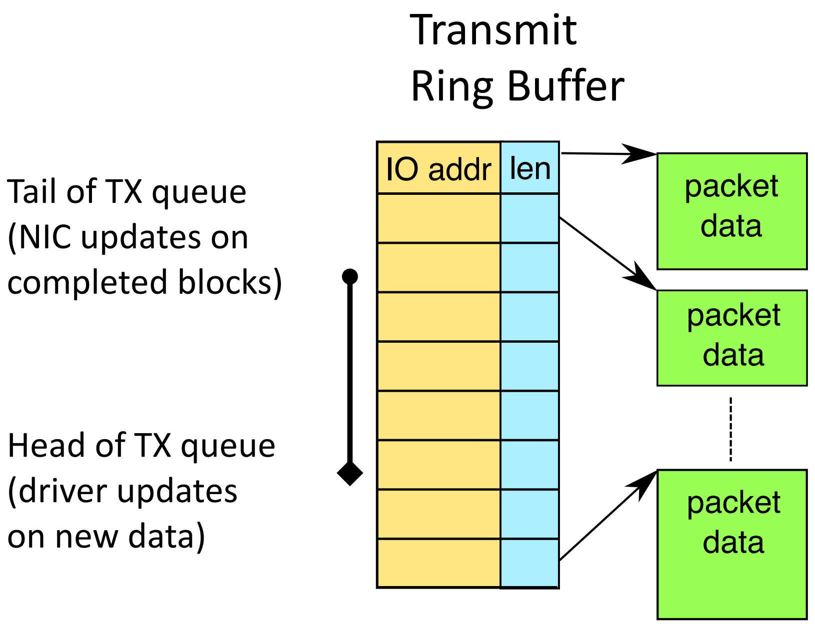

iMac (victim)
PCIe
Thunderclap
(Intel e1000 model)
Breaking KASLR
A single I/O address space increases attacker opportunity
I'll just search exposed I/O memory looking for kernel code pointers …
Device-driver private data leakage weakens secret-based vulnerability mitigation
.. Oh, look, there's a callback pointer in USB memory. Now I know the addresses of kernel functions despite KASLR.
Attacking kernel control flow
Descriptor rings allow the attacker to control free timing
Here's an mbuf holding a TCP ACK to send.
A single packet-buffer mapping conflates read and write access by DMA.
Rewrite mbuf external data "free()" function pointer.
This mbuf is sent, you can free it now.
I'll free this mbuf and … Darn it, you're running arbitrary code in my kernel.
Exposed kernel control-flow pointers and their parameters allow arbitrary ROP-like code execution

Attack variations

- FreeBSD
  - one page map per device
  - see other network traffic co-located on pages (traffic for other NICs, VPN plaintext)
  - no KASLR: root shell attack works
- Linux
  - one page map per device
  - data and metadata on different pages – can't overwrite free() pointer
  - general kernel allocator used by driver
    - see Unix domain socket traffic (as used by SSH agent)
    - kernel NAT jump tables, potentially lots more...

Spatio-temporal attack

- Driver increments head pointer of ring buffer to indicate new data to send
- NIC increments tail pointer to indicate a block has been sent and can be freed / IOMMU window closed
- NIC can hold on to pages it has been asked to transmit by not updating the tail pointer
  - 'I'm not done with this block yet, please keep the IOMMU window open'
- Watch other parts of these page change as they are reused multiple times
  - data for other NICs
  - VPN plaintext



Attack: Linux IOMMU bypass

- PCIe has a feature called Address Translation Services (ATS)
- Allows PCIe to carry pre-translated addresses
  - Performance mitigation to cache translations locally, don't have to go inter-socket to IOTLB on a multi-socket server
- ‘Pre-translated addresses’ means we can generate memory reads/writes to arbitrary physical addresses with no IOMMU interposing
- Set Thunderclap to advertise PCIe configuration registers saying it supports ATS
- Linux sees this and enables ATS on the PCIe switches
- Set a bit in the PCIe packet header saying an address is pre-translated
- We've completely bypassed IOMMU protection!

31 … 0

<table>
<tr><td>Fmt</td><td>Type</td><td>R</td><td>TC</td><td>R</td><td>Attr</td><td>AT</td><td></td><td colspan="2">Length</td></tr>
<tr><td colspan="5">Requester ID</td><td colspan="3">Tag</td><td>Last BE</td><td>1st BE</td></tr>
<tr><td colspan="10">Address</td></tr>
<tr><td colspan="10">Data word 0</td></tr>
</table>

MemoryWrite32
TLP

The IOMMU attack surface

- The attacks shared-memory devices can do are rich, complex and nuanced
  - Substantially more powerful than attacks by message-passing devices such as USB
- Most systems are poorly defended
- Look a lot like the syscall interface
  - OS kernels are protected from processes by the MMU and carefully vet syscalls from untrustworthy code
  - Syscalls have a long history of hardening and code audit
- OS kernels are barely protected from devices by the IOMMU and accesses from devices
  - A large body of buggy and poorly tested device driver code
  - Often provided by third-parties
  - Malicious device can pick its shape to target the most vulnerable device driver

How can it be this bad?

- Elephant in the room: performance
  - forcing all I/O through an additional layer of address translation is expensive
    - worst case: walking 6 level page table
  - IOTLB caches to mitigate this are typically too small to be effective
  - synchronously revoking mappings can be very time consuming
  - performance optimizations like ATS can be a security vulnerability
- Explains why the IOMMU is not enabled by default, or used minimally (as MacOS)

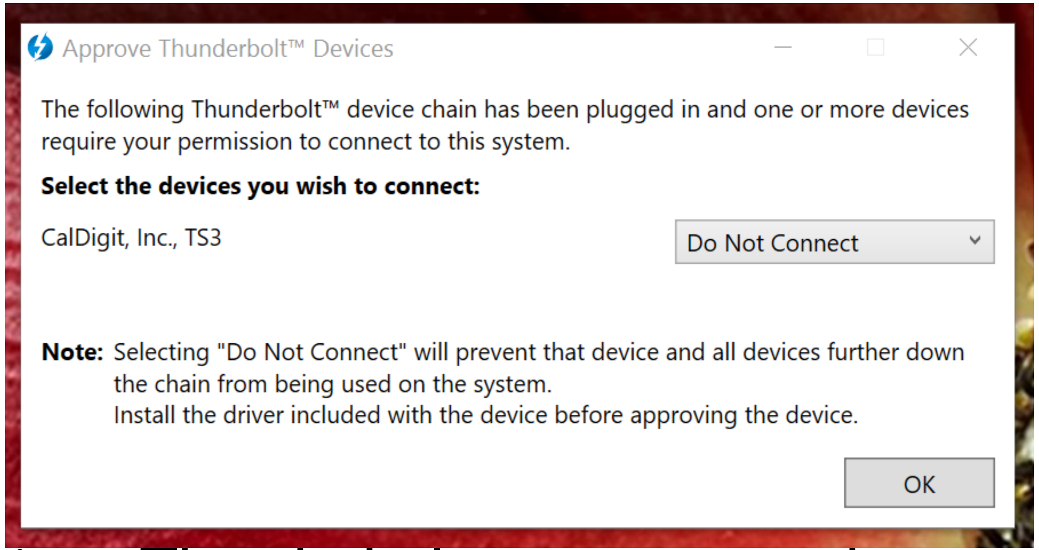
DON'T PANIC: Mitigations already fielded

- Collaborating with vendors since 2016
- Apple mitigated specific exploit in MacOS 10.12.4
  - encrypt the kernel pointer, hide the flags
- Microsoft shipped Kernel DMA Protection for Thunderbolt 3 in Windows 10 1803
  - IOMMU enabled for Thunderbolt devices (only)
  - Requires post-1803 firmware, ie new products only
- Intel enabled IOMMU for Thunderbolt in Linux 4.21 (now 5.0rc), disabled ATS
- We assume an active IOMMU, so our attacks still relevant for Windows and Linux
- Major laptop vendor: we won't ship Thunderbolt until we understand this attack vector better
- Eternal vigilance: DMA turning up in numerous new places – PCIe in phones, SD card 7.0, NVMe over Ethernet...

Conclusion

- We present the IOMMU attack surface as a new and rich field for vulnerabilities
- Open sourced Thunderclap, a research platform that allows exploration from an FPGA
- Told some stories of attacks across four major OS platforms
  - including a complete IOMMU bypass
- Vendors shipped mitigations to our attacks which are already fielded
- Solving the problem in the general case is harder than it appears, and some major work may be required
- Source code and FAQ: thunderclap.io

Thunderbolt access control

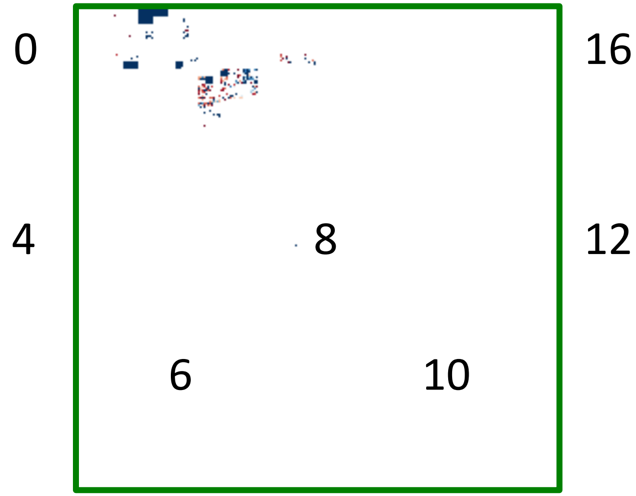


- On Windows and Linux, Thunderbolt can prompt when a new device is connected
- Prompt gives no information about the rights being requested
- Users can't make any kind of informed decision whether to allow it
- Can't detect modifications to a device above the Thunderbolt layer
- MacOS doesn't prompt, just need to buy a Thunderbolt dock on the whitelist

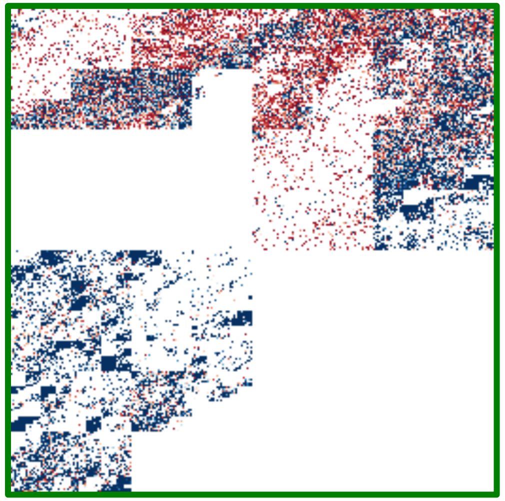
Physical address utilisation study

- Hypothesis: network devices reuse memory in a way that storage or GPUs don't
- Reuse-based mitigations might not be efficient?
- Method: use PCIe analyser to record physical address patterns of different devices, under real-world applications
- Result: very different access patterns
- Accelerators (GPUs, TPUs?) have deeper sharing behaviours that IOMMU wouldn't handle efficiently
- Need better strategies, or a better IOMMU?

Heat map of 16GiB physical address space



10G NIC, Linux pgbench



GPU, Windows 10, 4K demo

PCIe Address Translation Services (ATS)

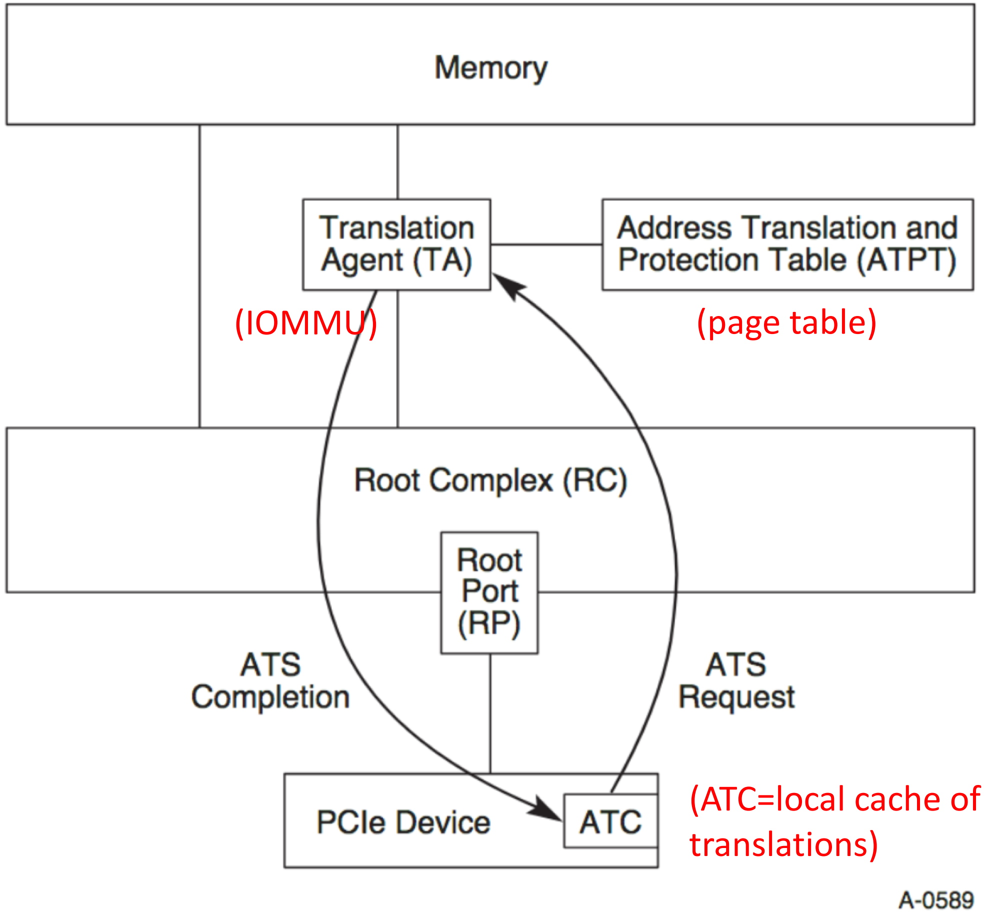


Figure 1-2: Example ATS Translation Request/Completion Exchange

Attack: MacOS data leakage

- First OS that really tried using the IOMMU
  - enabled by default since 10.8.2 (2012)
- All devices share a common page map
  - NIC can't see kernel or apps but can see USB, framebuffer...
- Network packets (mbufs) are all pre-allocated and exposed at boot time
  - both the data and the mbuf data structure is exposed due to 4KiB page granularity
- MacOS data leakage (10.8.2 to present)
  - when a NIC is given a packet to send...
  - look nearby for data stored in other packets
    - find traffic for other NICs, VPN plaintext...
  - worse: can see all of the network traffic all of the time

Attack: MacOS root shell

- notice that mbufs can contain a function pointer to a custom free() function
- NIC can change mbuf flags so that kernel calls ext_free() when NIC indicates a packet is sent
  - ext_free() arguments are also read from the mbuf
- replace function pointer with our own, control 3 arguments it is called with
- need to defeat KASLR address randomisation
  - USB driver also leaks kernel symbol
  - use it calculate the KASLR offset, generate a valid function pointer from any symbol in the kernel
- NIC sets function pointer to `KUNCExecute`, parameters -> ‘`Terminal.app`’
- We have a root shell ☺

```
struct mbuf {
  ...
  struct m_ext;
  ...
  // internal buffer
  char M_databuf[224];
};

struct m_ext {
  // external buffer pointer
  caddr_t ext_buf;
  // free() function pointer
  void (*ext_free)(caddr_t,
          u_int, caddr_t);
  u_int   ext_size;
  ...
  struct ext_ref {
    u_int32_t refcnt;
    // buffer is external flag
    u_int32_t flags;
  } *ext_refflags;
};
```

Attack: FreeBSD data leakage and root shell

- The same attack also works
- Tries harder, different page table per device
  - NIC devices can only see their own data, no other memory exposed
- Same mbuf data structure, MacOS attack still works
  - 4KiB page granularity means we can look in other parts of pages we are asked to transmit/receive
    - can't see all network packets, only ones we share pages with
  - no KASLR so root shell easier

Attack: Linux kernel leakage

- Linux gives each device its own page table
- *skbuffs* put data and metadata on different pages, so free() pointer isn't accessible to us
- but data is often allocated by drivers from a common pool, other parts of pages we are given still leak
  - Unix domain socket traffic (as used by SSH agent)
  - kernel NAT jump tables
  - potentially lots more...

SRI International
UNIVERSITY OF CAMBRIDGE

Outline

- A story of a new attack vector...
  - peripheral devices are not your friends
- A new platform for investigating security of peripheral devices and operating systems
- New classes of vulnerabilities, new exploit techniques
- Some real-world attacks
- Mitigations and... why didn't they do it right first time?
- Conclusion

Type C: 'The USB that does it all'

- USB Type C is a connector standard (not a communication protocol)
  - 'Alternate modes': cable switches port to carry a different protocol
- Thunderbolt is an 'alternate mode' of the Type C connector
  - Thunderbolt interconnect = packetised multiplex of PCI Express & DisplayPort