BARing the System

New vulnerabilities in Coreboot & UEFI based systems

Presenting: Yuriy Bulygin (@c7zero), Oleksandr Bazhaniuk (@ABazhaniuk)
Andrew Furtak, John Loucaides, Mikhail Gorobets

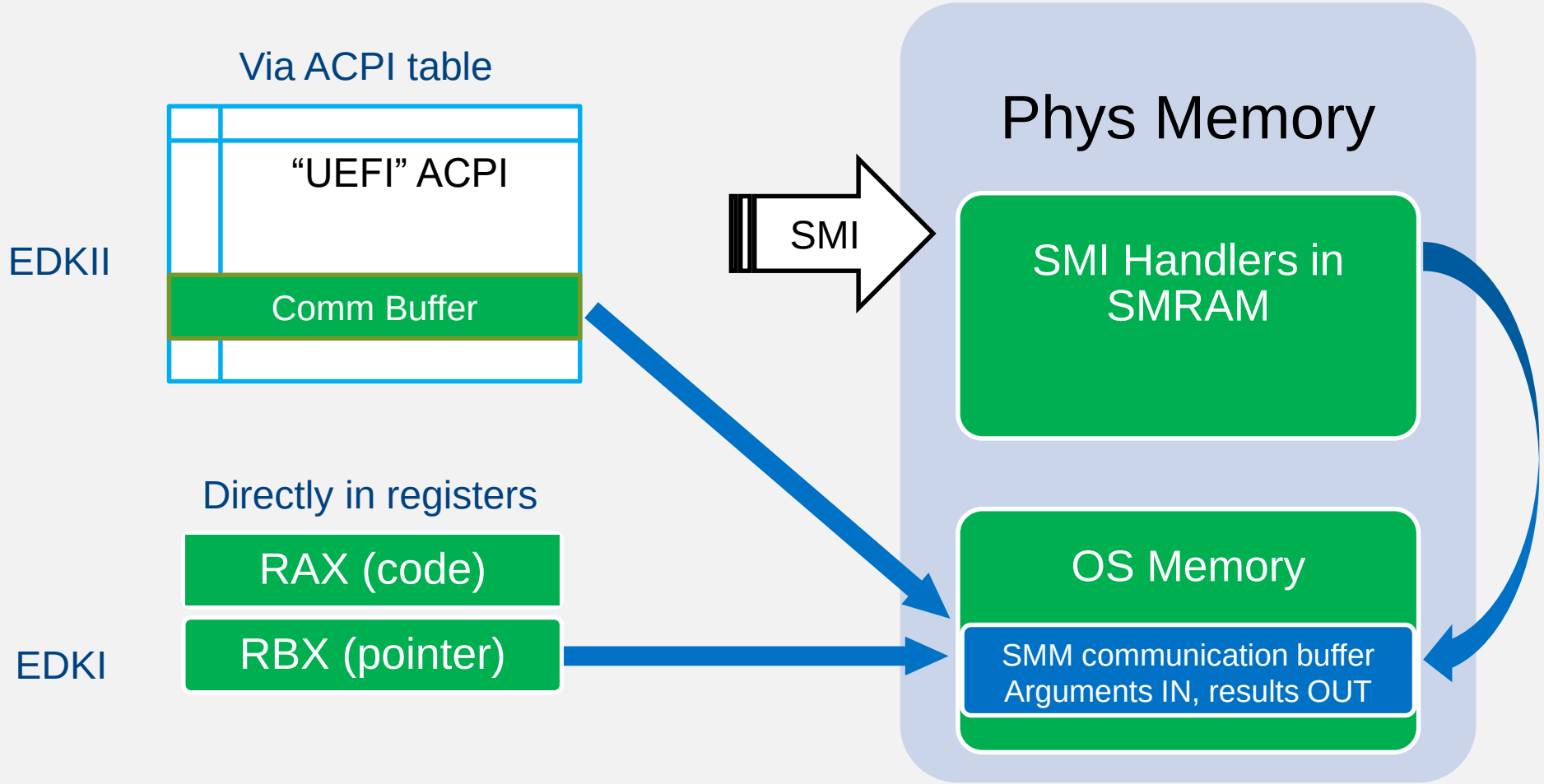
intel Security Advanced Threat Research

Agenda

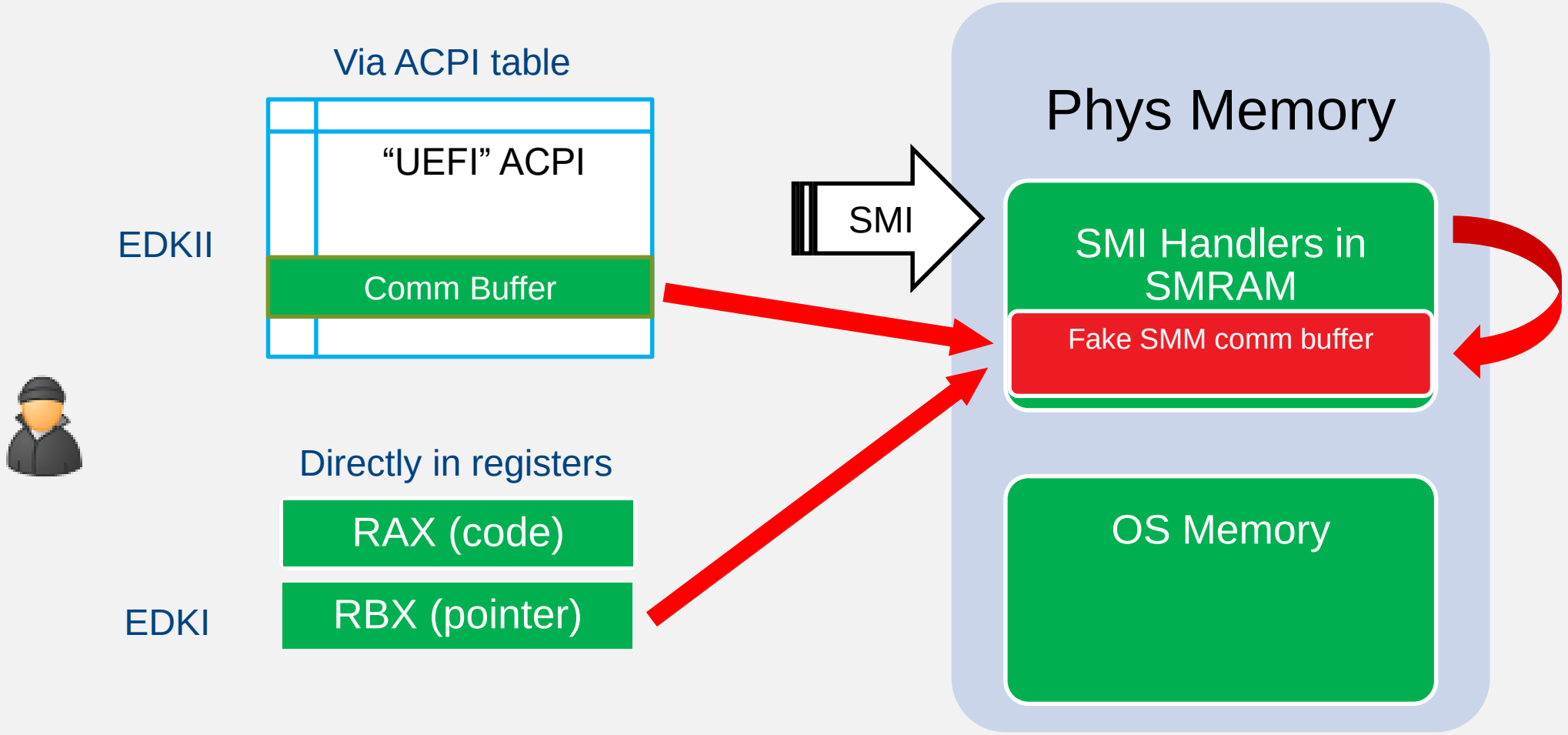
- Recap of SMM Pointer Vulnerabilities
- Intro to Memory-Mapped I/O
- MMIO BAR Issues
- MMIO BAR Issues in UEFI Firmware
- MMIO BAR Issues in Coreboot Firmware
- Limitations
- Mitigations
- Tools
- Conclusion

Recap of SMM pointer vulnerabilities

Pointer Arguments to SMI Handlers

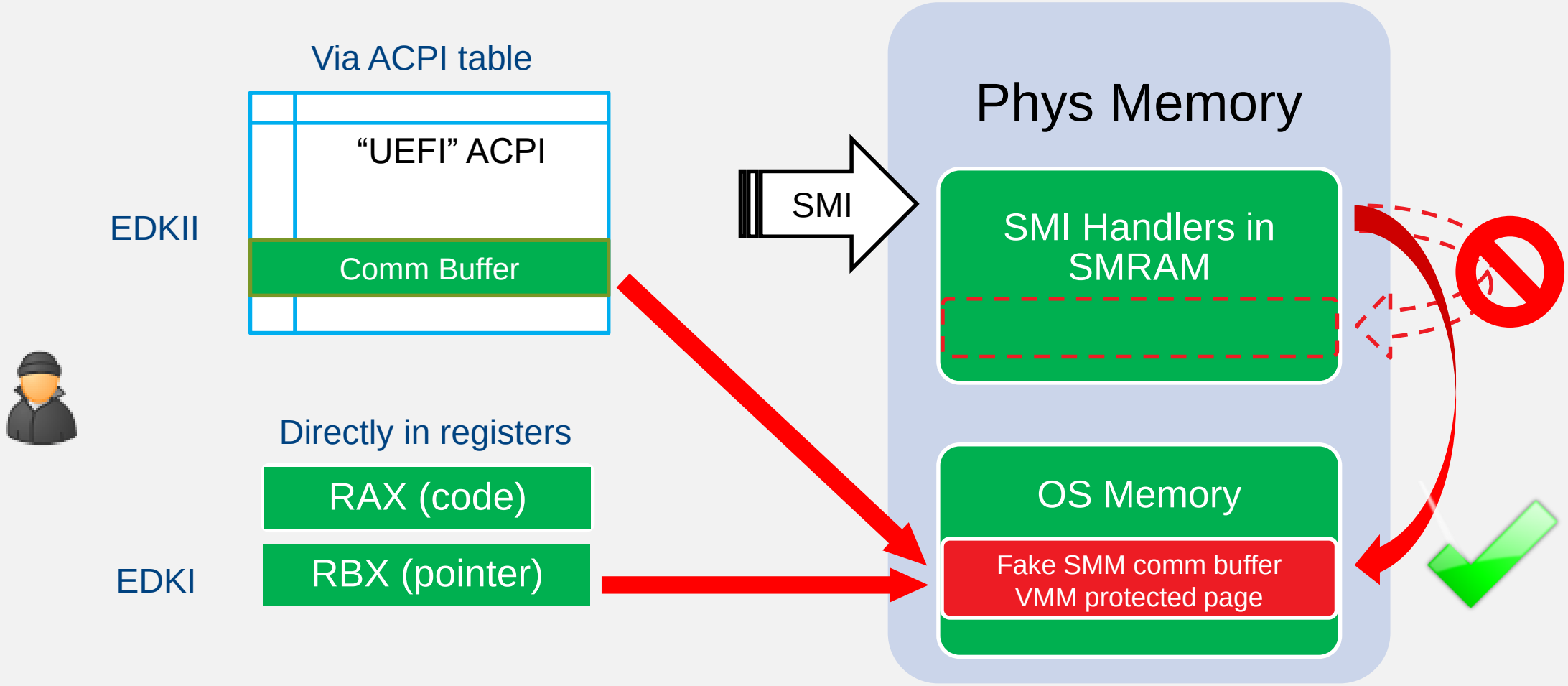


Exploiting SMM pointers…



Exploit tricks SMI handler to write to an address **in SMRAM** (Attacking and Defending BIOS in 2015)

Attacking hypervisors via SMM pointers…



Even though SMI handler check pointers for overlap with SMRAM, exploit can trick it to write to VMM protected page (Attacking Hypervisors via Firmware and Hardware)

Example: SMIFlash SMI Handler

```
1  VOID SMIFlashSMIHandler (
2  ..
3      SwSmi = (UINT8)DispatchContext->SwSmiInputValue;
4      CpuState = pSmst->CpuSaveState;
5      AddrHi = CpuState[Cpu].Ia32SaveState.ECX;
6      AddrLo = CpuState[Cpu].Ia32SaveState.EBX;
7      Buff = AddrHi;
8      Buff = Shl64(Buff, 32);
9      Buff += AddrLo;
10 ..
11         switch (SwSmi) {
12 ..
13             case 0x21:
14                 ReadFlashData( (FUNC_BLOCK *)Buff );
15 ..
16             case 0x25:
17                 ReadFlashInfo( (INFO_BLOCK *)Buff );
18         }
19 ..
20 EFI_STATUS ReadFlashData(IN OUT FUNC_BLOCK *func)
21 ..
22     sts = Flash->Read(
23               (UINT8*)(FlashStart + func->BlockAddr),
24               func->BlockSize,
25               (UINT8*)func->BufAddr
26         );
```

Reported by ATR to BIOS vendor in June 2014

Similar to publication by Sogeti ESEC Lab

```
void EFIAPI SwSMIDispatchFunction(EFI_HANDLE DispatchHandle, EFI_SMM_SW_DISPATCH_CONTEXT
{
    // ...
    struct smiflash_arg *pointer; // see bellow for the struct
    int smi_number = DispatchContext->SwSmiInputValue;

    // Retrieve a pointer from user provided value
    pointer = ecx << 32 | ebx;

    if (smi_number != 0x25)
        pointer->unknown = 0;

    // some init ...

    switch (smi_number) {
        case 0x20:
            // ...
            break;
        case 0x21:
            swsmi_handler21(pointer);
```

SMI handlers now validate input pointers

- SMI handlers now validate pointer + offsets received from the OS for overlap with SMRAM before using it (`SmmIsBufferOutsideSmmValid`). This does not block exploits using SMI handlers as proxies to attack hypervisor pages (Hyper-V, Windows 10 Virtual Secure Mode)
- Most recently, EDKII implemented `CommBuffer` at fixed memory location to mitigate attacks on hypervisors and reporting to Windows through the Windows SMM Mitigations ACPI Table (WSMT)

Table 2. Protection Flags Field

Length	Bit offset	Description
1	0	FIXED_COMM_BUFFERS If set, expresses that for all synchronous SMM entries, SMM will validate that input and output buffers lie entirely within the expected fixed memory regions.
1	1	COMM_BUFFER_NESTED_PTR_PROTECTION If set, expresses that for all synchronous SMM entries, SMM will validate that input and output pointers embedded within the fixed communication buffer only refer to address ranges that lie entirely within the expected fixed memory regions.
1	2	SYSTEM_RESOURCE_PROTECTION If set, expresses that firmware has taken steps ensuring that configuration for any system resources that are not configurable through an architectural mechanism (e.g. ACPI or PCI MMIO) must be locked by firmware before transitioning to Windows.

Memory-Mapped I/O (MMIO)

PCI Express

- PCI Express Fabric consists of PCIe components connected over PCIe interconnect in a certain topology (e.g. hierarchy)
- *Root Complex* is a root component in a hierarchical PCIe topology with one or more PCIe *root ports*
- Components: *Endpoints* (I/O Devices), *Switches*, PCIe-to-PCI/PCI-X *Bridges*
- All components are interconnect via PCI Express Links
- Physical components can have up to 8 physical or virtual *functions*
- Some endpoints are *integrated* into Root Complex

PCIe Config Space Layout

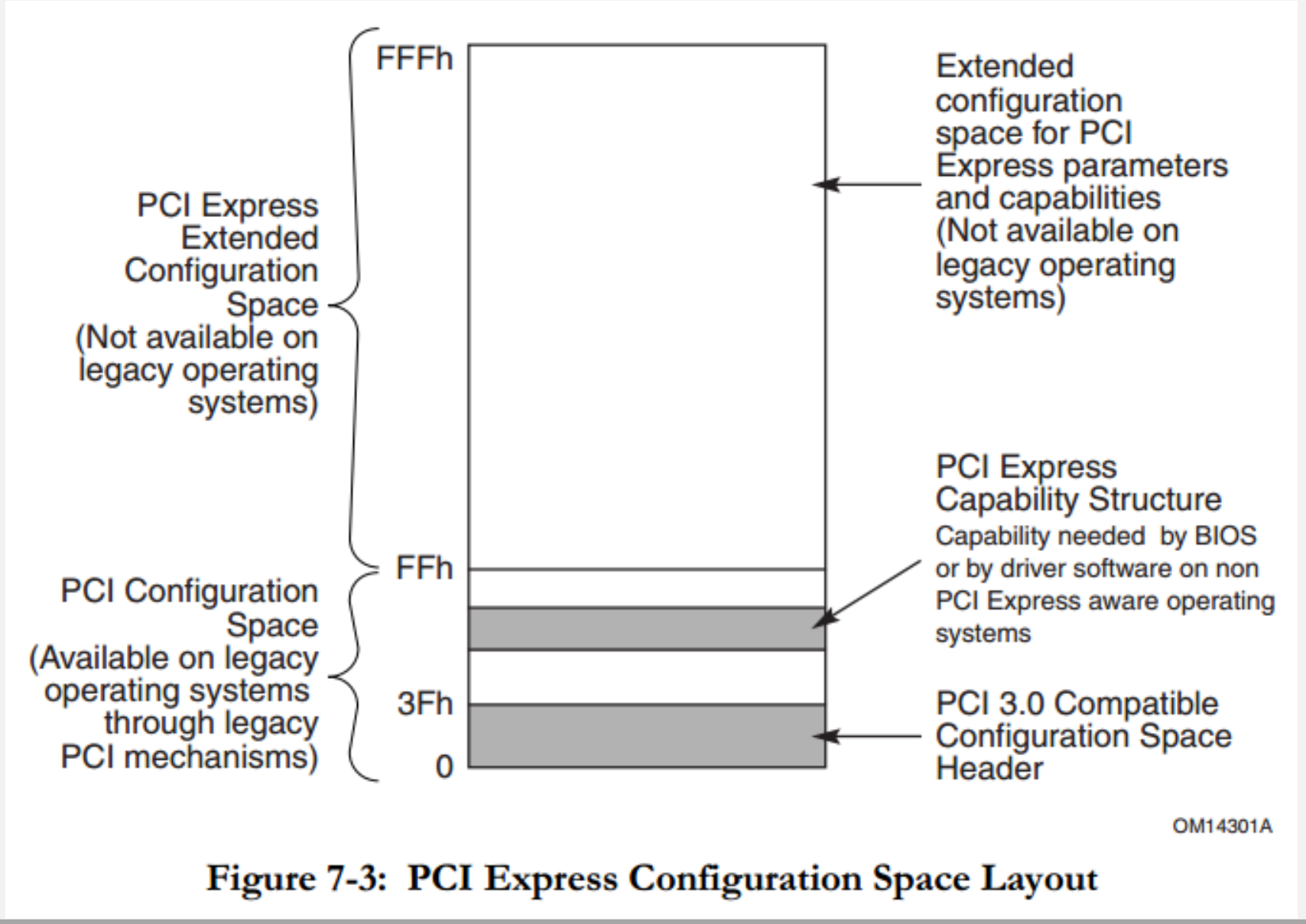


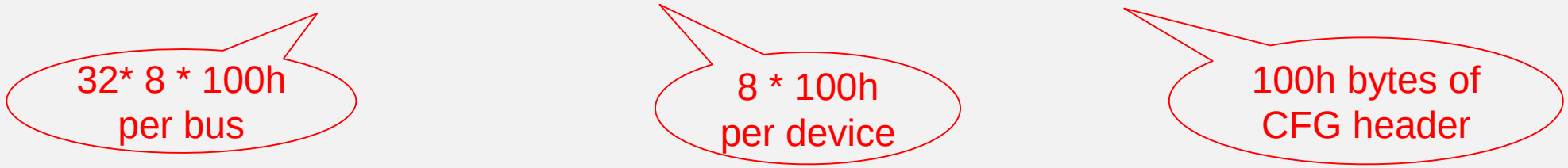
Figure 7-3: PCI Express Configuration Space Layout

Source: PCI Express Base Specification Revision 3.0

PCI/PCIe Config Space Access

1. Software uses processor I/O ports `CF8h` (*control*) and `CFCh` (*data*) to access PCI configuration of `bus/dev/fun`. Address (written to control port) is calculated as:

```
bus << 16 | dev << 11 | fun << 8 | offset & ~3
```

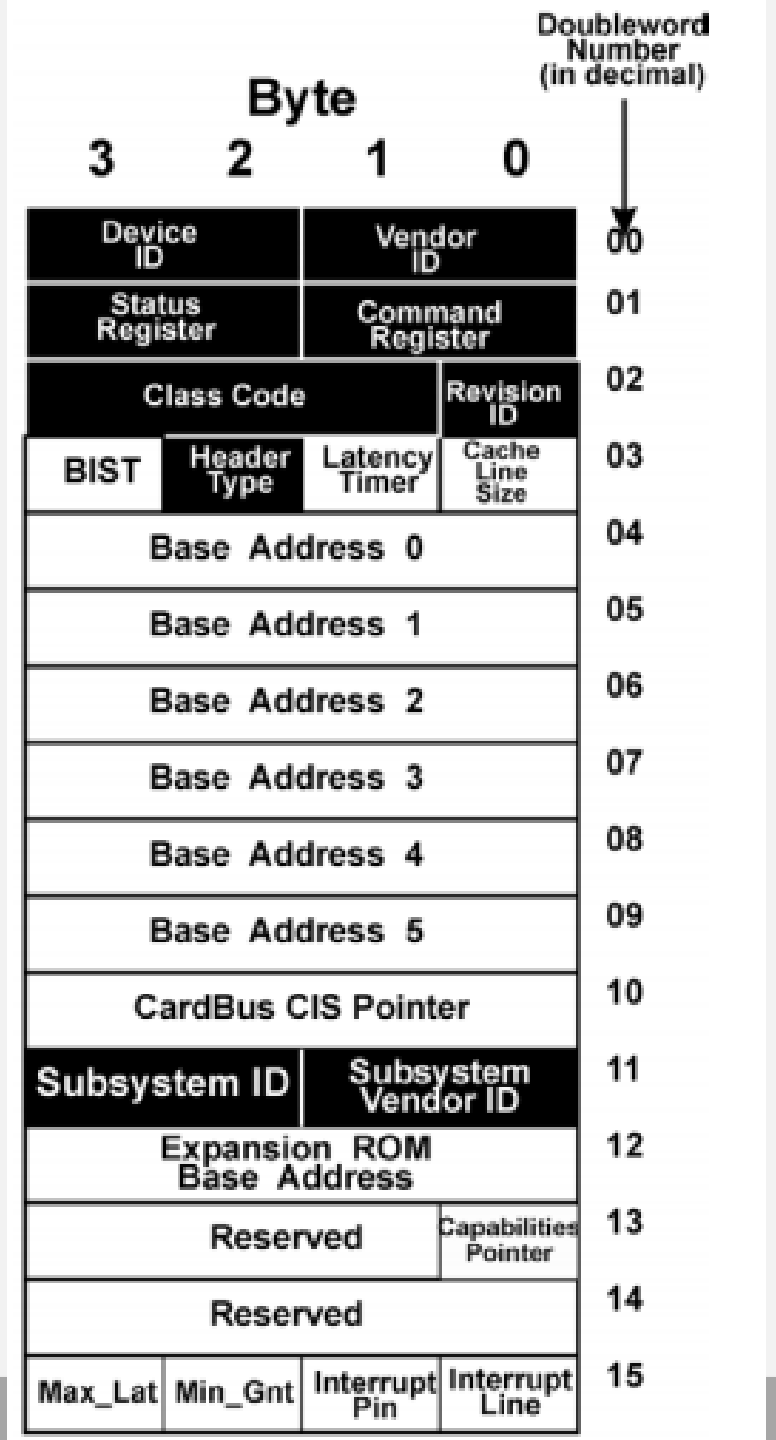


2. *Enhanced Configuration Access Mechanism* (ECAM) allows accessing PCIe extended configuration space (4kB) beyond PCI config space (256 bytes)
   - Implemented as memory-mapped range in physical address space split into 4kB chunks per B:D.F
   - Register address is a memory address within this range

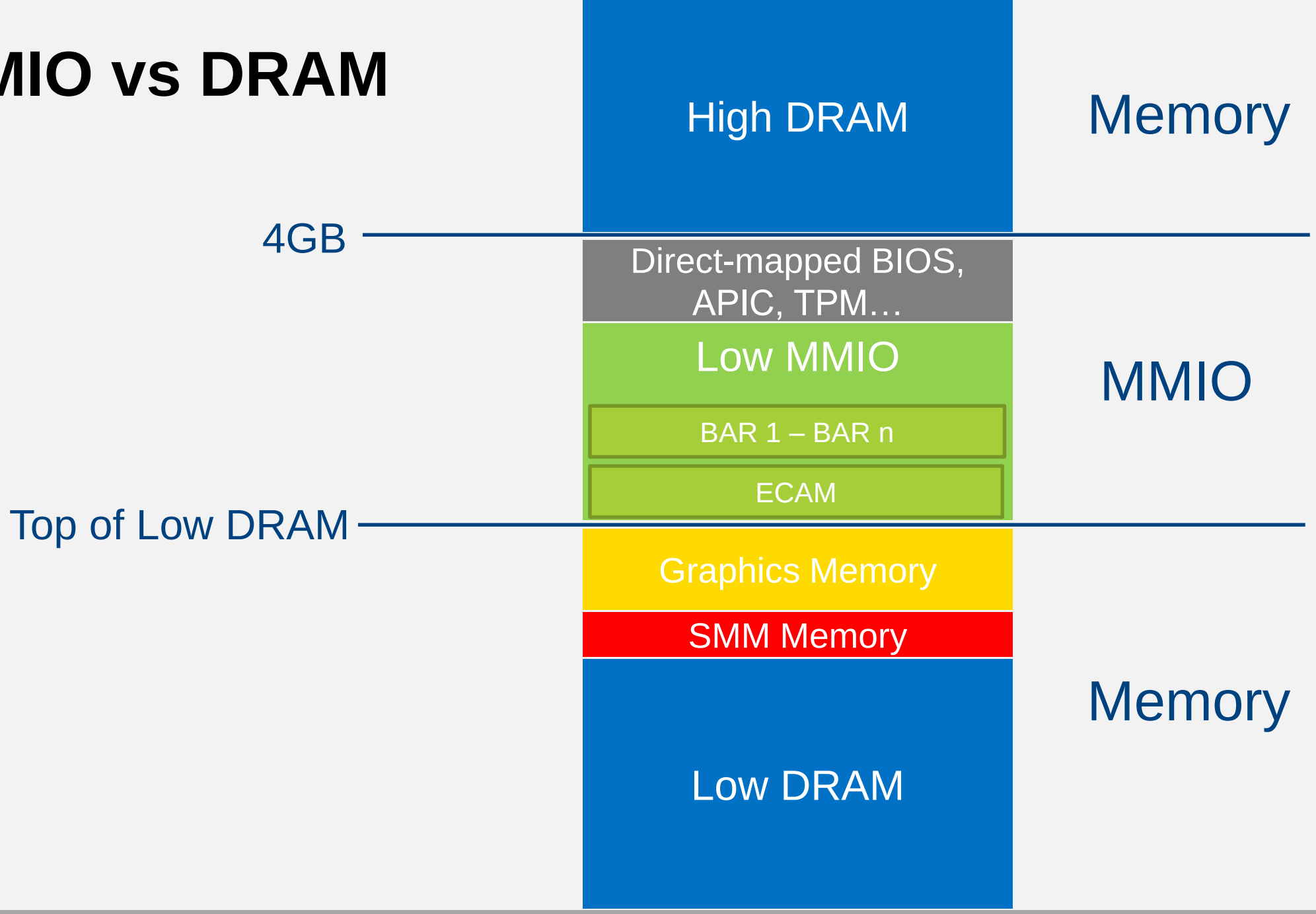
```
MMCFG base + bus*32*8*1000h + dev*8*1000h + fun*1000h + offset
```

Memory-Mapped I/O

- Devices need more space for registers
- ➔ Memory-mapped I/O (MMIO)
- MMIO range is defined by Base Address Registers (BAR) in PCI configuration header
- Access to MMIO ranges forwarded to devices



MMIO vs DRAM


High DRAM
Memory
4GB
Direct-mapped BIOS,
APIC, TPM…
Low MMIO
MMIO
BAR 1 – BAR n
ECAM
Top of Low DRAM
Graphics Memory
SMM Memory
Memory
Low DRAM

MMIO BARs

MMIO Range	BAR	Base	Size	En	Description
GTTMMADR	00:02.0 + 10	00000000F0000000	00400000	1	Graphics Translation Table Range
SPIBAR	00:1F.0 + F0	00000000FED1F800	00000200	1	SPI Controller Register Range
HDABAR	00:03.0 + 10	0000007FFFFFF000	00001000	1	HD Audio Register Range
GMADR	00:02.0 + 18	00000000E0000000	00001000	1	Graphics Aperture
DMIBAR	00:00.0 + 68	00000000FED18000	00001000	1	Root Complex Register Range
MMCFG	00:00.0 + 60	00000000F8000000	00001000	1	PCI Express Register Range
RCBA	00:1F.0 + F0	00000000FED1C000	00004000	1	PCH Root Complex Register Range
MCHBAR	00:00.0 + 48	00000000FED10000	00008000	1	Memory Controller Register Range

...

MMIO Range Relocation

- MMIO ranges can be *relocated* at runtime by the OS
  - OS would write new address in BAR registers

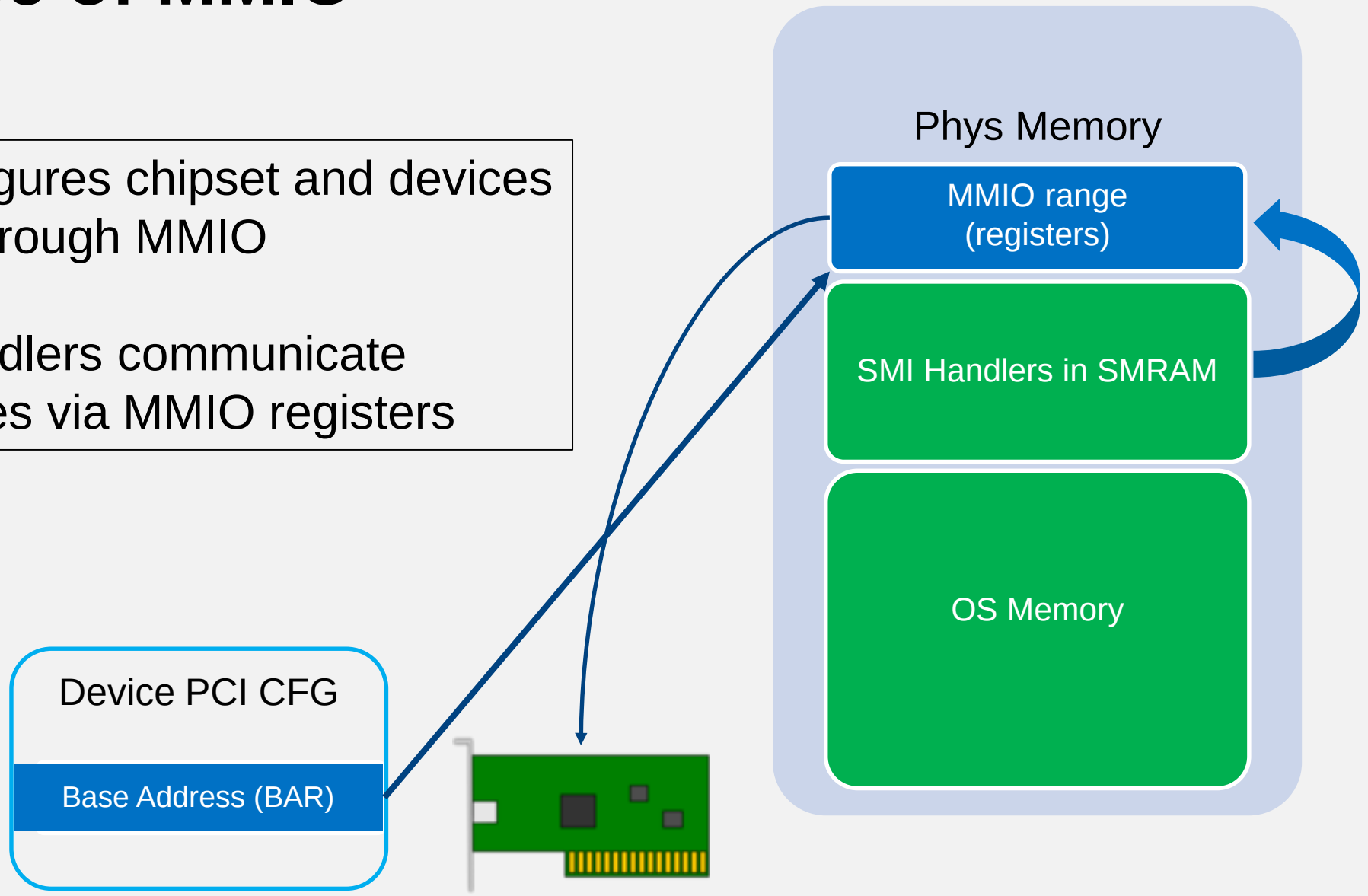
- Certain MMIO ranges cannot be relocated at runtime
  - Fixed (e.g. direct-access BIOS range)
  - Or locked down by the firmware (e.g. MCHBAR)

MMIO BAR Issues

Firmware use of MMIO

Firmware configures chipset and devices through MMIO

SMI handlers communicate with devices via MMIO registers

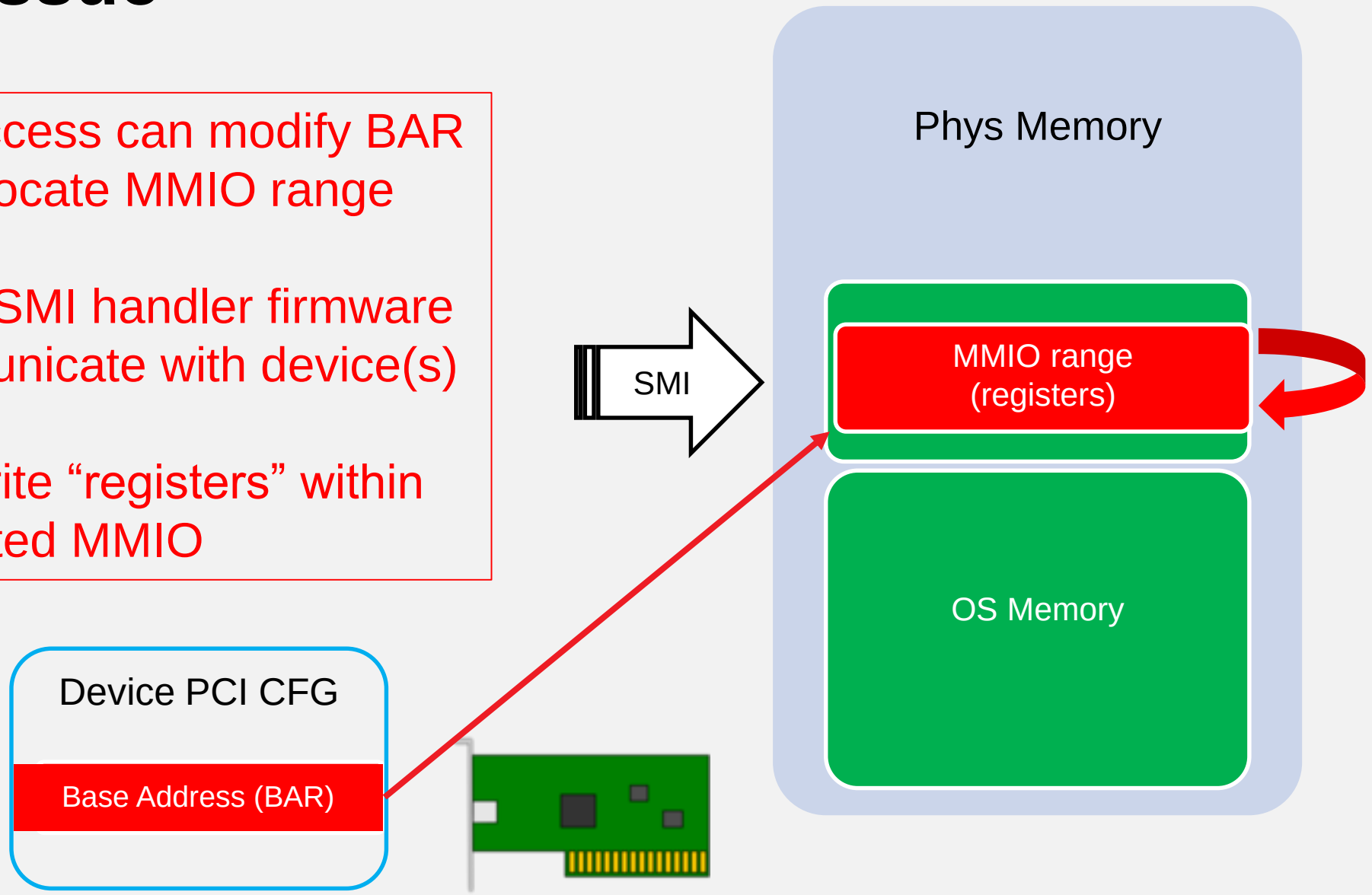


MMIO BAR Issue

Exploit with PCI access can modify BAR register and relocate MMIO range

On SMI interrupt, SMI handler firmware attempts to communicate with device(s)

It may read or write “registers” within relocated MMIO

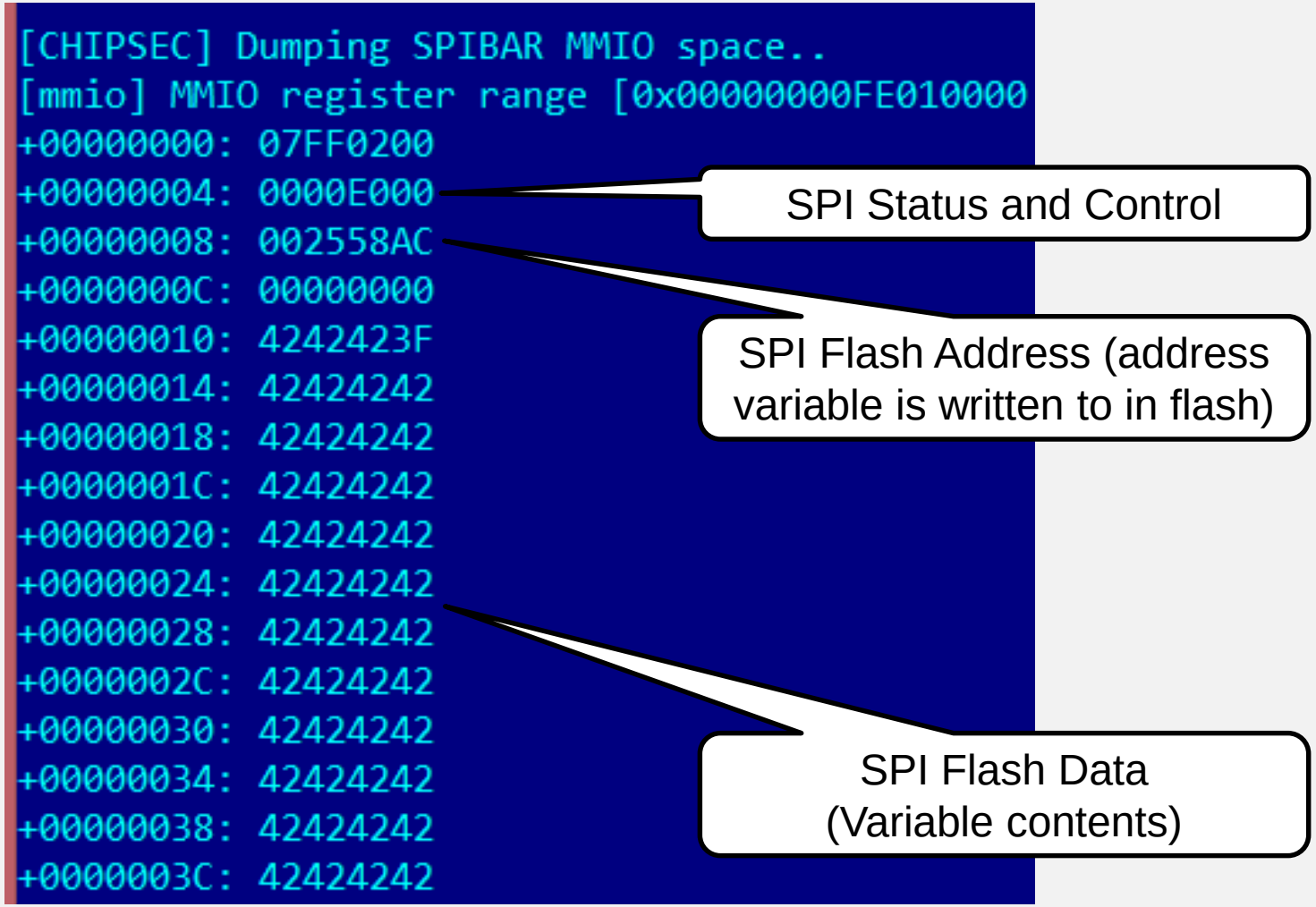


Examples of MMIO BARs accessed in SMM

- EHCI (USB 2.0) controller MMIO BAR (B0:D26:F0, B0:D29:F0)
- GBe LAN MMIO BAR (B0:D25:F0)
- Root Complex Block Address (RCBA) on earlier platforms (B0:D31:F0)
- SPI BAR on Skylake or later generations (B0:D31:F5)
- AHCI (SATA) controller MMIO BAR (B0:D31:F2, B0:D31:F5)
- xHCI (USB 3.0) controller MMIO BAR (B0:D20:F0)
- Integrated Graphics Device MMIO BAR (B0:D2:F0)
- B1:D0.F0 MMIO BAR
- …

SPI Controller MMIO BAR (Access to SPI Flash)

```
# chipsec_util.py uefi var-write B 55555555-4444-3333-2211-000000000000 B.bin
# chipsec_util.py mmio dump SPIBAR
```



Finding MMIO BAR issues at runtime

Goal: Find all MMIO registers modified by SMI handler

1. Dump MMIO range
2. Trigger SMI
3. Dump MMIO range and compare all registers

Problem: many registers are modified by devices all the time! Up to 30,000 registers change in Graphics Device MMIO

Finding MMIO BAR issues at runtime

Goal: Find all MMIO registers modified by SMI handler

1. Dump MMIO range multiple times
2. Find all registers which frequently change without SMM
3. Dump MMIO range
4. Trigger SMI
5. Dump MMIO range and compare all registers
6. Find registers which don’t normally change
7. Repeat this multiple times to confirm suspected registers are actually being modified in SMM
8. Copy original contents of MMIO range to memory
9. Relocate MMIO range (change its base address) to this memory
10. Generate SMI
11. Monitor changes in memory at suspected offsets

Monitoring changes in USB MMIO BAR

```
[x][ Module: Monitors MMIO changes done by SMI handlers
[x][ ============================================================
[*] Configuration:
    MMIO BAR names: ['USBBAR']
    Generate SMI: True
    SMI codes: [0x00:0x00]
[*] SMM comm buffer (EBX)  : 0x00000000D9469000
[*] MMIO BAR 'USBBAR': base = 0x00000000F063C000, size = 0x00001000
[*] reading contents of MMIO BARs ['USBBAR']
    reading 'USBBAR'
[*] calculating normal MMIO BAR differences..
[*] 'USBBAR' normal difference (5 diffs):
    diff0: 0 regs []
..
    diff19: 2 regs [70, 74]
    2 regs changed: [70, 74]
[*] fuzzing SMIs..
[*] SMI# 00: data 00, func (ECX) 0x00000000
    reading 'USBBAR'
    generating SMI
    reading 'USBBAR'
    diffing 'USBBAR' (1024 regs)
    2 regs changed: [70, 77]
    new regs: [77]
[!] New changes found!
    repeating SMI
    reading 'USBBAR'
    diffing 'USBBAR' (1024 regs)
    2 regs changed: [70, 74]
    new regs: []
```

MMIO BAR Issues in UEFI Firmware

Finding MMIO BAR issues in binaries

1. Consider MMIO BAR (MBARA) of GBe LAN device (B0:D25:F0) at offset `0x10`
2. Legacy PCIe config address is

   `(25<<11) + 0x10 =` **`0xC810`**

   `0x8000C810` if with Enable bit (31) set
3. Memory-mapped ECAM address is ECAM base + offset to 4kB page of B0:D25:F0 + BAR register offset

   `0xF8000000 + 0xC8010 =` **`0xF80C8010`**
4. Look for these constants in the binaries of SMI handlers

GBe LAN MMIO BAR (B0:D25:F0)

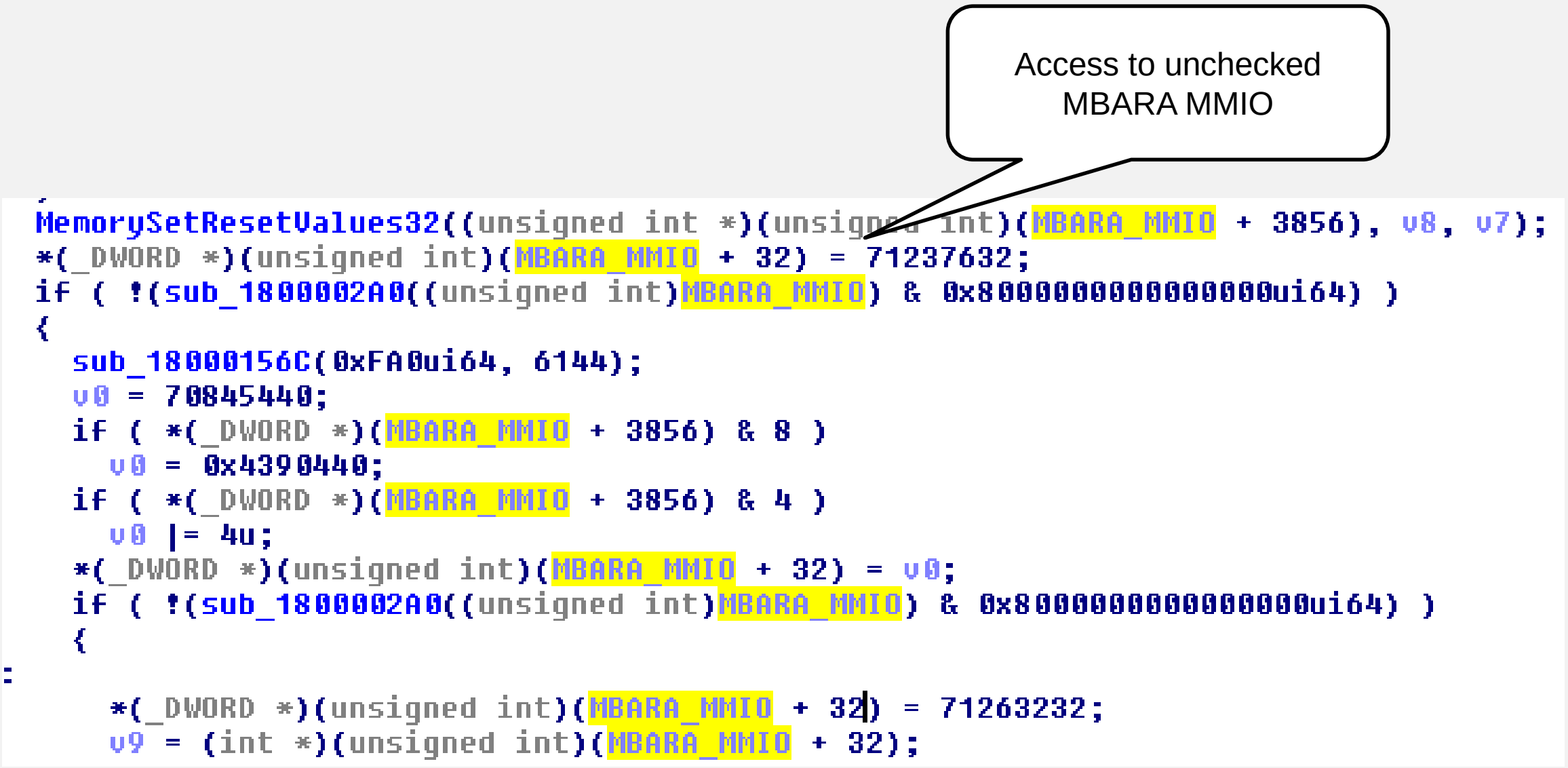
```
__int16 GBE_MMIO_access_func()
{
  signed int v0; // ebp@1
  unsigned int *v1; // rax@1
  __int64 MBARA_MMIO; // rbx@3
  unsigned int v3; // er12@4
  int v4; // er14@4
  unsigned int v5; // edi@4
  int v6; // er13@5
  int v7; // er8@10
  int v8; // edx@10
  int *v9; // rdi@18

  v0 = 0;
  LOWORD(v1) = MEMORY[0xF80F8048];
  if ( !(*(_WORD *)((MEMORY[0xF80F80F0] & 0xFFFFFF      0x3414i64) & 0x20) )
  {
    LOWORD(v1) = MEMORY[0xF80C80CC];
    if ( !(MEMORY[0xF80C80CC] & 3) )
    {
      MBARA_MMIO = MEMORY[0xF80C8010];
      LODWORD(v1) = *(_DWORD *)(MEMORY[0xF80C8010] + 0x5800i64);
      if ( (unsigned __int8)v1 & 1 )
      {
        v3 = *(_DWORD *)(MEMORY[0xF80C8010] + 0x5400i64);
        v4 = *(_DWORD *)(MEMORY[0xF80C8010] + 0x5404i64);
        v5 = 0;
        *(_DWORD *)(MEMORY[0xF80C8010] + 0xF00i64) |= 0x20u;
```

Read GBe MMIO BAR register from MMCFG:
`0xF8000000 + 0x19<<15 + 0x10 =`
0xF80C8010

Access to MBARA MMIO registers

GBe LAN MMIO BAR (B0:D25:F0)



EHCI MMIO BAR (B0:D29:F0)

```
16  v0 = 0;
17  dev1 = 0x1D;                                // EHCI controller
18  dev2 = 0x1A;                                // EHCI controller
19  v9 = 0;
20  v11 = 0;
21  result = sub_180001878();
22  if ( (unsigned __int8)result > 0u )
23  {
24    do
25    {
26      v2 = (unsigned __int8)*(&v9 + 2 * v0) << 12;
27      v3 = ((unsigned __int8)*(&dev1 + 2 * v0) << 15) + 0xF8000000i64;
28      Mem_BAR = *(unsigned int *)(v3 + v2 + 0x10);
29      v5 = (__int16 *)(v3 + v2 + 4);
30      v6 = *(_WORD *)(v3 + v2 + 0x54);
31      v7 = *v5;
32      if ( (_DWORD)Mem_BAR != -1 )
33      {
34        if ( (*(_WORD *)(((unsigned __int8)*(&dev1 + 2 * v0) << 15)
35                       + 0xF8000000i64
36                       + ((unsigned __int8)*(&v9 + 2 * v0) << 12)
37                       + 0x54) & 3) == 3 )
38        {
39          *(_WORD *)(((unsigned __int8)*(&dev1 + 2 * v0) << 15)
40                     + 0xF8000000i64
41                     + ((unsigned __int8)*(&v9 + 2 * v0) << 12)
42                     + 0x54) = v6 & 0xFFFC;
43          *(_DWORD *)(v3 + v2 + 0x10) = Mem_BAR;
44          sub_180001A28(v3 + v2 + 4, 2i64);
45        }
46        if ( !(*(_DWORD *)(Mem_BAR + 0x20) & 1) )
47        {
48          MemorySetResetValues32((unsigned int *)(unsigned int)(Mem_BAR + 0x20), 0, 48);
49          MemorySetResetValues32((unsigned int *)(unsigned int)(Mem_BAR + 0x20), 1, 0);
50          v5 = (__int16 *)(v3 + v2 + 4);
51        }
```

Calculate EHCI BAR register address in MMCFG:
`0xF8000000 + 0x1D<<15 + 0x10`

Read EHCI MMIO BAR

Modify MMIO register `0x20` in EHCI MMIO range

Finding MMIO BAR issues in binaries

Identify functions reading PCI config registers via legacy or ECAM access and find ones reading BAR registers using above constants

Legacy PCI config read

```
pci_dword_read  proc near               ; CODE XREF: sub_10008450+2E↑p
                                        ; sub_100085C4+B2↑p
                sub     rsp, 28h
                mov     edx, ecx
                mov     cx, 0CF8h
                call    outdword
                mov     cx, 0CFCh
                add     rsp, 28h
                jmp     indword
pci_dword_read  endp
```

Access to register `0x88` in `B0:D31:F0`

```
__int64 sub_10000320()
{
  unsigned __int32 v0; // eax@1

  v0 = pcie_read_dword(0i64, 0x1Fu, 0, 0xB8u);
  return pcie_write_dword(0, 0x1Fu, 0, 0xB8u, v0);
}
```

Extended PCIe config access through MMCFG

```
unsigned __int32 __fastcall pcie_read_dword(__int64 a1, unsigned __int8 a2, unsigned __int8 a3, unsigned __int16 a4)
{
  if ( a4 >= 0x100u )
    return *(_DWORD *)((a3 << 12) + (a2 << 15) + ((unsigned __int8)a1 << 20) + (unsigned int)a4 - 0x8000000);
  outdword(0xCF8u, a4 & 0xFC | ((a3 | 8 * (a2 | 32 * (unsigned __int8)a1)) << 8) | 0x80000000);
  return indword(0xCFCu);
}
```

MMIO BAR Issues in Coreboot Firmware

Finding MMIO BAR issues in the source code

1. Find functions within SMI handlers which read MMIO BAR PCI config registers (offsets `0x10-0x24` or chipset specific offsets for integrated devices)
   - BAR registers can be read using memory-mapped config reads (offsets in ECAM memory space). In this case, normal memory reads will be used

```
    reg_base = (void *)((uintptr_t)pci_read_config32(SA_DEV_IGD,
PCI_BASE_ADDRESS_0) & ~0xf);
```

2. Find all memory accesses to offsets off of BAR addresses within SMI handlers

```
    write32(reg_base + PCH_PP_CONTROL, pp_ctrl); <<< memory write of modified
pp_cntrl value
    read32(reg_base + PCH_PP_CONTROL);
```

3. Can often search the names of the BAR registers and MMIO ranges (e.g. `SPI_BAR0`, `RCBA/RCRB`, `PCI_BASE_ADDRESS` etc.)

mainboard_io_trap_handler SMI handler

```
1  static void mainboard_smi_brightness_down(void)
2  {
3      u8 *bar;
4      if ((bar = (u8 *)pci_read_config32(PCI_DEV(1, 0, 0), 0x18))) {
5          printk(BIOS_DEBUG, "bar: %08X, level %02X\n",  (unsigned int)bar,
6          *(bar+LVTMA_BL_MOD_LEVEL) &= 0xf0;
7          if (*(bar+LVTMA_BL_MOD_LEVEL) > 0x10)
8              *(bar+LVTMA_BL_MOD_LEVEL) -= 0x10;
9      }
10 }
11
12 static void mainboard_smi_brightness_up(void)
13 {
14 ...
15         if (*(bar+LVTMA_BL_MOD_LEVEL) < 0xf0)
16             *(bar+LVTMA_BL_MOD_LEVEL) += 0x10;
17     }
18 }
19
20 int mainboard_io_trap_handler(int smif)
21 {
22 ...
23     switch (smif) {
24 ...
25     case SMI_BRIGHTNESS_UP:
26         mainboard_smi_brightness_up();
27         break;
28
29     case SMI_BRIGHTNESS_DOWN:
30         mainboard_smi_brightness_down();
```

`bar` pointer points to MMIO range of device B1:D0.F0 which can be modified by an attacker

SMI handler then uses `bar` pointer to write to `LVTMA_BL_MOD_LEVEL` offset (when adjusting brightness level)

SMI handler can be invoked by properly configuring I/O Trap hardware with `BRIGHTNESS_UP/DOWN` function

southbridge_smi_sleep SMI handler

```
1  static void backlight_off(void)
2  {
3      void *reg_base;
4      uint32_t pp_ctrl;
5      uint32_t bl_off_delay;
6
7      reg_base = (void *)((uintptr_t)pci_read_config32(SA_DEV_IGD, PCI_BASE_ADDRESS_0) & ~0xf);
8
9      /* Check if backlight is enabled */
10     pp_ctrl = read32(reg_base + PCH_PP_CONTROL);
11     if (!(pp_ctrl & EDP_BLC_ENABLE))
12         return;
13
14     /* Enable writes to this register */
15     pp_ctrl &= ~PANEL_UNLOCK_MASK;
16     pp_ctrl |= PANEL_UNLOCK_REGS;
17
18     /* Turn off backlight */
19     pp_ctrl &= ~EDP_BLC_ENABLE;
20
21     write32(reg_base + PCH_PP_CONTROL, pp_ctrl);
22     read32(reg_base + PCH_PP_CONTROL);
```

`reg_base` pointer points relocateable IGD MMIO

SMI handler then uses `reg_base` to read-modify-write `PP_CONTROL` register

```
sta        southbridge_smi_sleep(void)
{

    /* Figure out SLP_TYP */
    reg32 = inl(ACPI_BASE_ADDRESS + PM1_CNT);
    printk(BIOS_SPEW, "SMI#: SLP = 0x%08x\n", reg32);
    slp_typ = (reg32 >> 10) & 7;
...
    switch (slp_typ) {
...
    case SLP_TYP_S5:
        printk(BIOS_DEBUG, "SMI#: Entering S5 (Soft Power off)\n");

        /* Turn off backlight if needed */
        backlight_off();
```

Vulnerable `backlight_off` is invoked when system goes to S5

Limitations

1. Exploit can overwrite specific offsets off of aligned addresses
   - MMIO ranges are typically normally (size) aligned
   - Most MMIO ranges are 4kB large (Graphics MMIO is 2-4MB)
   - Example: 16kB aligned Root Complex Base + `0x38xx` (SPI registers)
   - PCI architecture allows MMIO ranges as small as 16 bytes

2. Exploit may not be able to control values written
   - Firmware SMI handlers typically write specific values to MMIO registers
   - Often do Read-Modify-Write: `reg +/- 0x10`
   - Certain SMI handler may write attacker-supplied data
   - `SetVariable` SMI handler write contents of UEFI variable supplied by the OS to `SPI_DATAx` registers in SPIBAR MMIO range

Limitations

1. Many conditions for SMI handler to start communicating with I/O device/controller
   - Device present/enabled, mode/feature supported
   - Is platform in ACPI mode?
   - Other SMM code may also use fake MMIO (and hang)
   - SMI may get triggered on difficult events – power button, on S3 resume, etc.

2. Often, SMI handlers implement protocol rather than just reading or writing to MMIO registers
   - `IF Bit X in Reg1 is set THEN Write to Reg2`
   - Poll until certain bits are set/cleared in MMIO register (wait until SPI cycle complete)
   - When SMI handler waits for the device to respond or cycle to complete then it'll hang after MMIO BAR is relocated

3. Non PCI-architectural BAR registers are locked down by boot firmware and cannot be relocated (MCHBAR, DMIBAR etc.)

Mitigations

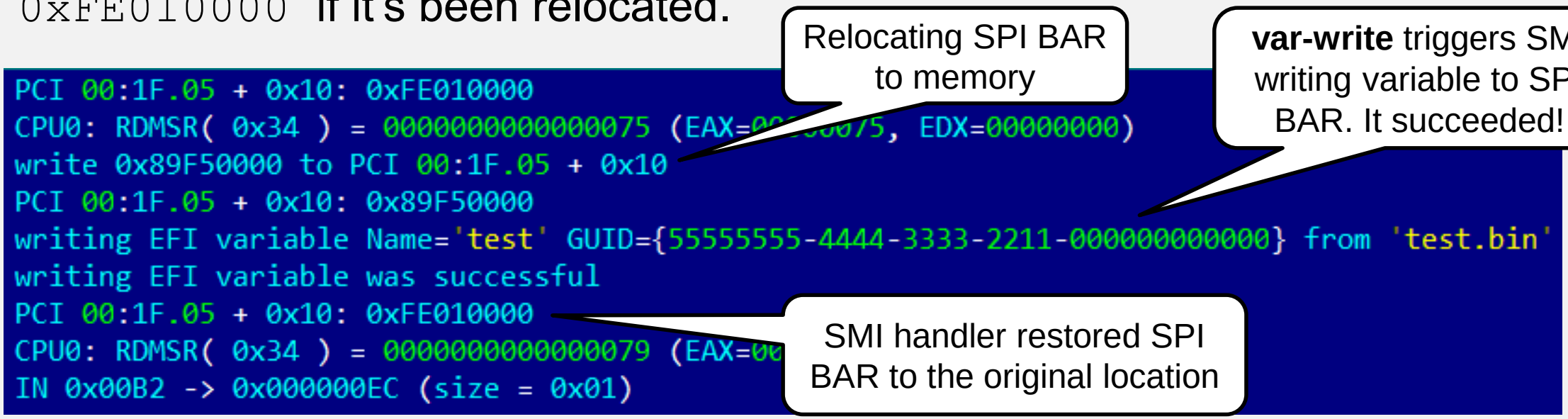
Option 1. SMI handlers can verify MMIO BAR doesn’t overlap with SMRAM

Option 2. Firmware can verify that MMIO BAR is not in DRAM (e.g. between TOLUD and 4GB or above TOUUD). This would ensure all BARs used by firmware are within MMIO

Option 3. Firmware can reserve default MMIO range for all BARs. Before accessing MMIO range, SMI handlers can relocate BARs to the default range if they point to somewhere else

SPIBAR Mitigation Example

- On latest platforms, SPI MMIO is a separate 4kB range rather than a part of Root Complex MMIO
- Firmware reserves `0xFE010000` page for SPI MMIO and programs `SPI_BAR0` register in SPI controller with this address.
- On any PCH SMI, SMI handler checks SPI_BAR0 and restores it to `0xFE010000` if it's been relocated.



Tools to assist in finding/analyzing these issues

`tools.smm.rogue_mmio_bar`

Attempts to create fake MMIO ranges in memory, relocate hardware MMIO BARs to the fake memory, then observe changes made by SMI handlers in relocated MMIO ranges

`tools.smm.bar`

Simply monitors changes made by SMI handlers in MMIO registers of specified MMIO BARs

Conclusion

- The root cause is that firmware assumes hardware is trusted
- Hardware registers like PCI Base Address Registers can be modified by runtime software (some are locked down)
- Firmware shouldn’t assume addresses in BAR registers are correct and should treat them as untrusted input
- Boot firmware should also validate contents of BAR registers upon resume from sleep if it restores them from S3 boot script

Thank You!