Beyond The CPU: Defeating Hardware Based RAM Acquisition (part I: AMD case)

Joanna Rutkowska
COSEINC Advanced Malware Labs

Focus

- In this presentation we focus on x86/x64 architecture, and specifically on AMD64 based systems.

Why do we need RAM acquisition?

- Find out whether a given machine is compromised or not
- Forensic Analysis
  - Find out how malware “works”
  - Use as an evidence
- Most forensics analysts focus on persistent memory – i.e. hard disk images
- This is obviously not enough, because malware can be non-persistent
- So, we need a reliable way to get an image of RAM…

Approaches to memory acquisition

- Software-based
  - Usually uses `/dev/mem` or `\Device\PhysicalMemory`
  - Requires additional software to be run on a target system
    - e.g. *dd/dd.exe, EnCase (?), ProDiscover(?)*
- Hardware-based
  - e.g. a PCI or PCMCIA card
  - Uses DMA access to read physical memory
  - No additional software on the target machine required
  - OS-independent

Software-based acquisition

- Not reliable!
  - Can be cheated by malware which runs at the same privilege level as the imaging software:
    - Shadow Walker Rootkit
    - `\Device\PhysicalMemory` memory hooking
    - Implementation Specific Attacks against acquisition software
- Requires additional software on the <u>target machine!</u>
  - This violates the requirement that forensic tools shall not cause data to be written to the target machine

Hardware-based solutions

- Reliable!
  - Direct Memory Access does not involve CPU
  - Acquisition device “talks” directly to the memory controller
  - Even if the whole OS is compromised, still we can get a real image of the physical memory
  - “The real image” – i.e. the same image as the CPU sees
- No additional software on the target – good!
- Possible race conditions when reading memory, because systems (i.e. CPU) is still “running”…
  - Is it possible for a PCI device to freeze the host’s CPU?

Hardware-based solutions

Tribble by Brian Carrier & Joe Grand
- A dedicated PCI card for RAM acquisition, presented in 2004
- http://www.grandideastudio.com/portfolio/index.php?id=1&prod=14
- Still not available for sale :(

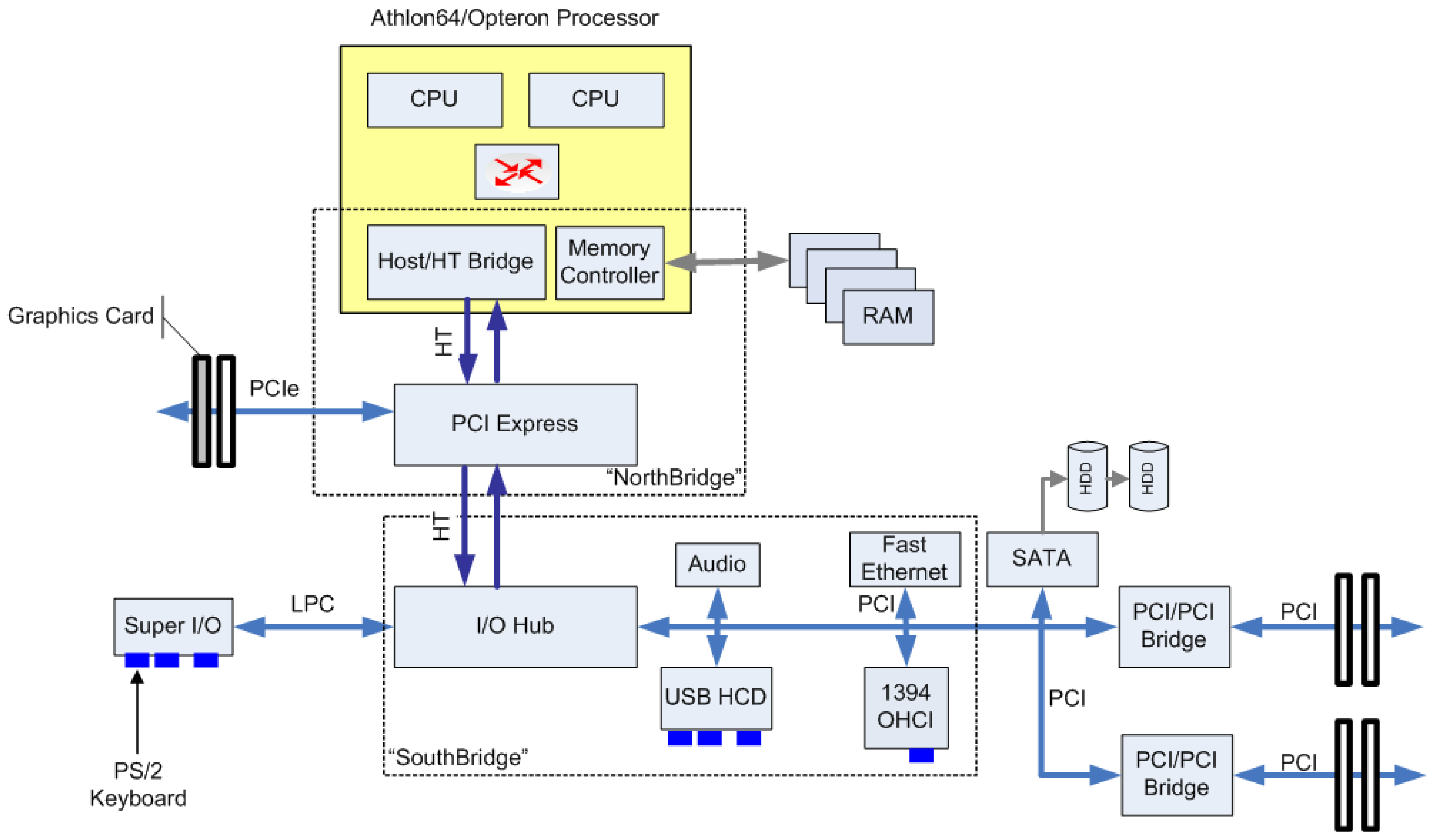
CoPilot by **Komoku**
- A dedicated PCI card – could be used for online system integrity monitoring and for RAM acquisition
- http://komoku.com/technology.shtml
- “not generally available right now“ :(

RAM Capture Tool by **BBN** Technologies
- A dedicated (PCI?) card for RAM acquisition
- http://www.tswg.gov/tswg/about/2005_TSWG_ReviewBook-ForWeb.pdf
- Not available?

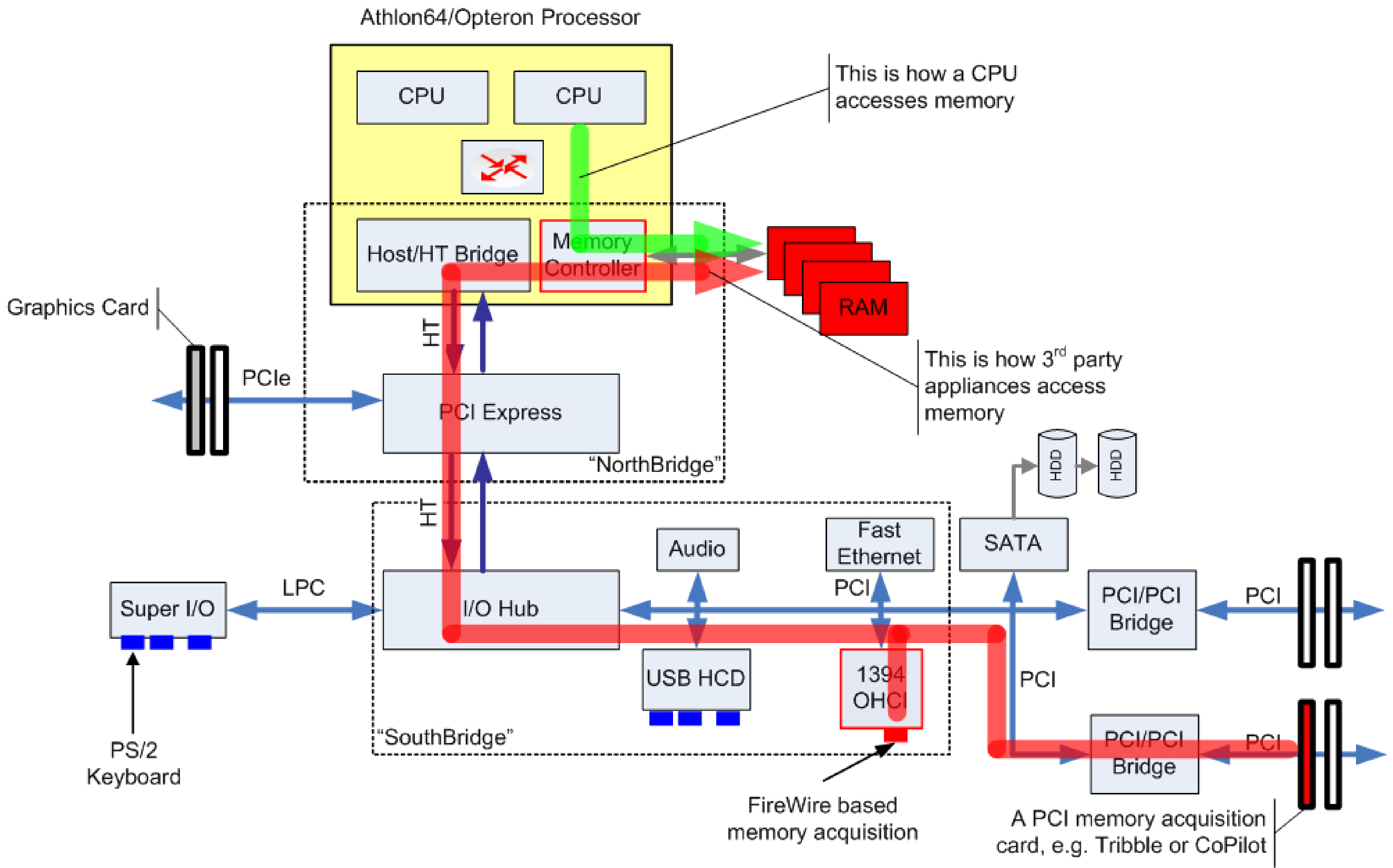
Using **FireWire** bus
- http://cansecwest.com/core05/2005-firewire-cansecwest.pdf
- http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf

How does hardware-based RAM acquisition work?

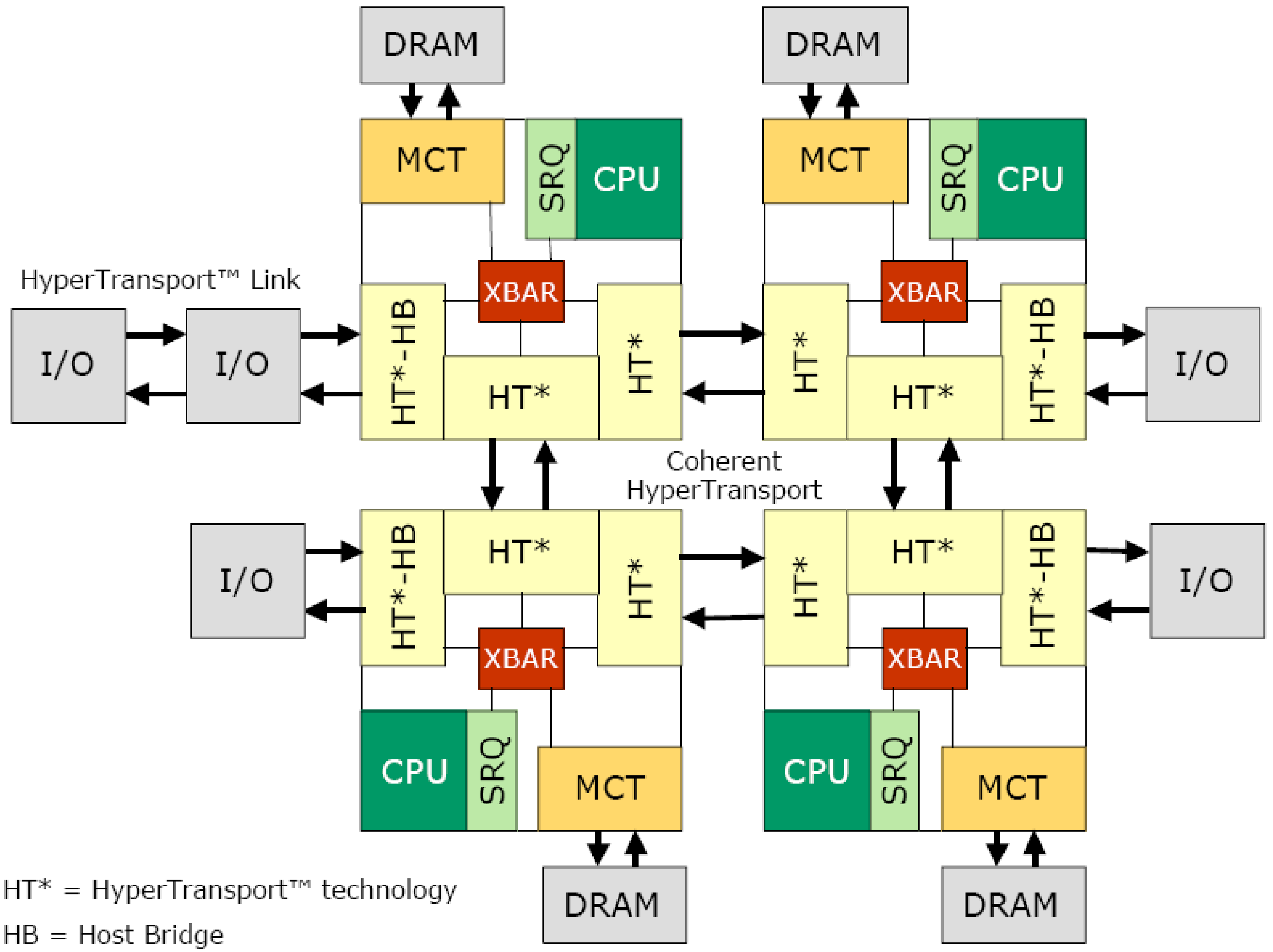
AMD System ex. (Single Processor)



Accessing Physical Memory



Multi Processor Systems (Opteron)



Source: developer.amd.com

So far, so good!

Attacks!

Attacker's goals

"DoS Attack"

- Crash/Halt machine when somebody tries to acquire RAM using DMA
- Can cause huge legal consequences for the investigator

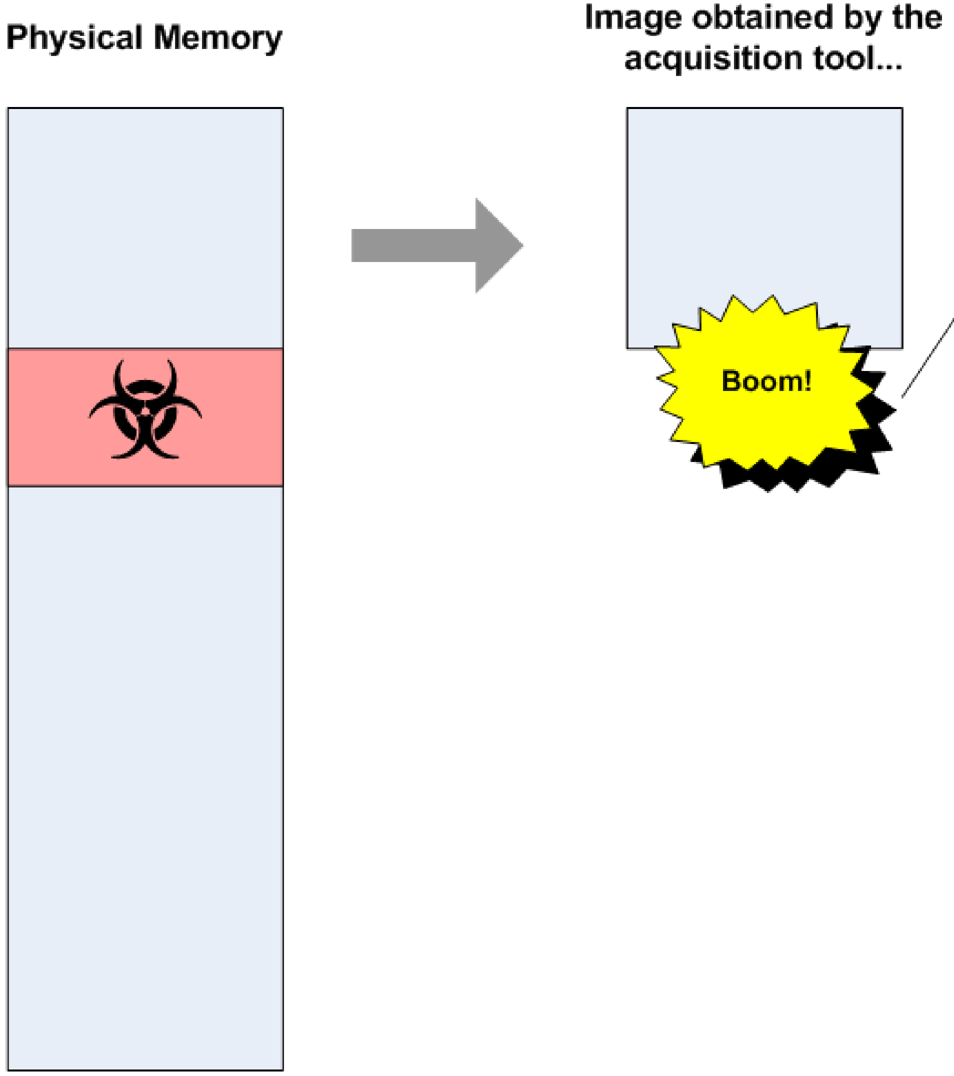
"Covering Attack"

- Acquisition tool can not read some part of physical memory – instead it reads some garbage (e.g. 0x00 bytes).
- CPU sees the real content, which e.g. may contain malicious code and data

"Full Replacing Attack"

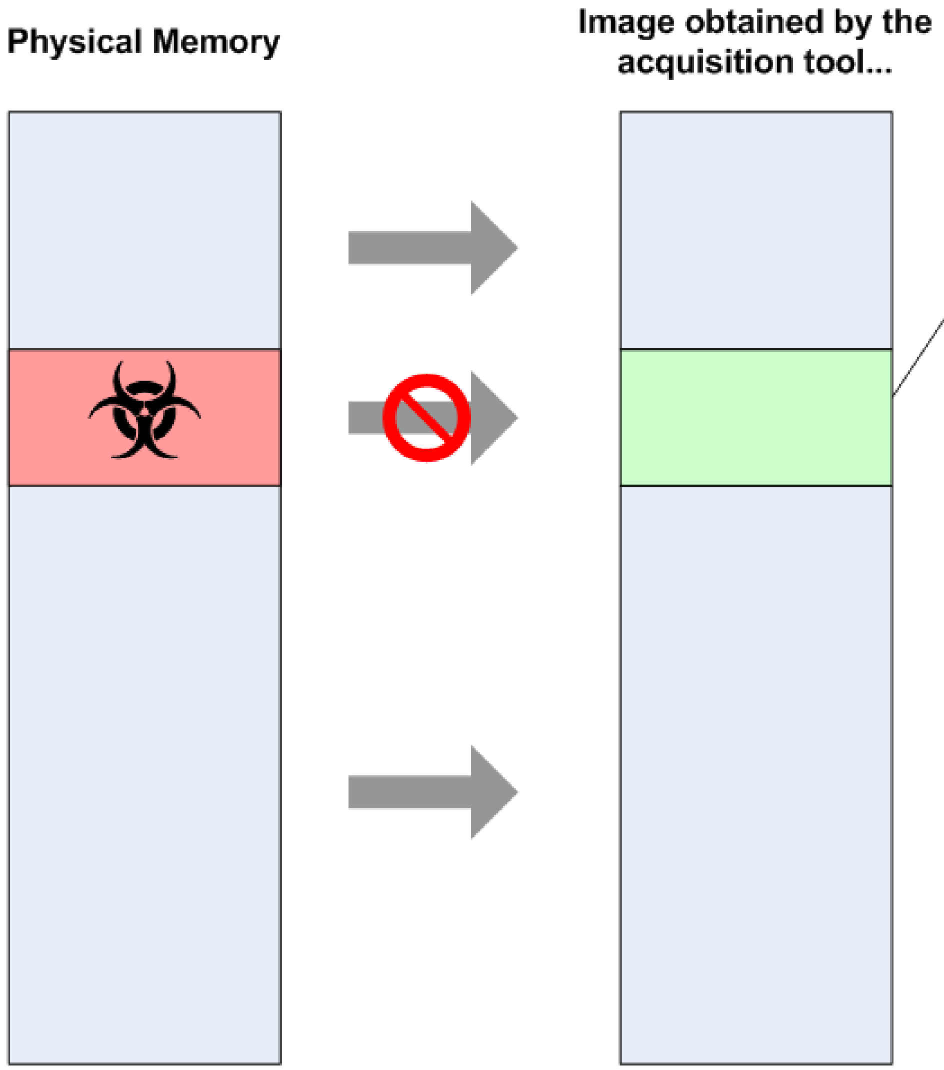
- Like Covering Attack, but the attacker can also provide custom contents (instead of "garbage") for the acquisition tool

DoS Attack Illustration



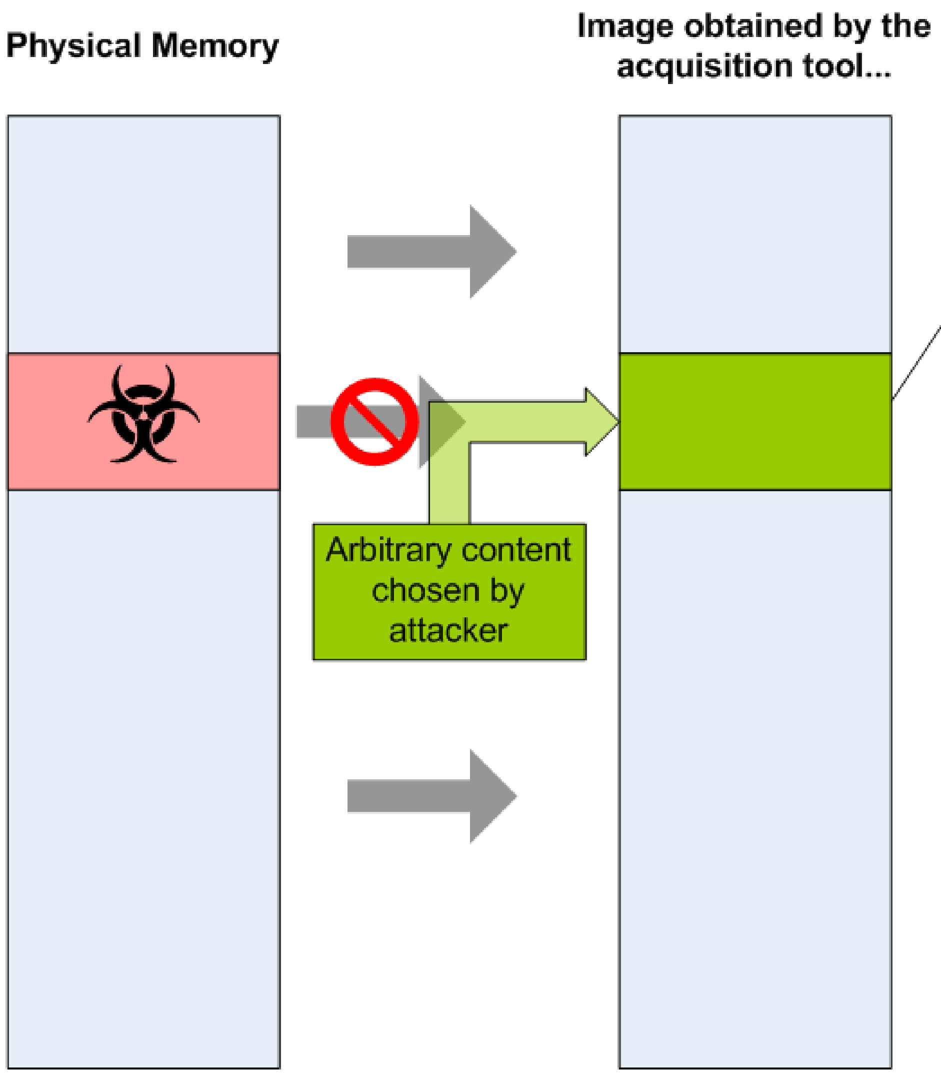
Acquisition tool causes crash or freezes the host OS, when accessing the fragment of memory occupied by malware

Covering Attack Illustration



Acquisition tool gets some "garbage" instead of the real contents of the physical memory. Attacker has *very limited* flexibility to control the contents of the garbage – e.g. all range could be just NULL bytes

Full Replacing Attack Illustration

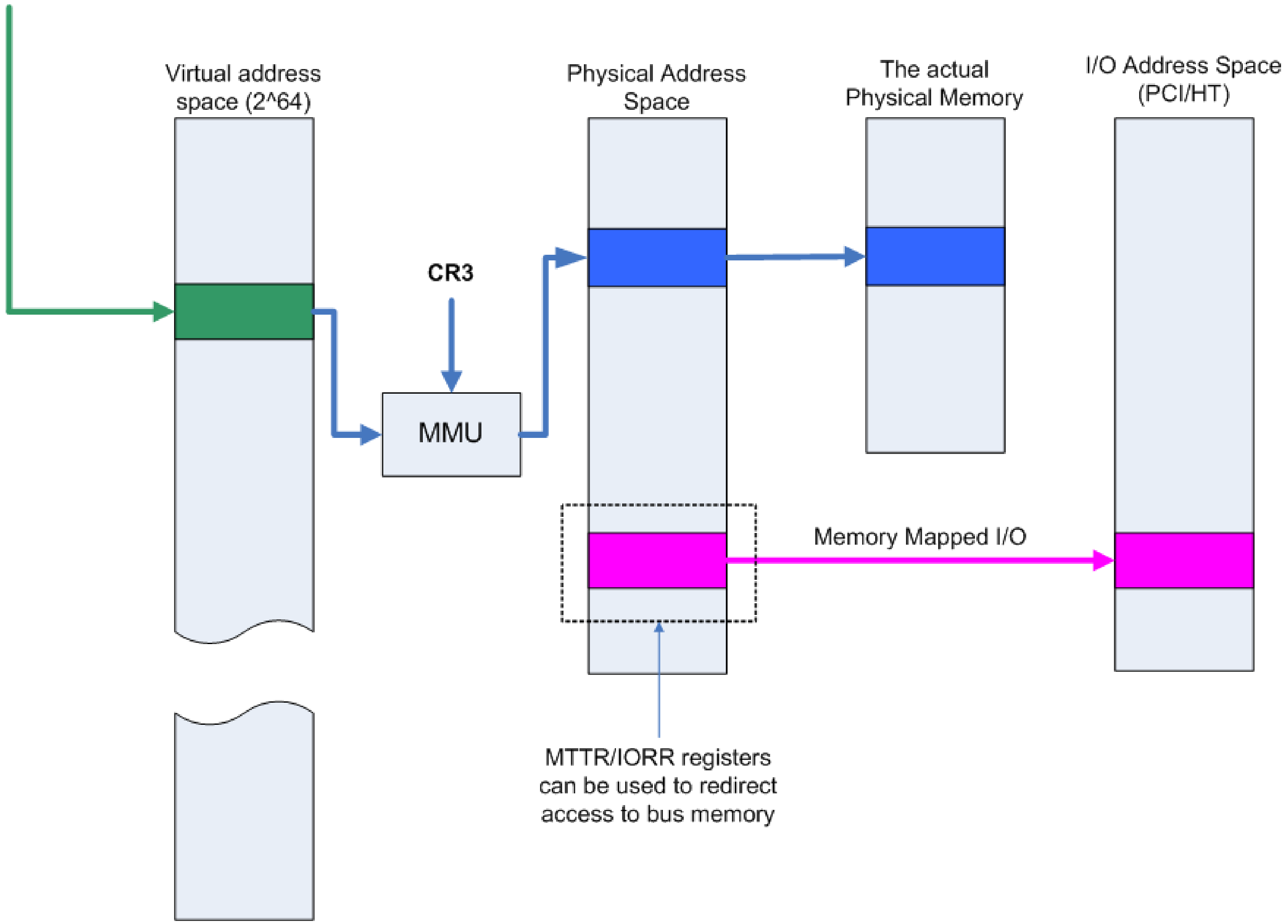

Physical Memory
Image obtained by the acquisition tool...
Arbitrary content chosen by attacker

Attacker can not only hide her malicious code from the acquisition tool, but also can provide arbitrary content to be read by the acquisition tool.

So how do we do this?

Memory Mapped I/O

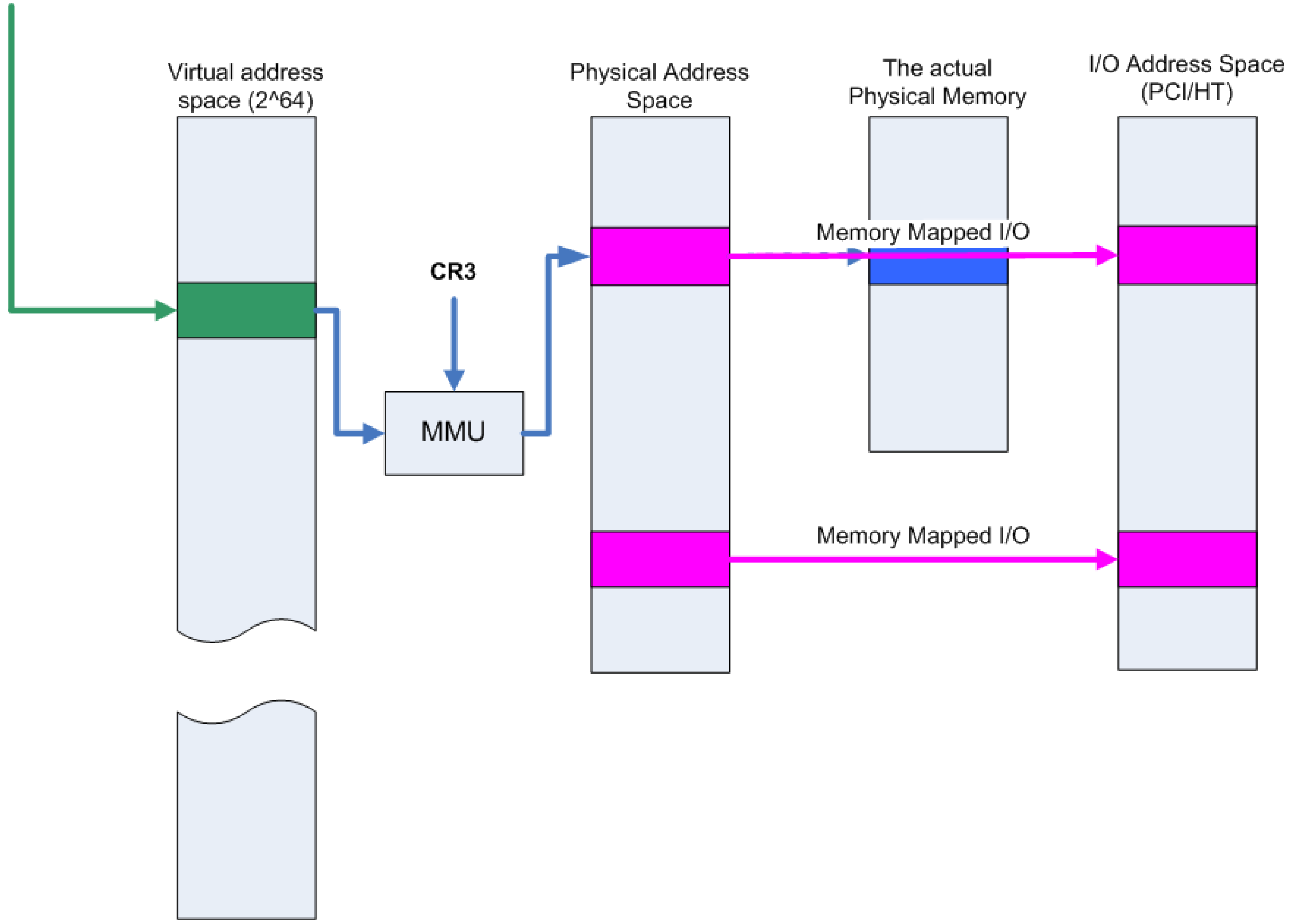
mov eax, [0xfffff80011223344]



Virtual address space (2^64)
CR3
MMU
Physical Address Space
The actual Physical Memory
I/O Address Space (PCI/HT)
Memory Mapped I/O
MTTR/IORR registers can be used to redirect access to bus memory

MMIO cont.

```
mov eax, [0xfffff80011223344]
```


Virtual address space (2^64)
CR3
MMU
Physical Address Space
The actual Physical Memory
I/O Address Space (PCI/HT)
Memory Mapped I/O
Memory Mapped I/O

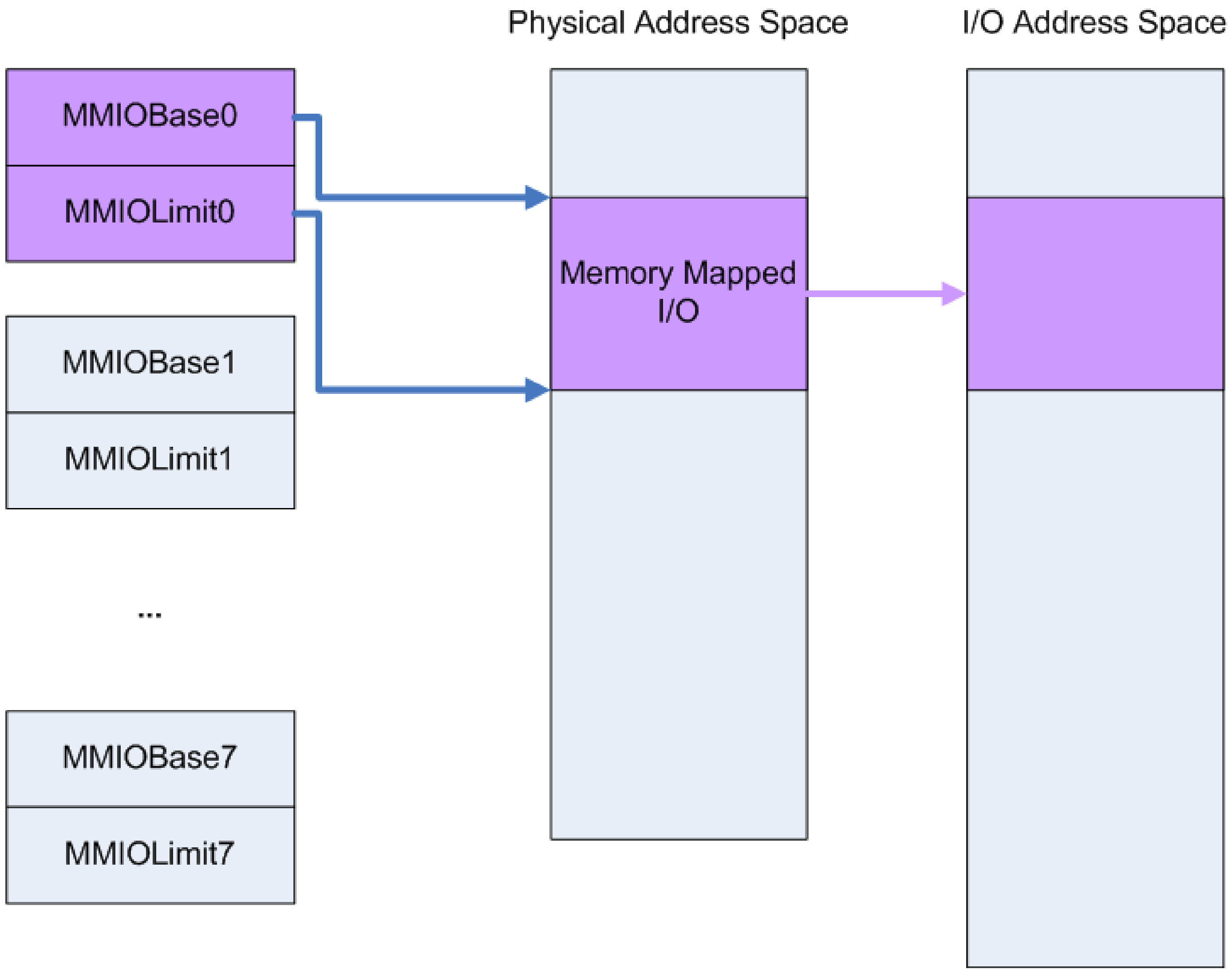
MMIO tricks

- By using MTTR and IORR registers we can assign arbitrary range of physical pages to be mapped into bus address space
- However, this is not what we want, because both processor and bus accesses would be redirected in the same way…
- But keep this in mind…

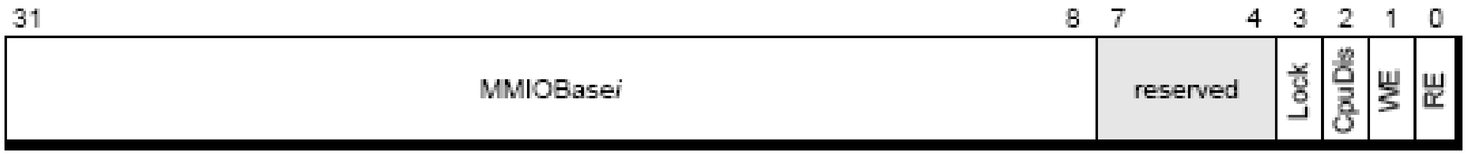
North Bridge’s Memory Map

- MTTR/IORR registers instructs the CPU, for a given physical address, whether to access the system memory or the bus address space (I/O space)
- They have no effect on DMA accesses originating from I/O devices
- DMA accesses are redirected by the Northbridge
- So, there must be some kind of address dispatch table in the Northbridge…

NB's MMIO Address Map

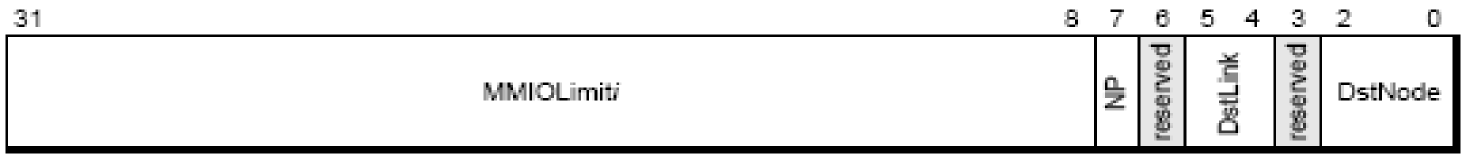


MMIO Map Registers



Bits	Mnemonic	Function	R/W	Reset
31–8	MMIOBase*i*	Memory-Mapped I/O Base Address *i* (39–16)	R/W	X
7–4	reserved		R	0
3	Lock	Lock	R/W	X
2	CpuDis	CPU Disable	R/W	X
1	WE	Write Enable	R/W	0
0	RE	Read Enable	R/W	0

"X" in the Reset column indicates that the field initializes to an undefined state after reset.



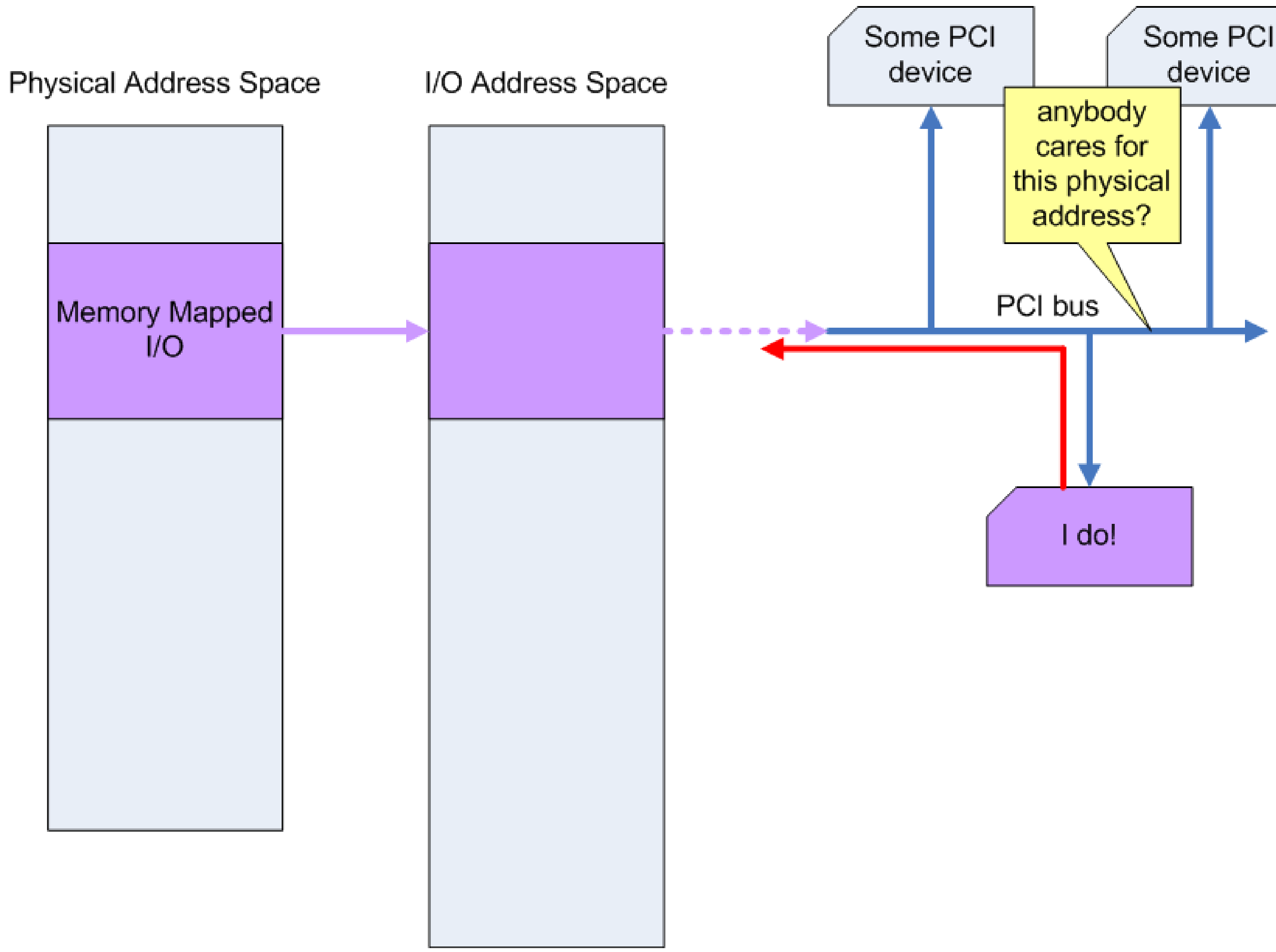
Bits	Mnemonic	Function	R/W	Reset
31–8	MMIOLimit*i*	Memory-Mapped I/O Limit Address *i*	R/W	X
7	NP	Non-Posted	R/W	X
6	reserved		R	0
5–4	DstLink	Destination Link ID	R/W	X
3	reserved		R	0
2–0	DstNode	Destination Node ID	R/W	X

"X" in the Reset column indicates that the field initializes to an undefined state after reset.

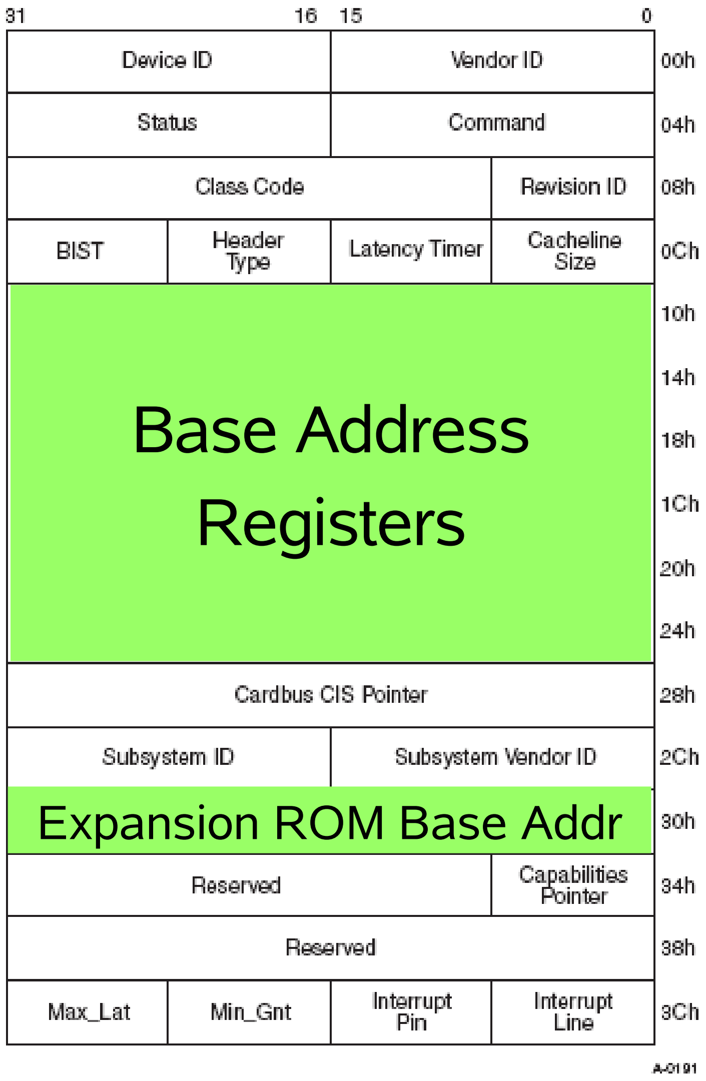
Where these MMIO accesses go?

- Each PCI/HT device can set their *address decoders* to “listen” on particular range of I/O addresses
- So, when Northbridge redirects access to address *pa* to I/O address space, then (hopefully) there will be a device who will respond to read/write request to address *pa*

How MMIOs are handled



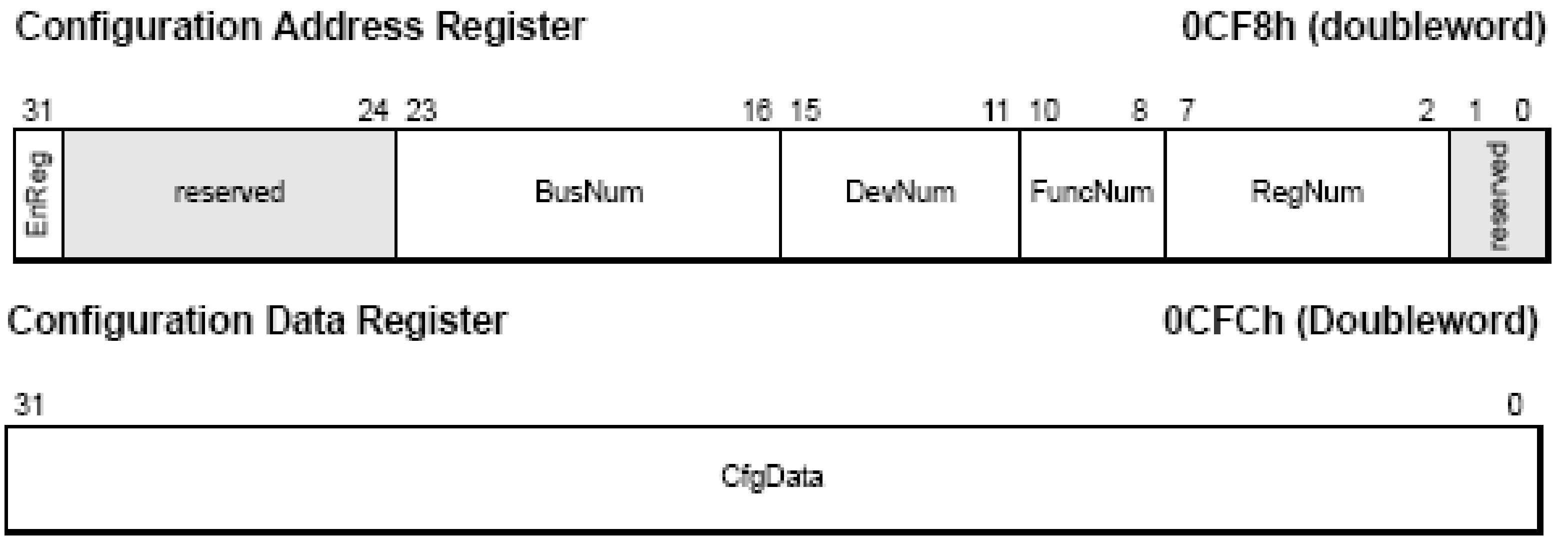
PCI device config space



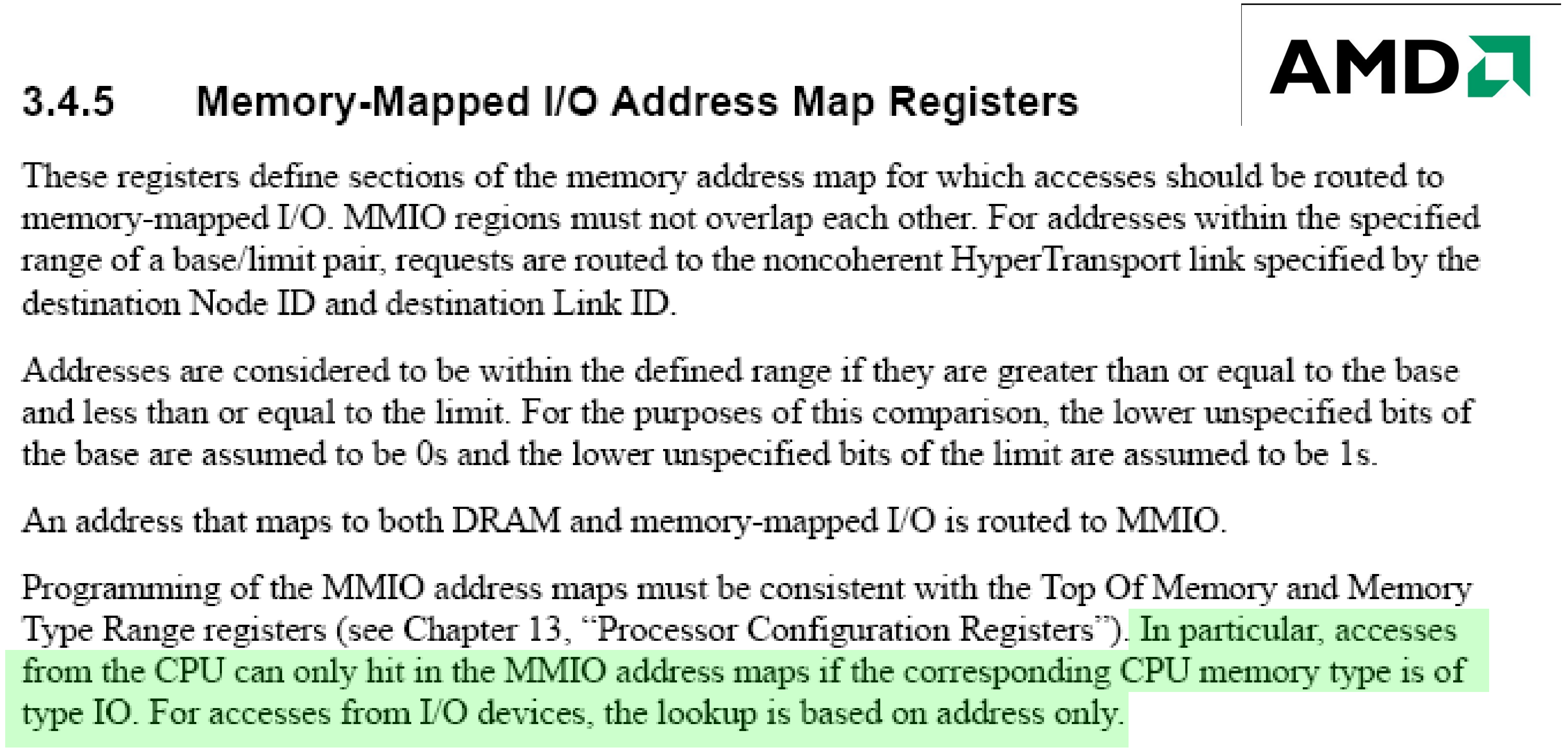
31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cacheline Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Addr				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

Accessing PCI/HT config registers

- Two dedicated I/O ports (to be accessed via IN/OUT instructions):
  - `0xCF8` – selects the address (Bus, Node, Function, Offset)
  - `0xCFC` – data port



An interesting behavior



AMD

3.4.5 Memory-Mapped I/O Address Map Registers

These registers define sections of the memory address map for which accesses should be routed to memory-mapped I/O. MMIO regions must not overlap each other. For addresses within the specified range of a base/limit pair, requests are routed to the noncoherent HyperTransport link specified by the destination Node ID and destination Link ID.

Addresses are considered to be within the defined range if they are greater than or equal to the base and less than or equal to the limit. For the purposes of this comparison, the lower unspecified bits of the base are assumed to be 0s and the lower unspecified bits of the limit are assumed to be 1s.

An address that maps to both DRAM and memory-mapped I/O is routed to MMIO.

Programming of the MMIO address maps must be consistent with the Top Of Memory and Memory Type Range registers (see Chapter 13, "Processor Configuration Registers"). In particular, accesses from the CPU can only hit in the MMIO address maps if the corresponding CPU memory type is of type IO. For accesses from I/O devices, the lookup is based on address only.

BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors (Publication #26094), page *73.*

Athlon/Opteron Northbridge

- Northbridge's Memory Configuration is accessible via HT configuration registers
- HT configuration space is compatible with PCI configuration space
- Each processor has its own Northbridge config space:
  - But all cores share the same one!
- Bus 0, Device 24-31, Functions 0-3
  - Device 24 → Node 0's Northbridge's Config Space
  - Device 31 → Node 7's NB's config space

AMD processors config space

Bus Address: Bus 0, Device 24-31,

- Function 0: HyperTransport™ Technology Configuration
- **Function 1: Address Map ← Yes!**
- Function 2: DRAM Controller
- Function 3: Miscellaneous Control

- So, we're interested in playing with
  - **Bus 0, Dev 24 (-31), Function 1**
  - Within this device, we want to play with Config Registers **MMIOBase** and **MMIOLimit**

Setting up the attack

- We need to add additional entry to processor's NB's memory map
- Let's assume that we would like to cover physical memory starting from address `pa1` until `pa2`
- So, we need to redirect all access from I/O devices to that physical range (`pa1-pa2`) back to I/O…

- First, we need to find *i* (from 0 to 7), so that `MMIOBase[i]` is NULL. This indicates an unused entry in the table…

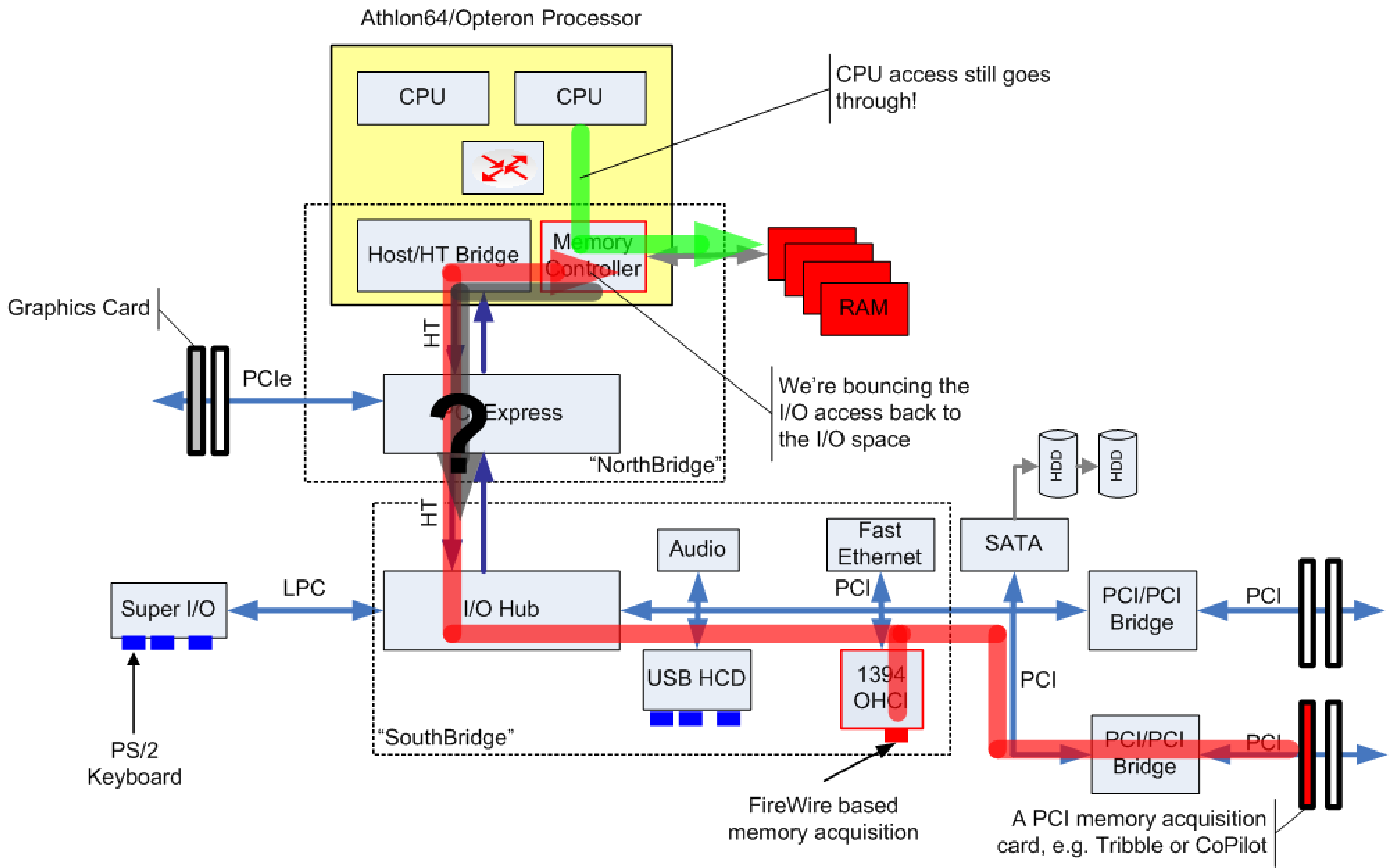
Setting up the attack – cont.

- Now we just need to set:
  - `MMIOBase[i].Base = pa1`
  - `MMIOBase[i].RE = 1`
  - `MMIOLimit[i].limit = pa2`

- And, of course, we do make sure that neither of MTTR/IORR registers marks this very range as MMIO from the CPU point of view

- Now, all accesses to <`pa1, pa2`) from I/O will be redirected back to I/O. While access from CPU will get to the real memory!

I/O Access Bouncing!



Deadlock!

- So, what memory is actually read by the I/O device after we bounce the access back to the H/T bus?
- After all, there is nobody on the HT link or PCI bus to answer the request to read that physical addresses…
- Experiments showed that systems will hang after the acquisition tool will try to read bytes from such a redirected memory!
- This is attack #1: DoS attack!

Getting around the deadlock

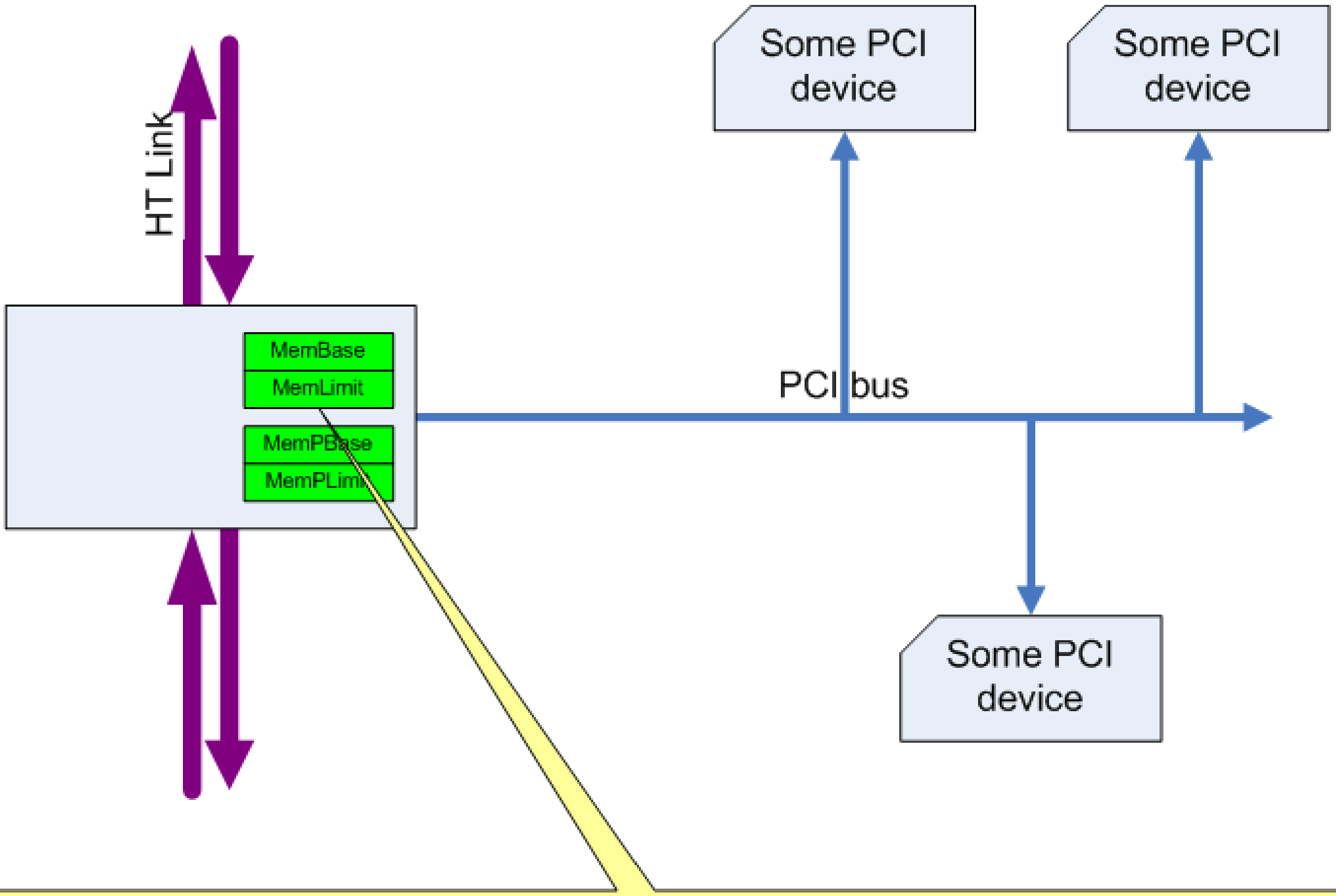
- We need to find a device (on HT link or on PCI bus) that would respond to the read request for our physical address,
- Usually there are many PCI Bridges in modern systems,
- Usually most of them are unused – i.e. no secondary bus is attached,
- We can use such a PCI bridge to be our "responder".

HT Bridge Config Registers

31 … 24	23 … 16	15 … 8	7 … 0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Primary Latency Timer	Cache Line Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	18h
Secondary Status		I/O Limit	I/O Base	1Ch
Memory Limit		Memory Base		20h
Prefetchable Memory Limit		Prefetchable Memory Base		24h
Prefetchable Base Upper 32 Bits				28h
Prefetchable Limit Upper 32 Bits				2Ch
I/O Limit Upper 16 Bits		I/O Base Upper 16 Bits		30h
Reserved			Capabilities Pointer	34h
Expansion ROM Base Address				38h
Bridge Control		Interrupt Pin	Interrupt Line	3Ch

Note: Shaded registers contain minimum-required read-write bits. Other registers are read-only or contain only device-dependent bits.

HT/PCI bridges

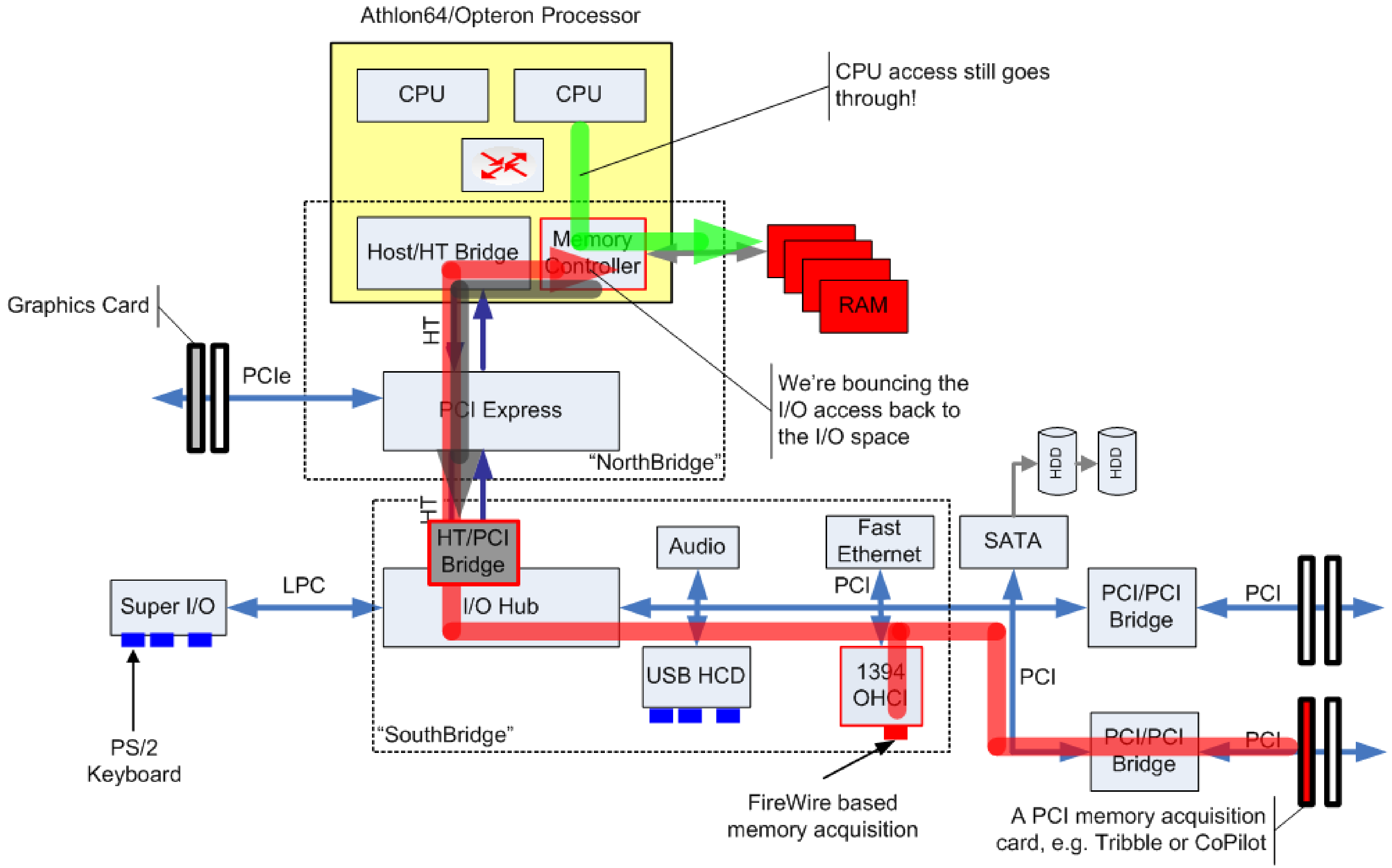


`MemBase` – the lowest address forwarded to the secondary bus
`MemLimit` – the highest address forwarded to the secondary bus
`MemPBase/MemPLimit` – same, but for prefetchable memory

Using a bridge to solve the deadlock

- We need to find unused bridge
  - Usually this is not a problem,
  - Also we might use both Non-Prefetachble and Prefetchable “part” of the bridge – just one of them should be unused.
- Now we do:
  - `Bridge.Mem(P)Base = pa1`
  - `Bridge.Mem(P)Limit = pa2`
- That’s all! :)
- Now the bridge will respond to read access request on an HT link, effectively eliminating the deadlock :)
- Experiments showed that the reading device will get bytes of value 0xff, for each redirected byte…
- This is attack #2: The Covering Attack!

Bouncing Attack with PCI Bridge



Demo!

Full Replacing Attack Discussion

- Using unused device’s RAM
- Using device’s ROM memory
- Using HT remapping capability

FRA: Using devices RAM (?)

- We can remap one of the Base Address Registers of *some* device, so that device thinks that its memory has been mapped starting from `pa1` address…
- Then we need to fill the device's memory with our arbitrary content…
- Now, all access to `pa1` from I/O devices will be redirected back to I/O and will be answered by the device whose memory we've stolen.
- Problem – if the memory is really used for something, we will break the device's functionality
  - E.g. if we used graphics card memory and the card is really used to display some hi-res or 3D graphics…

FRA: Using device's ROM (?)

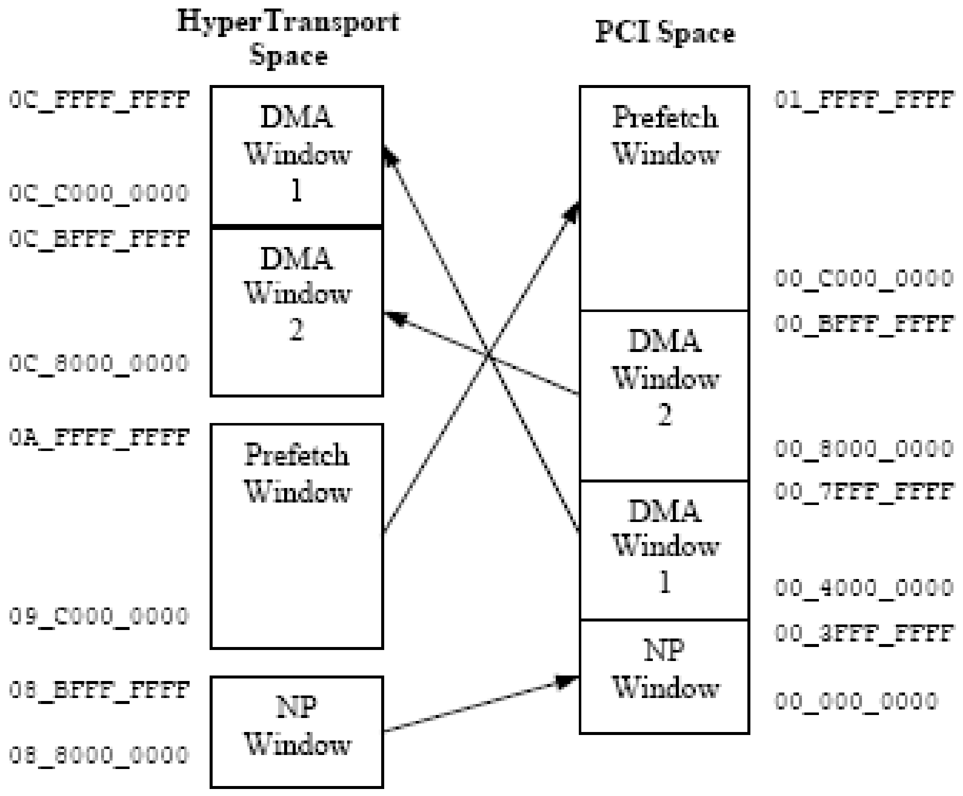
- Expansion ROM is not used after system initialization,
- If the ROM is programmatically re-flashable (EEPROM) we can replace it with our content…
- We then set ROM Base Address to `pa1`
- Then the device will answer to all requests to read `pa1+`
- Problems
  - This is type I infection (and we don't like type I infections!)
  - Most likely will be easily detected when OS uses TPM to verify its booting process…
  - Possible workaround: re-flash back, before rebooting the system… But, not elegant :(

Some Considerations

- Because of the layout of `MMIOBase` and `MMIOLimit` registers both `pa1` and `pa2` should be 64kB aligned,
- That also determines the minimal size of the region to be 64kB at least,
- That means, in order to implement Full Replacing Attack, we need to find a PCI or HT device
  - having at least 64kB of RAM memory
  - having at least 64kB of reflashable ROM
- That should not be a big problem – think about all those graphics cards we have today and that they are often used in servers which run in 80x25 text mode…

FRA: Using HT Remapping capabilities

Some HT bridges may implement Address Remapping Capability, which supports so called “DMA Window Remapping”:



FRA: Using HT Remapping capabilities

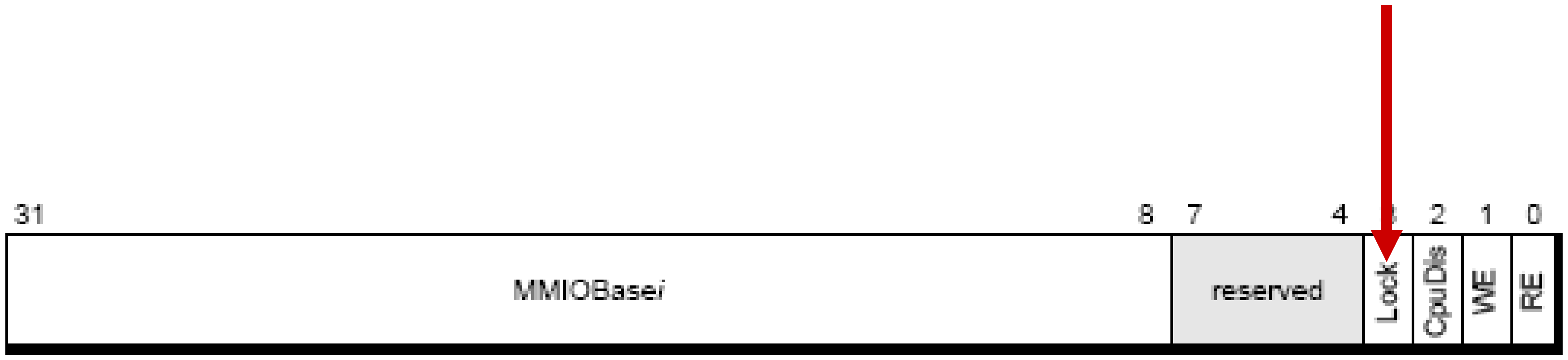
- Problem: there must be at least one such HT bridge in the system which supports this functionality,
- On all authors AMD systems that was not the case,
- However that seems like a very flexible and powerful technique,
- Further research is needed.

Defense?

Defense?

- Maybe a smart PCI device could remove the malicious entry from the Northbridge's map table?
- It's not clear whether PCI device can access Northbridge's config space (i.e. Bus 0, Dev 24-31)?
  - I don't know the answer
- Even if they could…
- …they should not be able to remove the offending entry!
- The "lock bit" is to assure that!

The “Lock” Bit



Bits	Mnemonic	Function	R/W	Reset
31–8	MMIOBase*i*	Memory-Mapped I/O Base Address *i* (39–16)	R/W	X
7–4	reserved		R	0
3	Lock	Lock	R/W	X
2	CpuDis	CPU Disable	R/W	X
1	WE	Write Enable	R/W	0
0	RE	Read Enable	R/W	0

"X" in the Reset column indicates that the field initializes to an undefined state after reset.

Locking the MMIO entry

- If we set the lock bit in MMIO entry, this entry will become read-only!
  - This means, **nobody** will be able to modify it without rebooting the system!
- PCI/HT device can not remove our malicious MMIO entry!
  - even if the device is smart enough to find it!
- It seems then, that:

There is no way to defeat this hack, using a hardware only solution!

Demo

Repercussions

- DoS Attack: investigator who causes system crash/hang might face legal actions for disturbing the work of mission critical servers.
- Covering Attack:
  - Makes it impossible to analyze malware (even though we might find its “hooks” in case of type I and II malware),
  - We can’t learn how it works and in consequence can’t find the “bad guys” behind it :(
- Full Replacing Attack
  - Full stealth even for type I and type II malware
  - Falsify digital evidences → legal consequences

The Near Future: IOMMU

- Arbitrary translations between address space seen by the PCI/HT devices and the physical memory
- Using IOMMU to cheat hardware based acquisition will be trivial
- AMD and Intel are expected to release processors/northbridges fully supporting IOMMU in 2008
  - IOMMU will be part of the hardware virtualization extensions

- say goodbye to hardware based memory acquisition :(

Final notes

- Hardware based memory acquisition was considered as the most reliable way to gather evidence or check system compromises…
- Now, when it has been demonstrated that it is not that reliable as we believed, the question remains:
- **What is the proper method to obtain image of volatile memory?**
  - We live in the 21st century, but apparently can't reliably read memory of our computers!
- Maybe we should rethink the design of our computer systems, so that they were somehow verifiable…

Thank you!

joanna@research.coseinc.com