Modern Platform-Supported Rootkits

Rodrigo Rubira Branco (@bsdaemon) & Gabriel Negreira Barbosa (@gabrielnb)
{ rodrigo.branco || gabriel.negreira.barbosa } @ intel.com

"As the area of our knowledge grows, so too does the perimeter of our ignorance"

Disclaimers

- We don’t speak for our employer. All the opinions and information here are of our responsibility (actually no one ever saw this talk before)
- The talk focus mostly in Intel Architecture. We’ll try to generalize as much as possible though – in that, probably many mistakes might happen, so…
- Interrupt us if you have questions or important comments at any point.
  - IMPORTANT: No, We are not part of the Intel Security Group (McAfee)

Agenda

- Motivation
- Objectives
- Current Rootkits Challenges
- Wrong Assumptions
- AES-NI
- Cache Tricks
- Miscellaneous
  - NVDIMM Surprises
  - ECC-based faults
  - CAT/CMT
  - MPX
- Resources (github link)

Figure Index

- Huahuahuahua, you really expected that?

- Most images were taken from Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes

Motivation

- A quick (but not dirty) history on why we decided to talk about this…

Objectives

- We have two main concerns in regards to computer security:
  - Assumptions
  - Composition

Current Rootkit Challenges

- OS dependency and security mechanisms
  - Different versions of the same OS have different capabilities
  - Windows platforms usually run additional defense mechanisms (such as AV). *nix-based platforms change frequently (difficult to cross-work on multiple systems)
- The computer is an amazing platform, but also has lots of differences between different models
  - Difficult to work across the board
  - Some functionalities remain the same for many years though
  - Most security tools don’t expect things to work in a different way than

    the usual expected platform-provided features

Widely Known Rootkit Capabilities

- Connect-back to bypass perimeter outgoing filters
- Keyl
- Update mechanism (for deploying new functionalites)
- Pivoting mechanism (for attacking additional machines within the perimeter)

We will NOT discuss any of them ;) Instead, we will abuse the platform to hide and protect our code

Platform Capabilities Covered

- Generally available architectural features (ECC, Caches, IVT, DMA)
- Modern architectural features (AES-NI, MPX, NVDIMM, CAT, CMT)
- Some tips on other architectures (along the way, nothing specific)

IDT/IVT – What we will see next

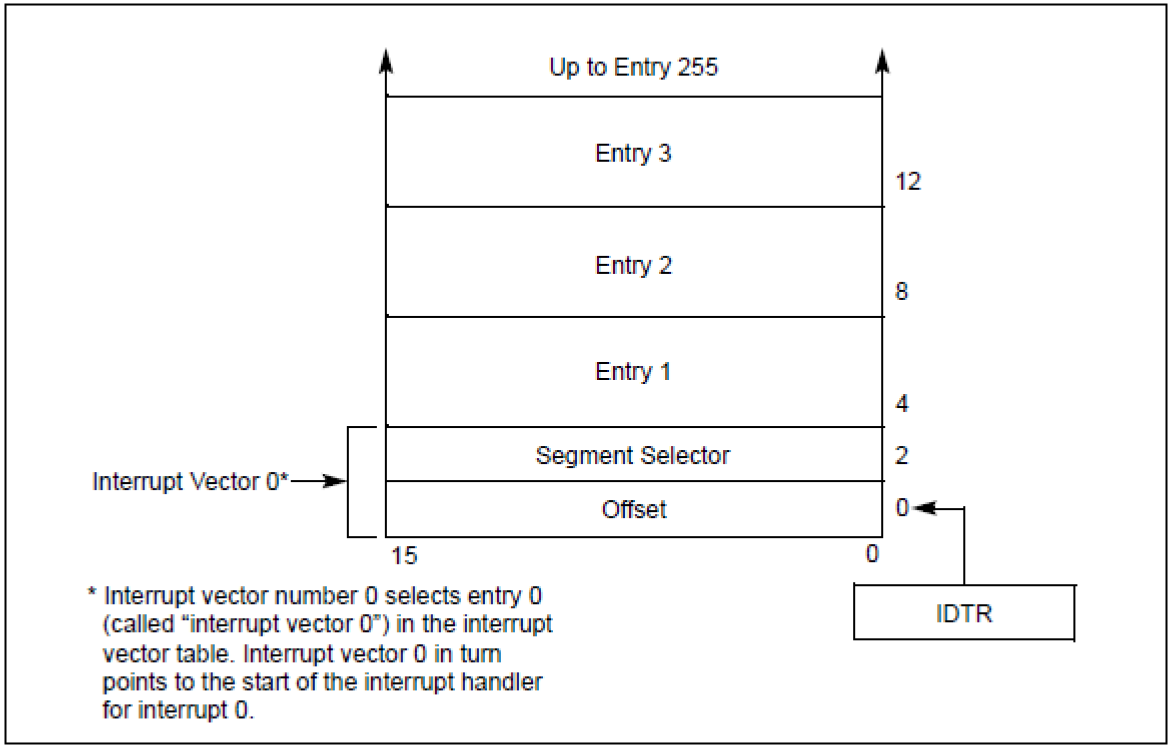
- Quick Introduction
- Related Abuses:
  - Assumption: Fixed location for IDTR target?
  - Assumption: Always protected mode on dumps?
  - Assumption: 32-bits only malware? (ahn??)

IVT Introduction

- Real-mode counterpart of the Interrupt Descriptor Table of Protected Mode

- Let’s visualize the differences…

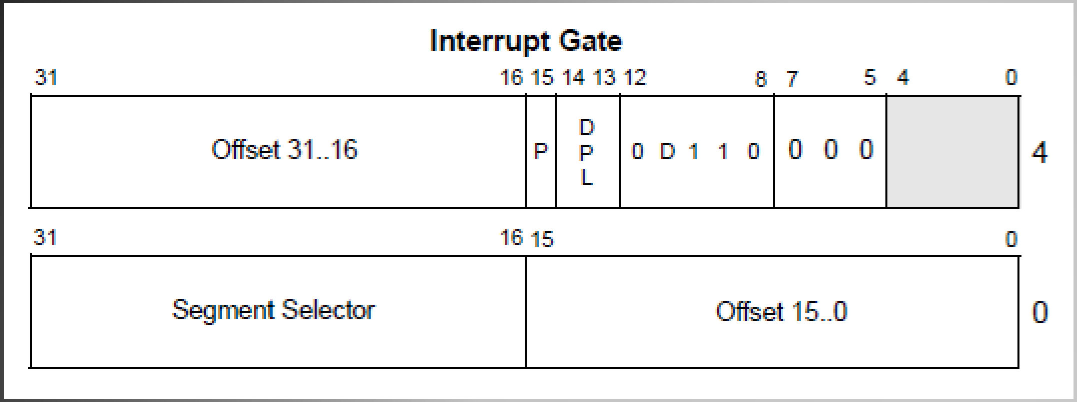
IVT Introduction (2)

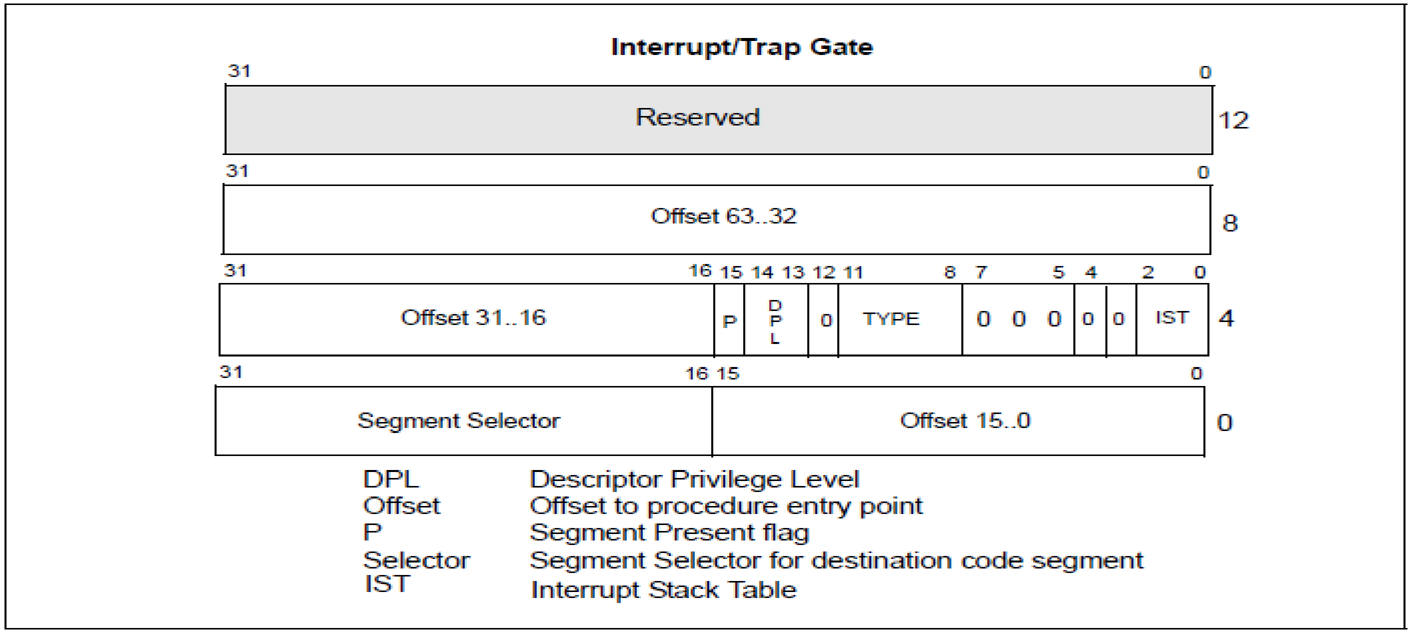


* Interrupt vector number 0 selects entry 0 (called "interrupt vector 0") in the interrupt vector table. Interrupt vector 0 in turn points to the start of the interrupt handler for interrupt 0.

Figure 20-2. Interrupt Vector Table in Real-Address Mode

IDT 32 and 64 bits





DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Figure 6-7. 64-Bit IDT Gate Descriptors

IVT Abuse – idt.py (Volatility) Where is the mode check?

```
if not addr_space is None and not symtab is None:
      KPCR = 0xFFDFF000

      if not addr_space.is_valid_address(KPCR):
         print "KPCR is UNAVAILABLE"
         return

      idt_mem = read_value(addr_space, 'unsigned long', KPCR+0x38)

      print 'IDT#\tISR\tunused_lo\tsegment_type\tsystem_segment_flag\tDPL\tP'

      for i in range((
         offset = (i) *

         LowOffset = read_value(addr_space, 'unsigned short', idt_mem+0 + offset)
         selector  = read_value(addr_space, 'unsigned short', idt_mem+2 + offset)
         unused_lo = read_value(addr_space, 'unsigned char', idt_mem+4 + offset)
         options   = read_value(addr_space, 'unsigned char', idt_mem+5 + offset)

         segment_type = options & 0x0f
         system_segment_flag = (options & 0x10) >> 4
         DPL = (options & 0x60) >> 5
         P = (options & 0x80) >> 7
         HiOffset = read_value(addr_space, 'unsigned short', idt_mem+6 + offset)

         print '%s\t%4.4x:%4.4x%4.4x\t%x' % (str(i), selector, HiOffset, LowOffset, unused_lo),
         print '\t%2.2x\t%x\t%x\t%x' % (segment_type, system_segment_flag, DPL, P)
```

If a malware changes IDTR but do not update the Windows structure, it will still hook calls, but will be 'invisible' to Volatility

Bonus Abuse? check_idt.py (Volatility) Where is the mode check?

```
# hw handlers + system call
    check_idxs = list(range(0, 20)) + [128]

    if self.profile.metadata.get('memory_model', '32bit') == "32bit":
        idt_type = "desc_struct"
    else:
        idt_type = "gate_struct64"

    # this is written as a list b/c there are supposdly kernels with per-CPU IDTs
    # but I haven't found one yet...
    addrs = [self.addr_space.profile.get_symbol("idt_table")]

    for

        table = obj.Object(theType = "Array", offset = tableaddr, vm = self.addr_space, targetType = idt_type, count = tblsz)

        for i in check_idxs:

            ent = table[i]

            if not ent:
                continue

            idt_addr = ent.Address

            if idt_addr != 0:
                if not idt_addr in sym_addrs:
                    hooked = 1
                    sym_name = "HOOKED"
                else:
                    hooked = 0
                    sym_name = self.profile.get_symbol_by_address("kernel", idt_addr)

                yield(i, idt_addr, sym_name, hooked)
```

If you hook a function, but export the target’s function symbol, it is all fine?

And just for fun? Btw, also does not check mode…

- https://code.google.com/p/volatility/source/browse/trunk/volatility/plugins/malware/idt.py

“# Currently we only support x86. The x64 does still have a GDT

    # but hooking is prohibited and results in bugcheck. “

Disclaimer 2

- We appreciate the work the guys did in Volatility, it is an interesting and useful tool
- The idea of our point is to clarify that assumptions are everywhere and we need to understand what we use so we can have a better grasp on the limitations

AES-NI – What we will see next

- AES-NI Quick Introduction
- AES-NI Abuse:
  - D the
    i

Everyone read the PoC || GTFO released in this conference, right??

Should we skip this part?

AES-NI Introduction

- Intel® Advanced Encryption Standard New Instructions (AES-NI)

“**What is it?**

- Intel® AES-NI is a new encryption instruction set that improves on the Advanced Encryption Standard (AES) algorithm and accelerates the encryption of data”.

AES-NI Locking and Enabling

MSR 0x13C

Bit	Description
0	Lock bit (always unlocked on boot time, BIOS sets it)
1	Not defined by default, 1 will disable AES-NI
2-32	Reserved

The Code

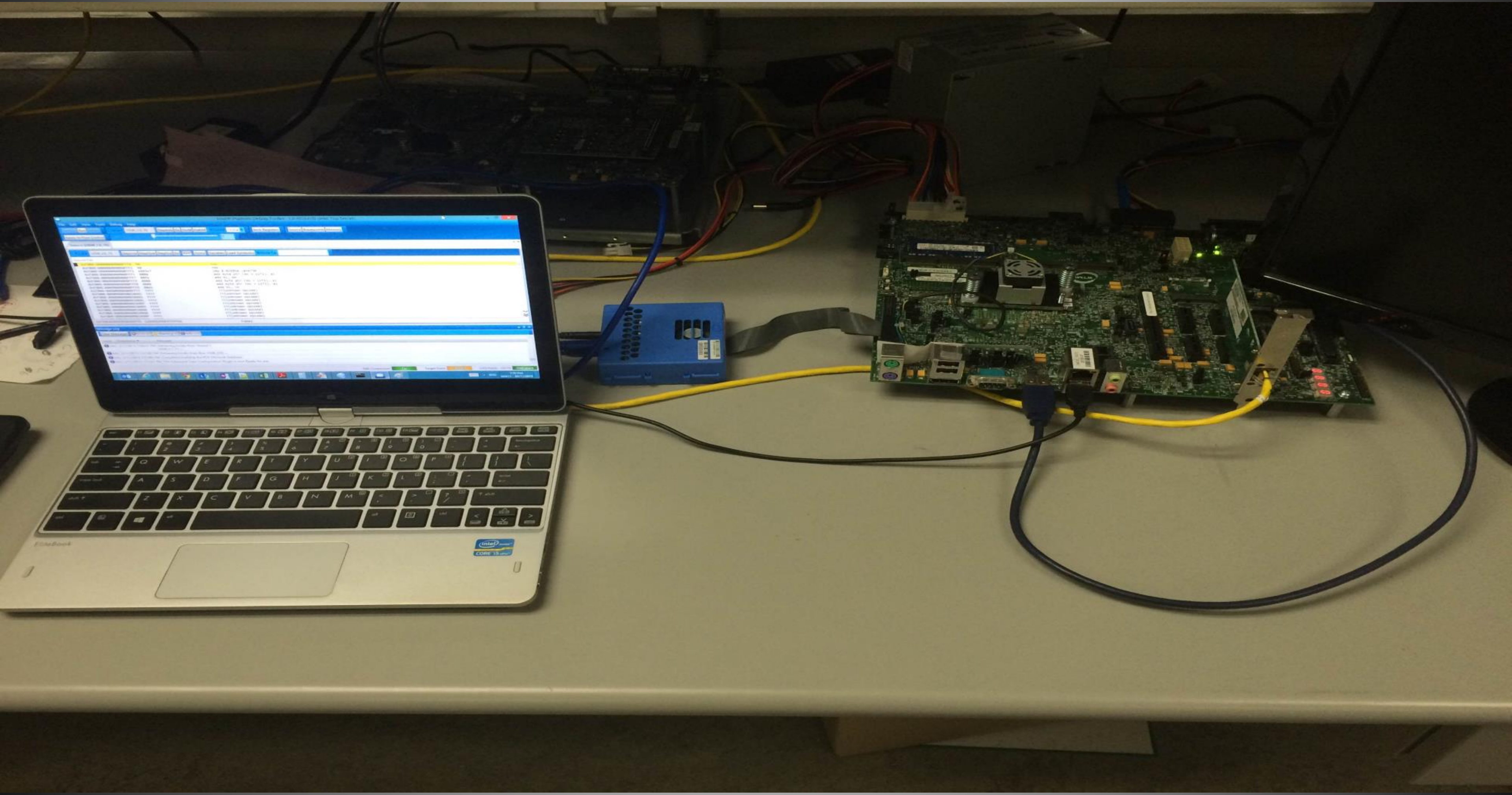
- Please, see at your magazine ;)
- There is a chipsec check we created for you to check your system (also in the article and soon on chipsec github): http://github.com/chipsec/chipsec
- We add a handle to the #UD and we set the MSR to disable AES-NI (we don’t let BIOS set the lock bit, so we can modify it at runtime)

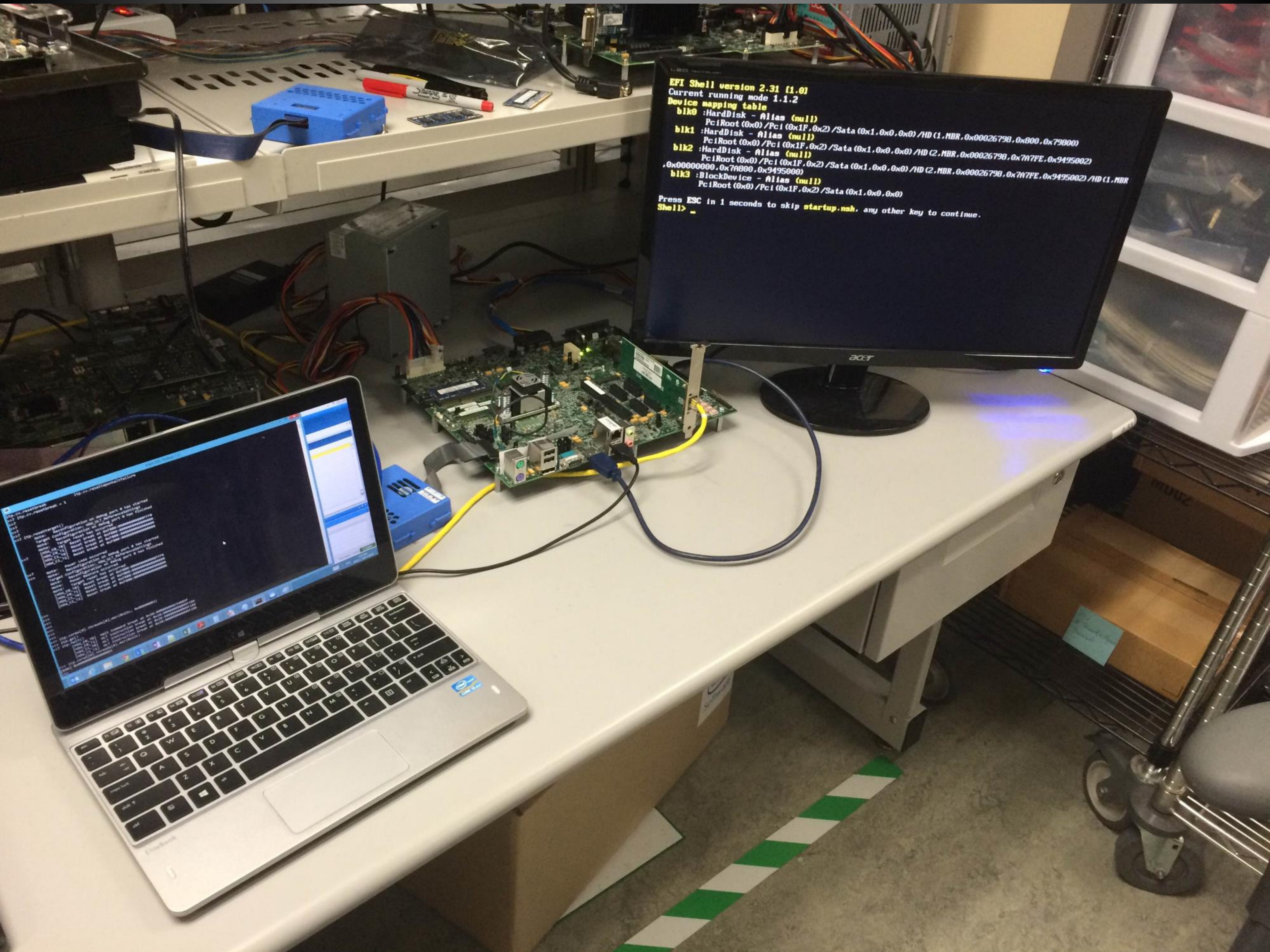
AES-NI Disable for other purposes

- Useful anti-disassembly mechanism (kernel driver + malware implementation)
- One can obviously use this idea to analyze a malware that counts on AES-NI

• The BIOS part…

JTAG-based debugger





```
EFI Shell version 2.31 [1.0]
Current running mode 1.1.2
Device mapping table
  blk0 :HardDisk - Alias (null)
        PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x1,0x0,0x0)/HD(1,MBR,0x00026798,0x800,0x79800)
  blk1 :HardDisk - Alias (null)
        PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x1,0x0,0x0)/HD(2,MBR,0x00026798,0x7A7FE,0x9495002)
  blk2 :HardDisk - Alias (null)
        PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x1,0x0,0x0)/HD(2,MBR,0x00026798,0x7A7FE,0x9495002)/HD(1,MBR
,0x00000000,0x7A800,0x9495000)
  blk3 :BlockDevice - Alias (null)
        PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x1,0x0,0x0)

Press ESC in 1 seconds to skip startup.nsh, any other key to continue.
Shell> _
```

BIOS Analysis?

We start stopping the machine execution at the BIOS entrypoint.   We defined some functions to be used thru our code, such as:

```
get_eip():  Get the current RIP
get_cs():      Get the current CS
get_ecx():  Get the current value of RCX
get_opcode():  Get the current opcode (disassembly the current instruction)
find_wrmsr():  Uses the get_opcode() to compare with the '300f' (wrmsr opcode) and
        return True if found (False if not)
search_wrmsr():
        while find_wrmsr() == False:  step() -> go to the next instruction (single-step)
find_aes():
        while True:
                step()
                search_wrmsr()
                if get_ecx() == '0000013c':
                        print "Found AES MSR"
                        break
```

BIOS Analysis Tip

- All BIOS we know sets its General Protection Fault (#GP) handler to just resume the execution
  - Thus, if you set any MSR locks after giving the MSR the value you like, the BIOS will try to set it, will receive a #GP that it ignores and just continue happily

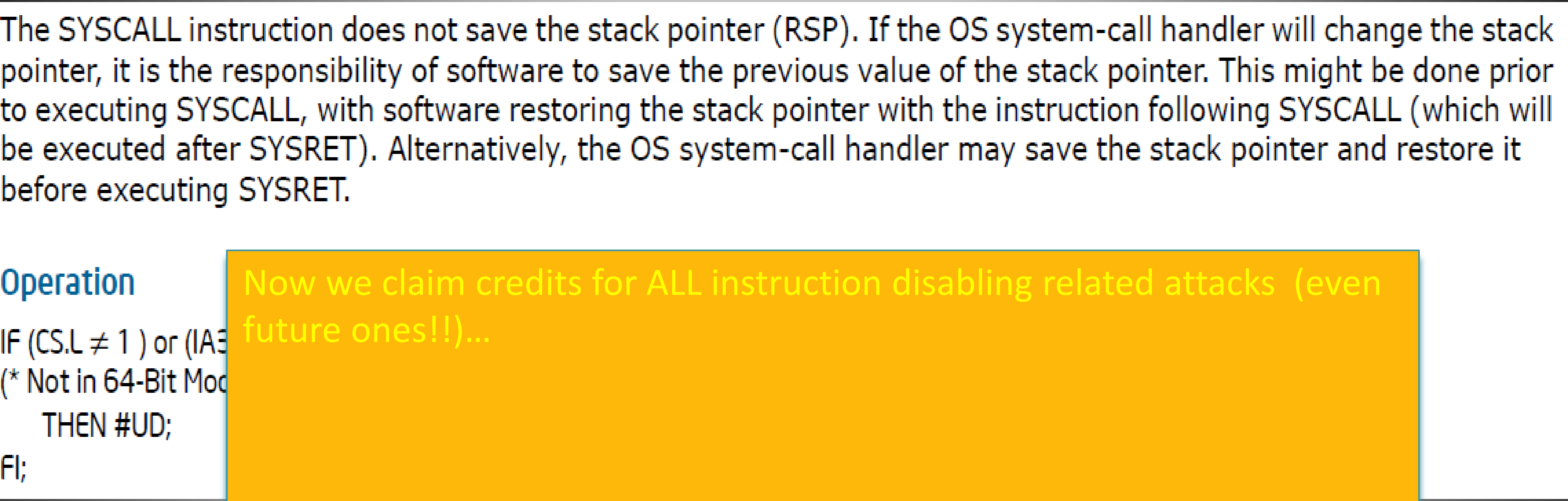
Talking about disabled instructions

- As we just saw, some instructions can be disabled…
- Next Abuse:
  - Disabling the fast system call handler in 64 bit machines to create a stealth syscall hook (is it ok to say hooker?)

Changing subjects – check_syscall.py (Volatility)

```
memory_model = self.addr_space.profile.metadata.get('memory_model', '32bit')



    if memory_model == '32bit':

        mode = distorm3.Decode32Bits

        func = "sysenter_do_call"

    else:

        mode = distorm3.Decode64Bits

        func = "system_call_fastpath"
```

We got inspired... Thanks!



The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

Operation

```
IF (CS.L ≠ 1 ) or (IA3
(* Not in 64-Bit Moc
    THEN #UD;
FI;
```

Now we claim credits for ALL instruction disabling related attacks (even future ones!!)...

huahuahuAHUAhuAhuahuaU

IF the rootkit, in a 64-bit Linux machine:

- Unset: IA32_EFER.SCE
- Hook the #UD handler
- On the #UD exception, disassemble the instruction: if syscall
- Hook as wanted ☺ - New way to hook system calls on 64 bit Machines (should work on Windows too)

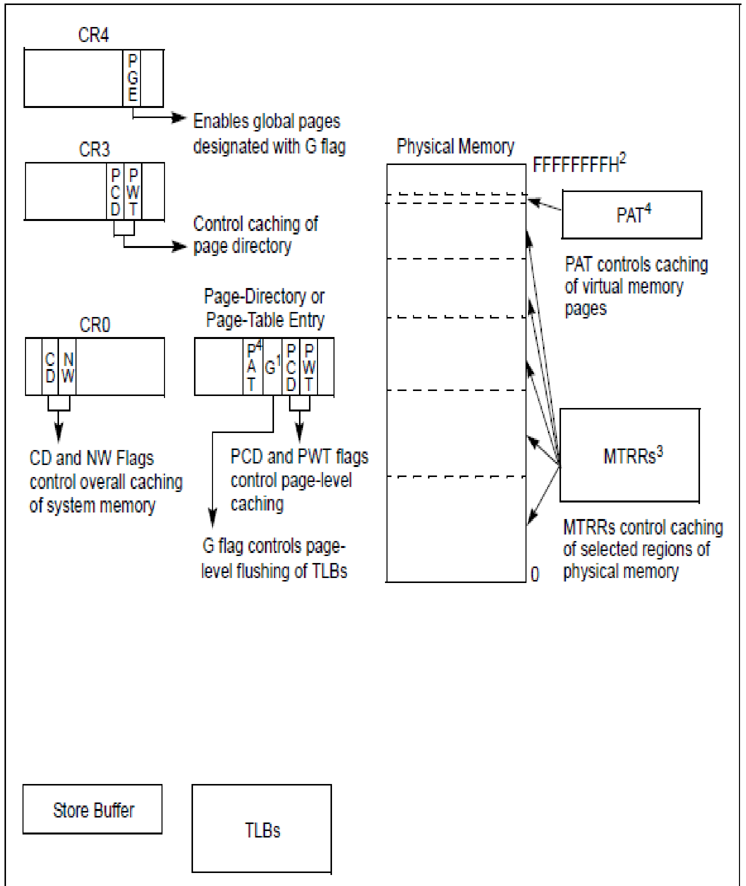
Cache – What we will see next

- Cache Quick Introduction
- Cache-related Abuse:
  - Forcing async between cache and memory

Quick Introduction

- Write-back
- Cache Hit, Cache Miss
- Cache Eviction, Cache Fill

Cache Control


CR4
PGE
Enables global pages designated with G flag
CR3
PCD PWT
Control caching of page directory
CR0
CD NW
CD and NW Flags control overall caching of system memory
Page-Directory or Page-Table Entry
PAT4 G1 PCD PWT
PCD and PWT flags control page-level caching
G flag controls page-level flushing of TLBs
Physical Memory
FFFFFFFFH2
0
PAT4
PAT controls caching of virtual memory pages
MTRRs3
MTRRs control caching of selected regions of physical memory
Store Buffer
TLBs

1. G flag only available in P6 and later processor families
2. The maximum physical address size is reported by CPUID leaf function 80000008H. The maximum physical address size of FFFFFFFFFH applies only If 36-bit physical addressing is used.
3. MTRRs available only in P6 and later processor families; similar control available in Pentium processor with the KEN# and WB/WT# pins.
4. PAT available only in Pentium III and later processor families.
5. L3 in processors based on Intel NetBurst microarchitecture can be disabled using IA32_MISC_ENABLE MSR.

Abusing Cache

- Hide malicious code in disabled cache
  - L3 disabling is possible in netburst (up to July 2006 accordingly to Wikipedia) microarchitecture (and all caches can be disabled in Atom cores)
  - When L3 is enabled, malicious code is executed. When is disabled, malicious code is hidden
  - Causes performance loss
  - Life was easy ☺

Cache Async

- “FrozenCache – Mitigating Cold-Boot Attacks For Software-Based Full-Disk Encryption” presentation, 27C3 (2011)
- A big challenge: How to prove the cache async? The author discuss some ideas (like performing the cold-boot attack as a test), but had not figured out a way to do from software
- Some proposed using DMA…

DMA x Cache

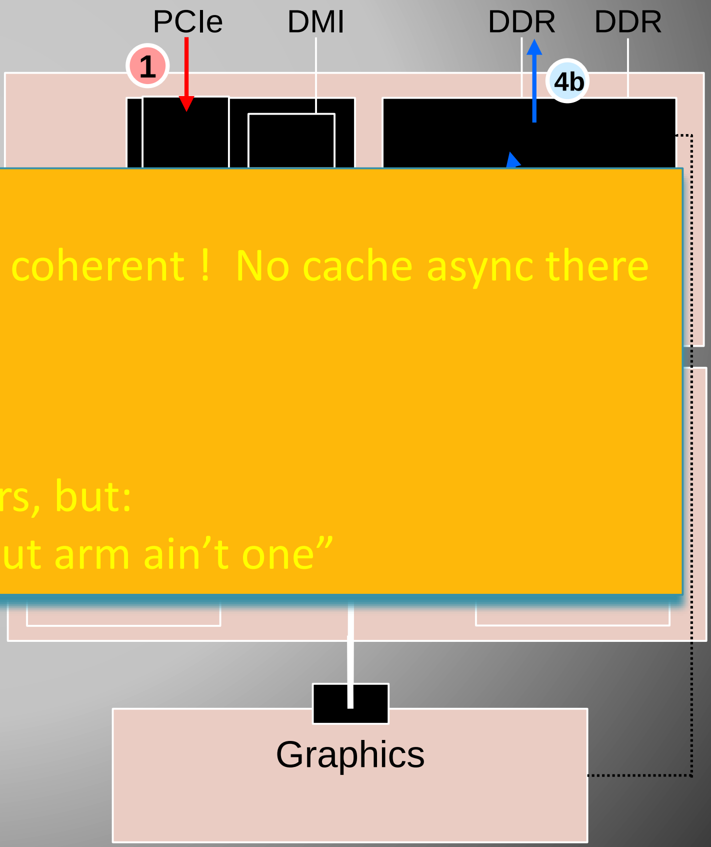
1. IO device issues a DMA request on the PCIe port
2. The PCI CPU Agent forwards the request to the DMA arbitrator

A

B

3. The DMA Arbitr
request to the C
4. If cache hit, dat
sent to the DMA

5. DMA Arbitrat
completes the



Conclusion:
- DMA accesses in Intel Platform is coherent ! No cache async there

Conclusion 2:
- We don't want to point our fingers, but:
  "We have x86 problems, but arm ain't one"

Cache Async – Proof Introducing Concepts

- Page-walking mechanism

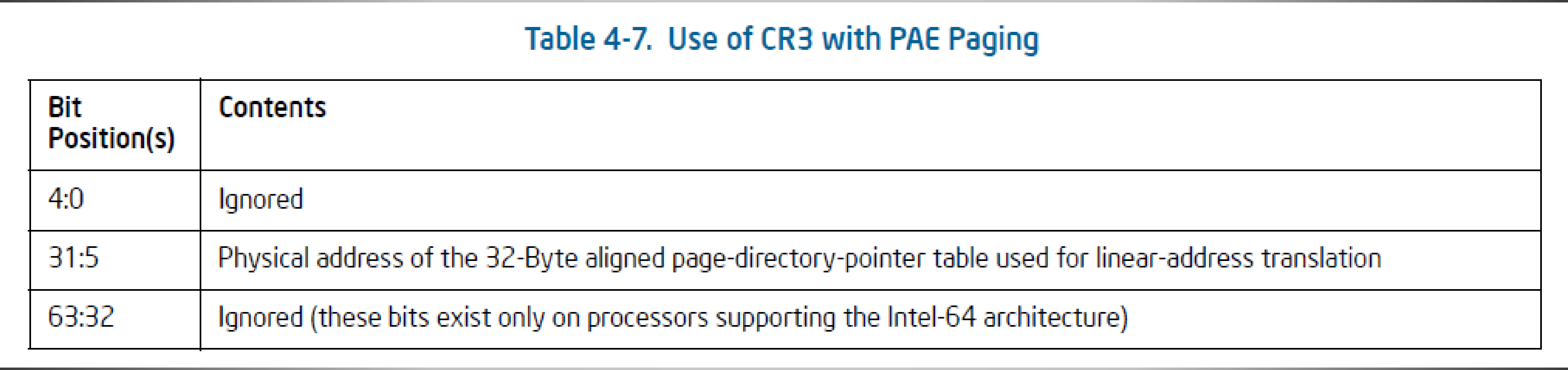


Table 4-7. Use of CR3 with PAE Paging

Bit Position(s)	Contents
4:0	Ignored
31:5	Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

Cache Async – Proof
Introducing Concepts

- Page Directory Pointer Table

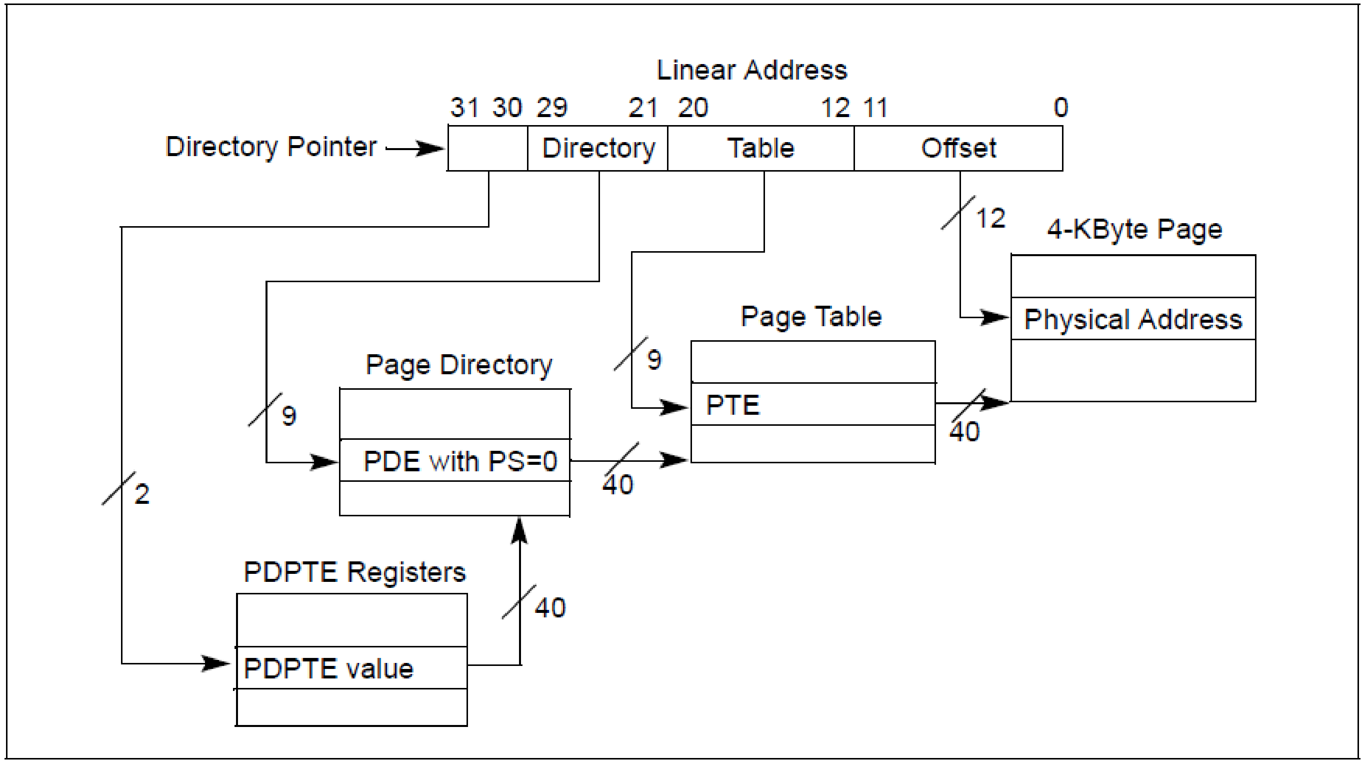


Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

PDPTE Structure

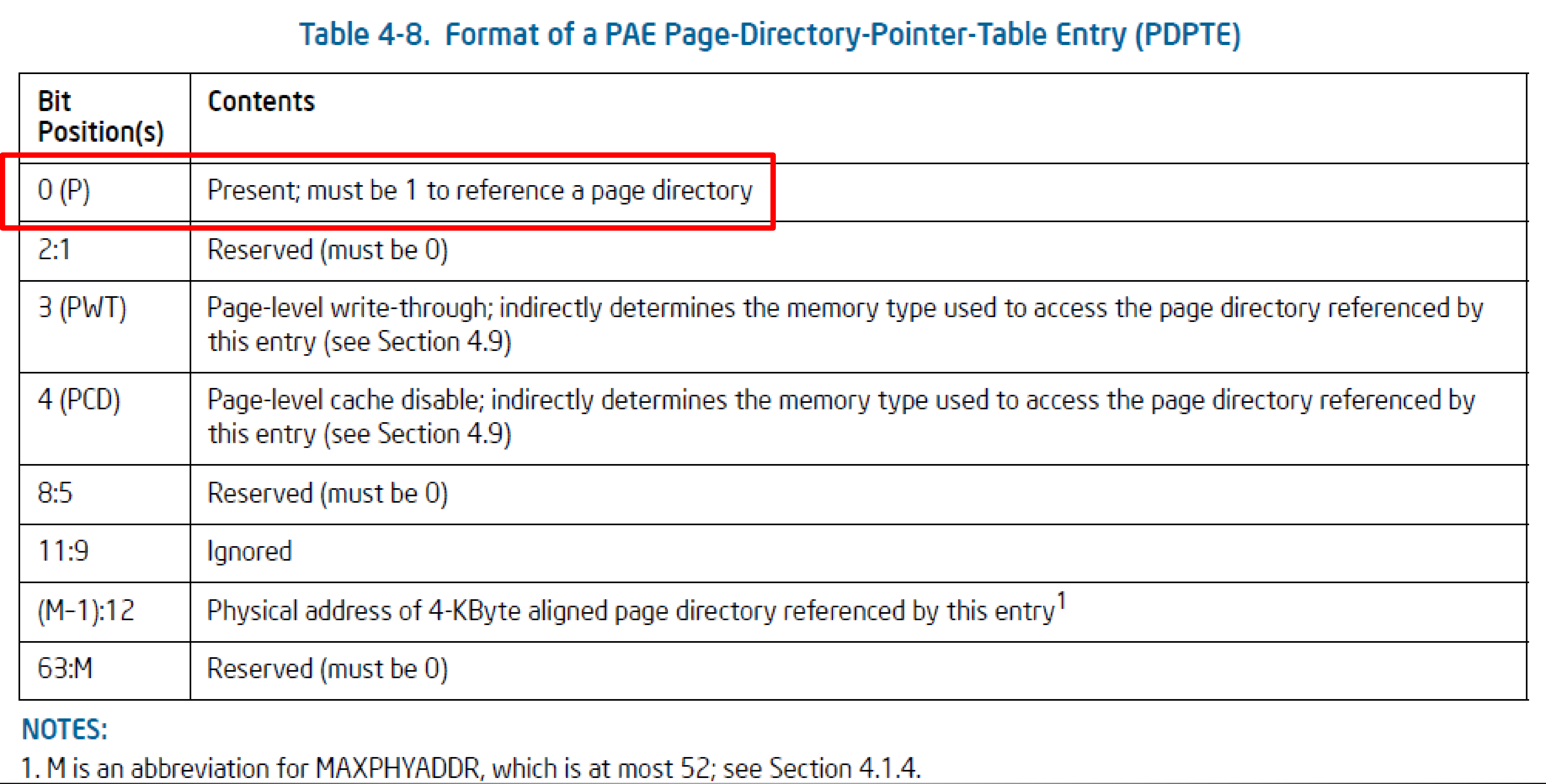


Table 4-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

NOTES:

1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

Theory – We prove it next

- Page-walk hardware does not use cache when PAE is enabled
  - Not part of the theory, but JUST FYI: If it did, we could actually create nice attacks with the async of cache and memory of the cr3 pointed memory

- If we prove that, it is also proven that we can force a situation where cache and memory are not sync'ed

Proofs
Reminder: PoCs on github

- Given the way that pagewalking works (PDPTE, cr3, etc)
- Without interrupts and with only one thread we:
  - We make sure that the PDPTEs are marked as write-back
  - Invalidate the cache (wbinvd)
  - We access the PDPTE entries, thus forcing it to the cache
  - We change the present bit (not present) – given the cache hit, it changed in the cache
  - We do random memory accesses, using virtual memory (forcing pagewalking to occur) – not enough accesses to fill the cache (otherwise, eviction would occur)
    - The system did not crash, thus pagewalking does not use cache when PAE is enabled

Now, if you don't want the data anymore
Reminder: PoCs on github

- Just add a 'invd'
  - It will not write back memory

- Proof:
  - The same steps as before, adding the *invd* in the end, before returning (the system will not crash later, even after 'eviction' – there will be no eviction)

Small consideration on FrozenCache

- In the blog (http://frozencache.blogspot.com/) he uses the ordering:

“

3-) Flush the cache (thus truly overwriting the encryption key in RAM)
wbinvd

4-) Add

5 and 6 should be inverted (no-fill mode will not let cache to be filled)

It probably worked for him, because wbinvd takes time to complete (more on It later) and thus, he had a cache hit, which will update the cache entry

5-) Disa

movl %
orl 0x4
movl %eax, %cr0

6-) Write the encryption key from the CPU registers to RAM (data remains in the cache, doesn't get written to memory)

movq %xmm0, [X]
movq %xmm1, [X+8]
movq %xmm2, [X+16]
movq %xmm3, [X+24]”

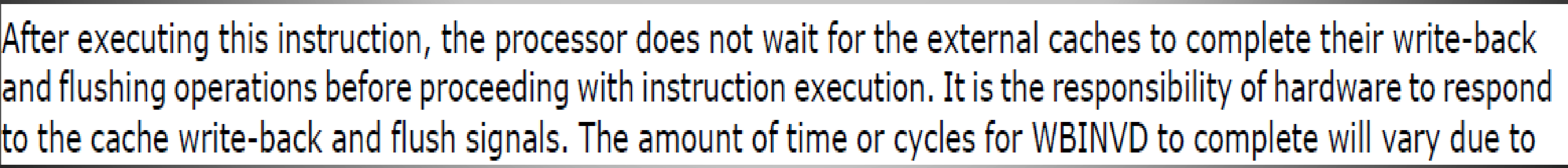
Comments on ‘no-fill’ mode

- If you are in no-fill mode, once you exit, it will write-back to memory
- ‘invd’ does not invalidate the cache entries while in the no-fill mode
  - We mentioned it was performing a write-back, but it was happening because of the exit from no-fill mode
- Interesting Coreboot file to take a look as well (src/cpu/intel/haswell/cache_as_ram.inc)

Challenges

- Besides 'someone' commenting out our assembly function called in the C code and we spending like 3 hours (total) analyzing something that was not even called (twice)…

- wbinvd


After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to

Miscellaneous – What we will see next Just sharing some other ideas…

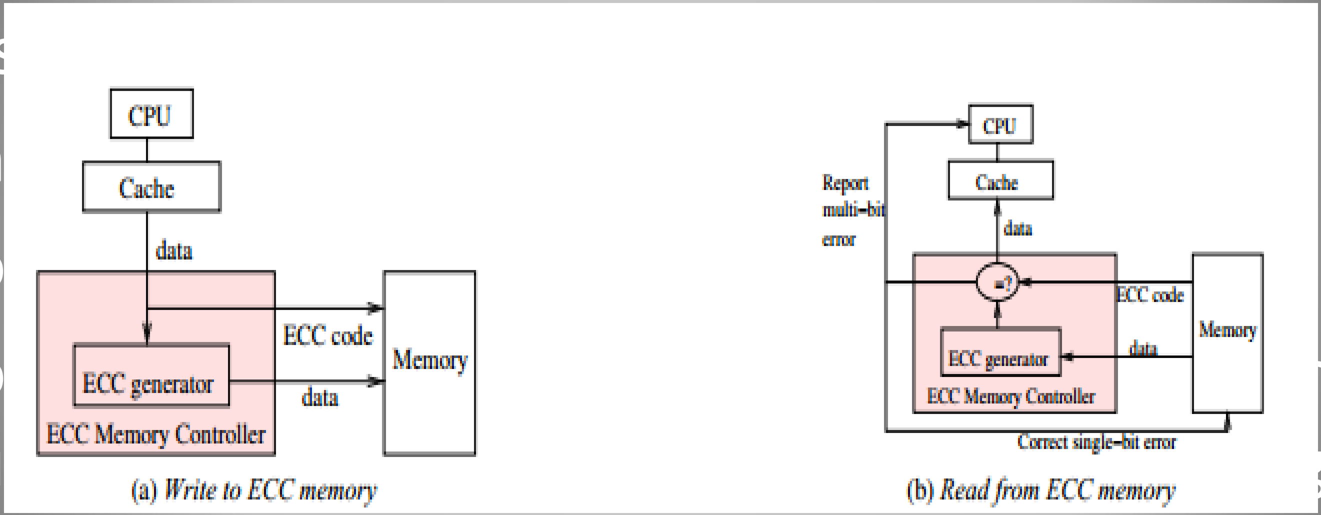
- ECC
- NVDIMM
- CAT/CMT
- MPX

ECC - Introduction

- ECC (Error Checking & Correction) memory was introduced to detect (and correct) common kinds of internal data corruption in data storage (memory, flash drives, etc)
- ECC-capable memory controllers (workstation and servers) are able to detect and correct errors of a single bit per 64-bit word or only detect in two bits per 64-bit word

ECC – Introduction (2)

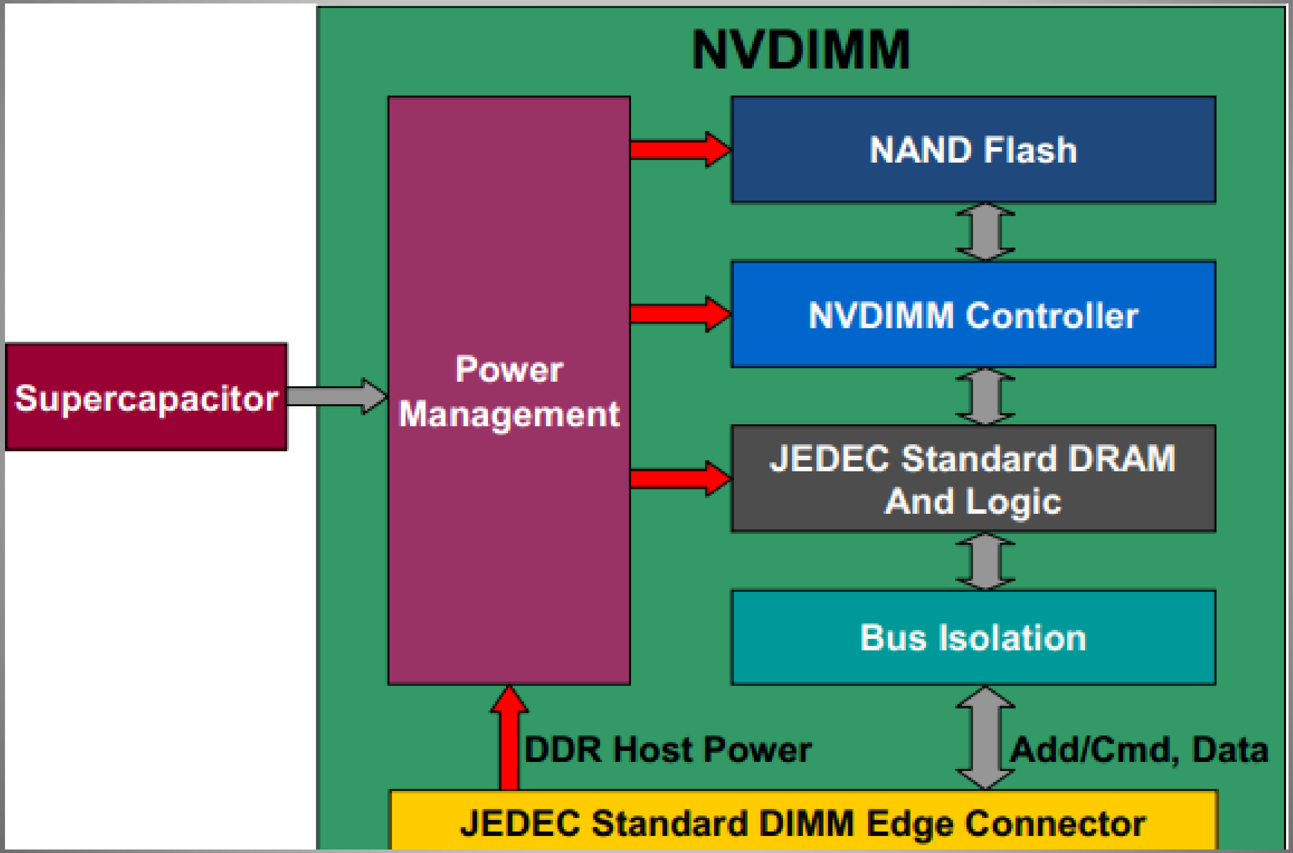
- Most controllers support four modes:
  - Dis
  - Ch
  - Co
  - Co ors, bu g errors)



(a) *Write to ECC memory* (b) *Read from ECC memory*

Source: SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs

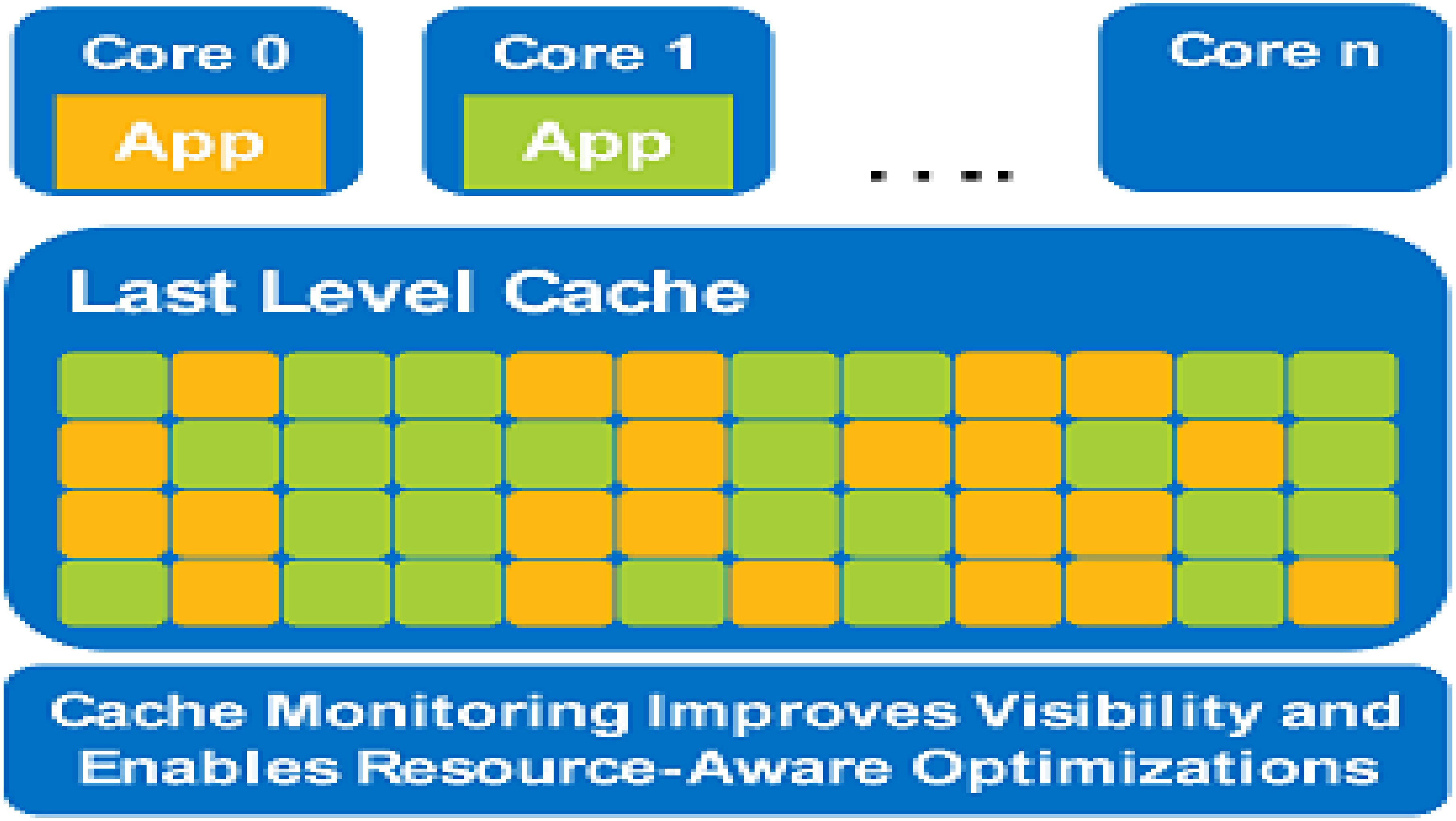
NVDIMM


NVDIMM
NAND Flash
NVDIMM Controller
Supercapacitor
Power Management
JEDEC Standard DRAM And Logic
Bus Isolation
DDR Host Power
Add/Cmd, Data
JEDEC Standard DIMM Edge Connector

NVDIMM Surprises

- How does it affect platform attestation??
- How does it affect software disk-encryption technologies???

New Intel Cache Technologies - CMT


Core 0
App
Core 1
App
....
Core n
Last Level Cache
Cache Monitoring Improves Visibility and Enables Resource-Aware Optimizations

Cache Coloring x Cache Partitioning

- Cache Coloring is a software-based way to implement cache partitioning:
  - You map logical addresses to physical addresses that will create a split in the cache lines between different processes
  - Used by RTOS OSes to implement partitioning in hardware that do not support it

New Intel Cache Technologies

- CAT – Cache Allocation Technology
  - Intended to enforce a cache QoS
  - Does that through Dynamic Cache Partitioning based on a mask that is applied into the associative cache, thus permitting one to map certain addresses to a given cache location (changing the fixed locality of the cache associativity)

MPX (Memory Protection eXtensions)

- New ISA set of instructions
  - Configured thru registers that define memory bounds
  - Instructions executed before memory accesses (to check the bounds)
    - On bound violation, #BR

MPX (cont.)

```
int main(int argc, char* argv) {
        int buf[100];
        return buf[argc];
}
```

Original Assembly (no-MPX):

```
    movslq  %edi, %rdi
    movl    -120(%rsp,%rdi,4), %eax
```

Assembly (gcc supporting MPX):

```
    movl    $399, %edx                  // load array length to edx
    movslq  %edi, %rdi                  // rdi contains value of argc
    leaq    -104(%rsp), %rax            // load start address of buf to rax
    bndmk   (%rax,%rdx), %bnd0          //  create bounds for buf
    bndcl   (%rax,%rdi,4), %bnd0     // check that memory access doesn't violate buf's low bound
    bndcu   3(%rax,%rdi,4), %bnd0  // check that memory access doesn't violate buf's upper bound
    movl    -104(%rsp,%rdi,4), %eax  // original memory access
```

Source:
https://software.intel.com/en-us/blogs/2013/07/22/intel-memory-protection-extensions-intel-mpx-support-in-the-gnu-toolchain

MPX (Abuse)

- Two components in a malware code:
  - User-land
  - Kernel-land
- Kernel-land hooks #BR, user-land, uses bound exceptions as trampoline to the kernel
- Kernel-land defines the flow of user-land application, inviable for AV to track the flow

Conclusions

- Assumptions are dangerous. When we compose a system, lots of assumptions are part of the final design
- The architecture is well defined, but complex. Software on top of that, even more
- SGX (already discussed past year) and other technologies will change the way we imagine computer forensics, it is time to evolve!

Resources

- All the code discussed will be available at:

https://github.com/rrbranco/Troopers2015

- We will update it with other ideas that we had no time to implement/test thus we did not present

- Latest PoC || GTFO has the AES-NI part, but we will put it there as well ;)

“As the area of our knowledge grows, so too does the perimeter of our ignorance”

The end! Really is??

Rodrigo Rubira Branco (@bsdaemon) & Gabriel Negreira Barbosa (@gabrielnb)
{ rodrigo.branco || gabriel.negreira.barbosa } @ intel.com