Getting into the SMRAM: SMM Reloaded

Loïc Duflot, Olivier Levillain,
Benjamin Morin and Olivier Grumelard

Central Directorate for
Information Systems Security
SGDN/DCSSI 51 boulevard de la Tour Maubourg 75007 Paris

loic.duflot@sgdn.gouv.fr

Premier ministre
Secrétariat général de la défense nationale

Introduction

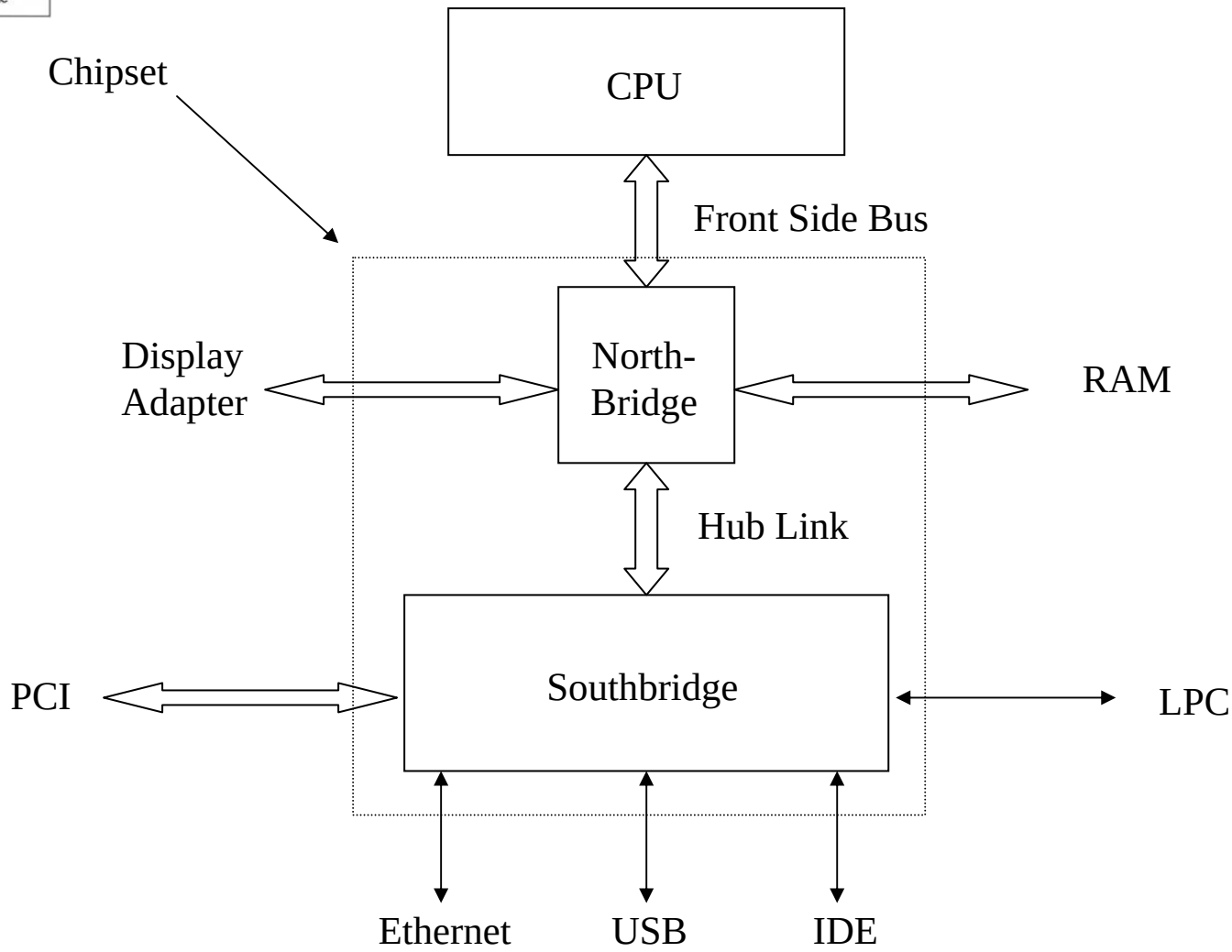
- System Management Mode (SMM) is one of the x86 (and x86-64) CPUs operating modes.
- It has been shown (CanSecWest 2006) how system management mode could be used by attackers as a means for privilege escalation.
- Additional details have been given by Ivanlef0u and Branco (BSDDeamon) et al. in Phrack.
- A first SMM rootkit has been presented during Black Hat briefings 2008 (Sparks, Embleton).
- All these presentations/papers concluded that attackers could do various interesting things (at least for them) if they were able to modify the content of the so-called SMRAM, but that there were limitations in practice.

- The goal of this presentation is to show that some of these limitations can be overtaken.

Outline

- Introduction
- A (short) description of SMM
- Offensive use of SMM
  - Potential uses: privilege escalation schemes, rootkits
  - Limitations
- Circumventing the D_LCK bit
  - Memory caching
  - Cache poisoning
  - Applications and demo
- Impact
- Countermeasures and conclusion

Simplified PC architecture



What is System Management Mode?

- A 16-bit mode.
- Used for motherboard control and power management, for instance:
  - Thermal management.
  - Power management on laptops (sometimes called by ACPI).
- Can only be entered when the CPU receives a so-called hardware System Management Interrupt (SMI).
  - SMIs are generated by the chipset (Northbridge).
  - Writes to the Advanced Power Management Control Register (AMPC) trigger a SMI (outl(something, 0xb2)).
  - This can be done by anyone with input/output privileges.

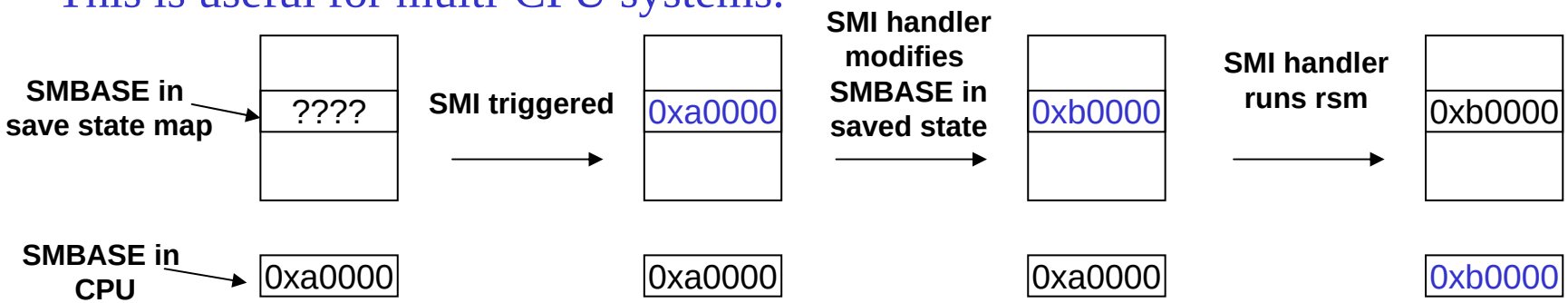
System Management Mode entry

- Upon entry, almost every single CPU register is saved in a “memory saved state map” that is itself stored into a memory zone called SMRAM.
- SMRAM is located in RAM.
- An SMI handler is executed from SMRAM.
- When the SMI handler runs the “rsm” assembly language instruction the CPU state is entirely restored from the map saved in memory.

- The operating system does not even notice when it is being interrupted by management software running in SMM.

Location of the SMRAM

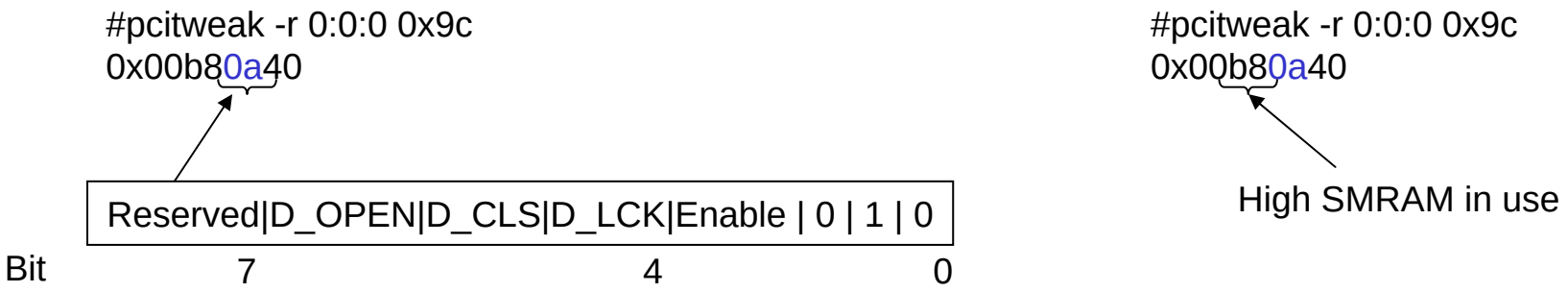
- Specified in the SMBASE register of the CPU. This register cannot be accessed at all. Its content is copied in the SMRAM saved state map and can only be changed when the CPU state is restored.
- This is useful for multi-CPU systems.



- In practice SMBASE is usually :
  - 0xa0000: legacy SMRAM location.
  - 0xfeda0000 (+/- 0x8000): high SMRAM location.
  - Something else: TSEG (Extended SMRAM)
- The base address of the SMI handler is SMBASE + 0x8000 (fixed offset).

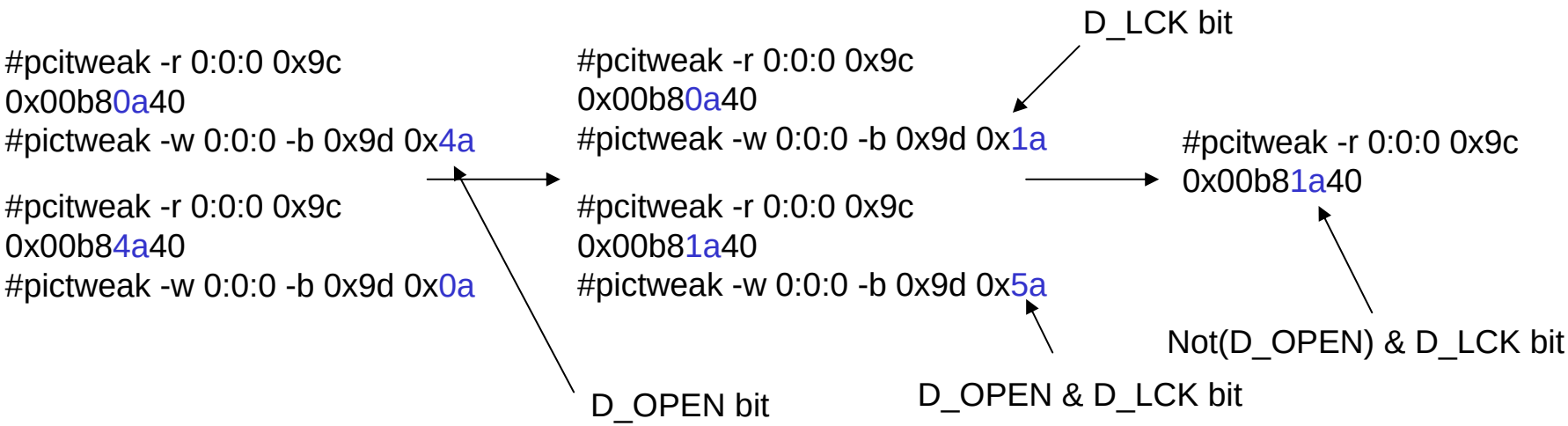
SMRAM security model

- System Management Mode code runs with full privileges on the platform (even more privileged than operating system kernels).
- There is a need to prevent access to the SMRAM when the system is not in SMM so that only the SMI handler can modify the content of the SMRAM.
- The rule is thus that legacy, high SMRAM and TSEG can only be accessed if the CPU is in System Management Mode unless the D_OPEN bit is set in the chipset.



SMRAM protection mechanism

- The main mechanism to prevent modification of the SMI handler is the D_LCK bit.
- If the D_LCK bit is set, configuration bits for the SMRAM in the chipset become read only (D_OPEN bit included).



Outline

- Introduction
- A (short) description of SMM
- Offensive use of SMM
  - Potential uses: privilege escalation schemes, rootkits
  - Limitations
- Circumventing the D_LCK bit
  - Memory caching
  - Cache poisoning
  - Applications and demo
- Impact
- Countermeasures and conclusion

Offensive use

- Privilege escalation schemes:
  - See CanSecWest 2006 presentation.
  - Actual privilege escalation schemes (restricted root to kernel, restricted X server to kernel for instance).
- Rootkits:
  - See Sparks, Embleton Black Hat 2008 presentation.
  - Rootkits can hide functions in the SMI handler (example of a keylogger).
- Bypass late launch restrictions on D-RTM based trusted platforms:
  - See Rutkowska and Wojtczuk Black Hat Federal 2009 presentation.

- From that point on, we will consider that the attacker is willing to conceal a rootkit in SMRAM.

Limitations

- SMM rootkits have strong limitations:
  - They do not survive a reboot of the platform.
  - It is difficult to design a generic SMM rootkit (SMM code is specific to each platform).
  - The strongest limitation (in my opinion): most recent platforms set the D_LCK bit at boot time preventing SMM modification.

- So far, no efficient way to bypass the D_LCK bit has been presented.

Outline

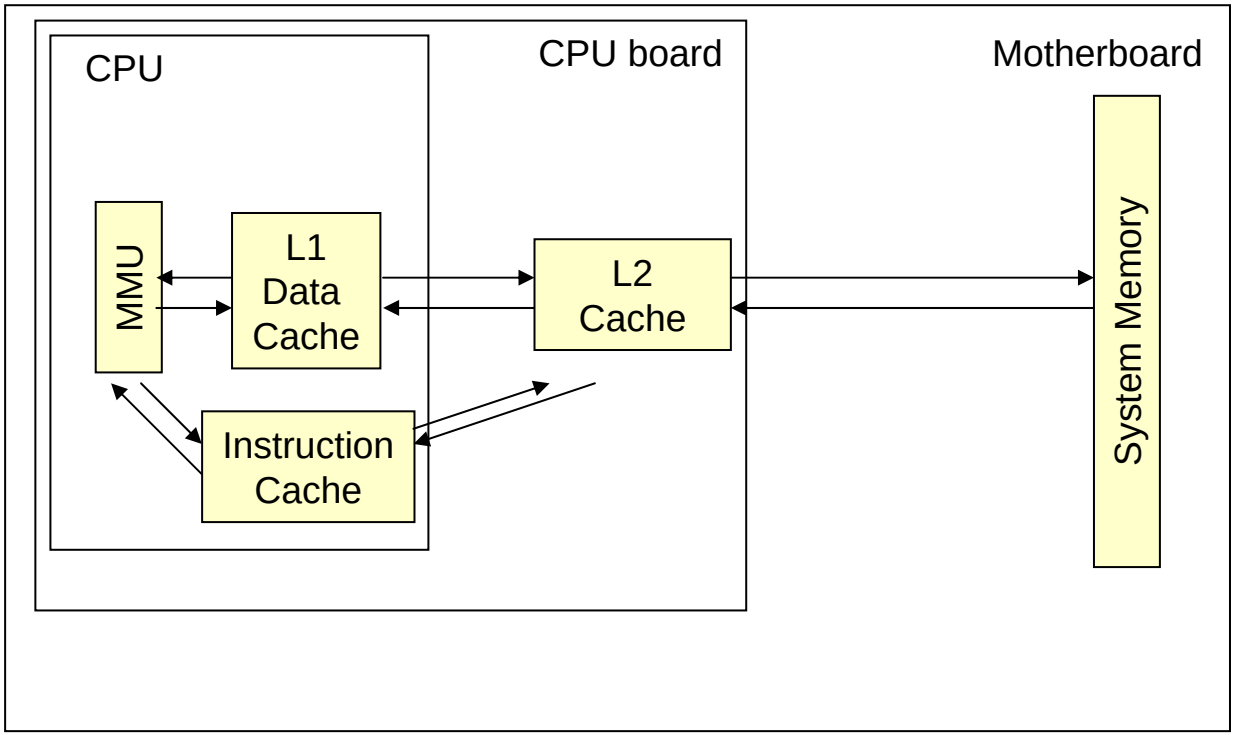
- Introduction
- A (short) description of SMM
- Offensive use of SMM
  - Potential uses: privilege escalation schemes, rootkits
  - Limitations
- Circumventing the D_LCK bit
  - Memory caching
  - Cache poisoning
  - Applications and demo
- Impact
- Countermeasures and conclusion

Circumventing the D_LCK bit: First idea, chipset translation mechanisms

- Chipset translation mechanisms modify physical memory mappings.
- Chipsets implement different translation mechanisms:
  - See Duflot and Absil PacSec 2007 presentation on the graphics aperture (GART) functionality.
  - See Wojtczuk and Rutkowska presentations on the Q35 chipset during Blackhat 2008.
- But some chipset translation mechanisms are obsolete (GART).
- Translations tables can be locked (use of lock bits similar to the D_LCK one).
- Need to come up with another (and better) idea:
  - Cache poisoning.

Cache hierarchy

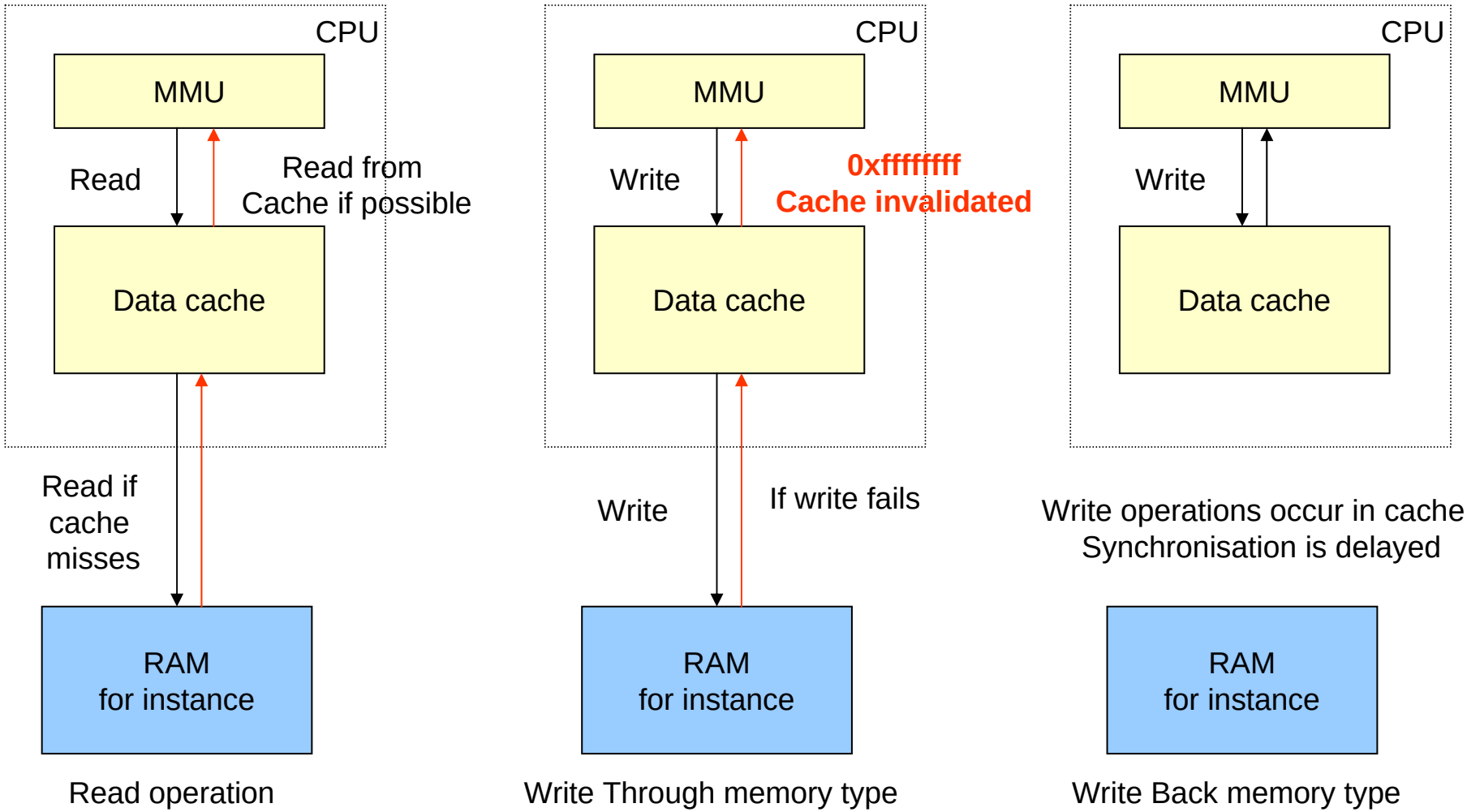
- To speed up memory accesses, caching is used.
- Description of the cache hierarchy of a x86 processor (example):



Memory caching

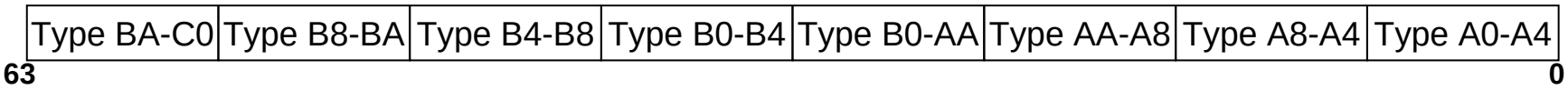
- There is a need to synchronise CPU caches with memory. Different memory caching strategy (memory types) can be specified, for instance:
  - WB: write back.
  - WT: write through .
  - UC: not cacheable.
- Which memory zones are cached and which are not is specified in the memory management unit of the CPU (responsible for the translation between logical, virtual and physical addresses).
- Two different mechanisms to specify memory caching strategies: Page directories and tables (the hard way) and MTRR (the easy way).

Cached memory types

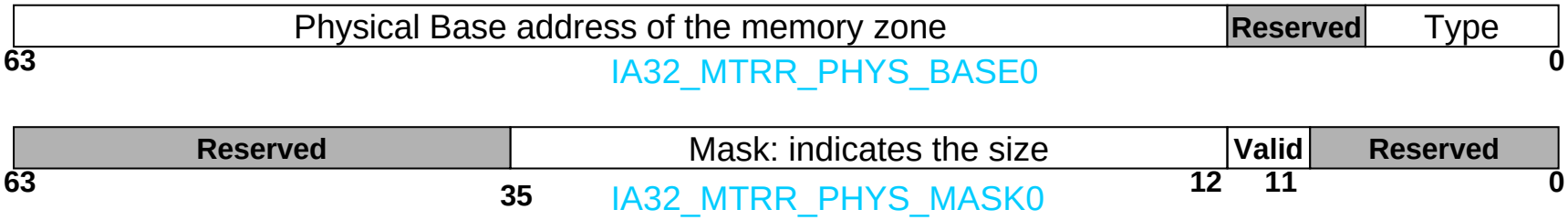


Use of MTRRs

- MTRRs (Memory Type Range Registers) are Model Specific Registers (MSR). There are two different types, fixed and variable.
- “Fixed MTRRs” can be used to specify the caching strategy of legacy memory areas used by the BIOS for instance.
- “Variable MTRRs” can be used to specify the caching strategy of any physical memory zone.
- Structure of a fixed MTRR (example MTRR_FIX16K_A0000)



- Structure of a variable MTRR (example (MTRR_PHYS_BASE0)



Premier ministre
Secrétariat général de la défense nationale

So…

- The access control point is in the chipset and the chipset does not “see” what happens inside the cache.
- Code running on the CPU can decide the caching strategy.
- Plus, the chipset does not even know where the SMRAM really is (SMBASE only known to the CPU).

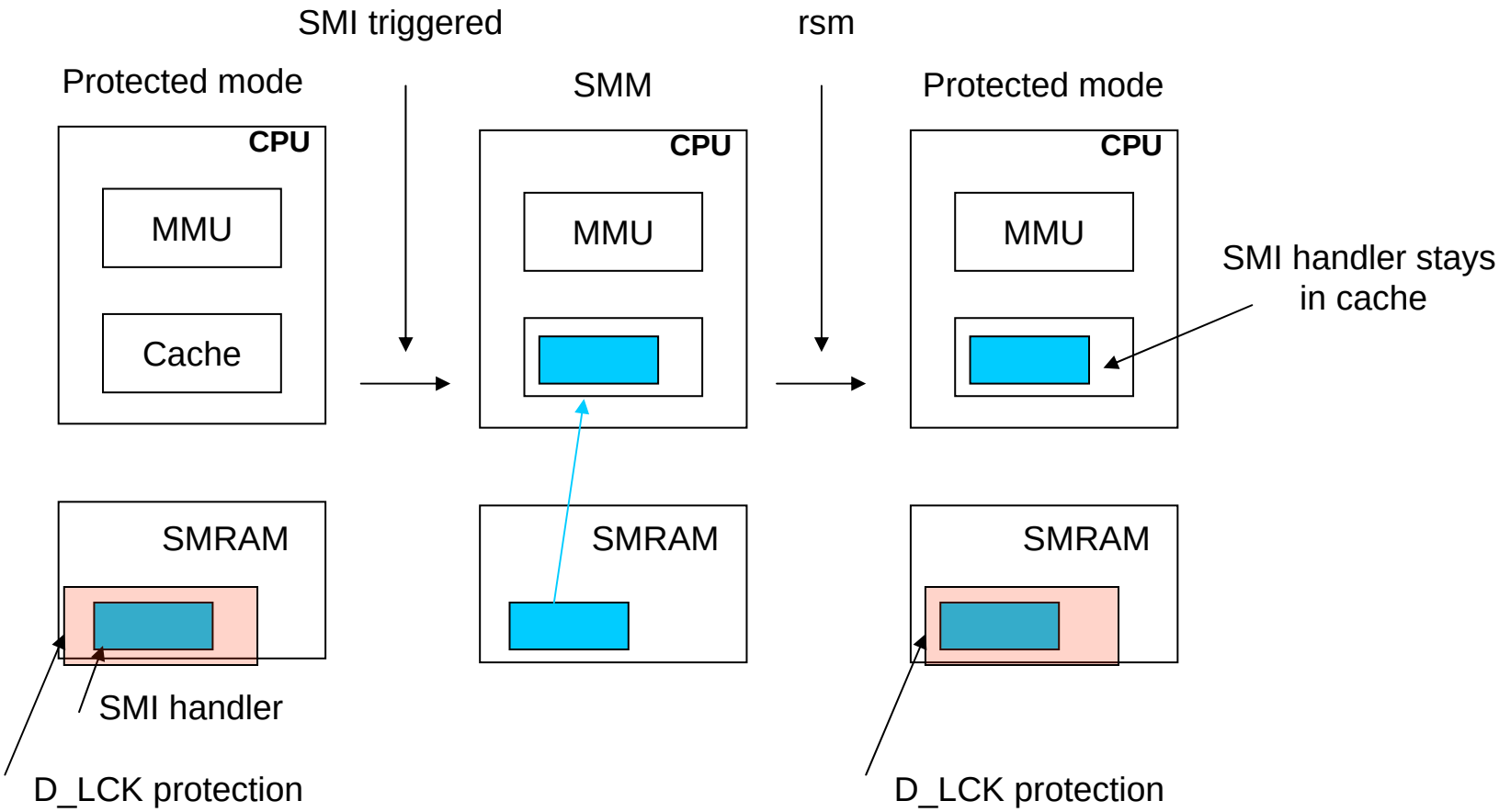
- Isn’t there a coherency problem here?

SMRAM and caching

- It is advised that SMRAM should not be cached especially when the SMRAM address space conflicts with other address spaces (legacy SMRAM).
- Exception: it is explicitly stated in chipset documentation that high SMRAM (0xfeda0000) can be cached.

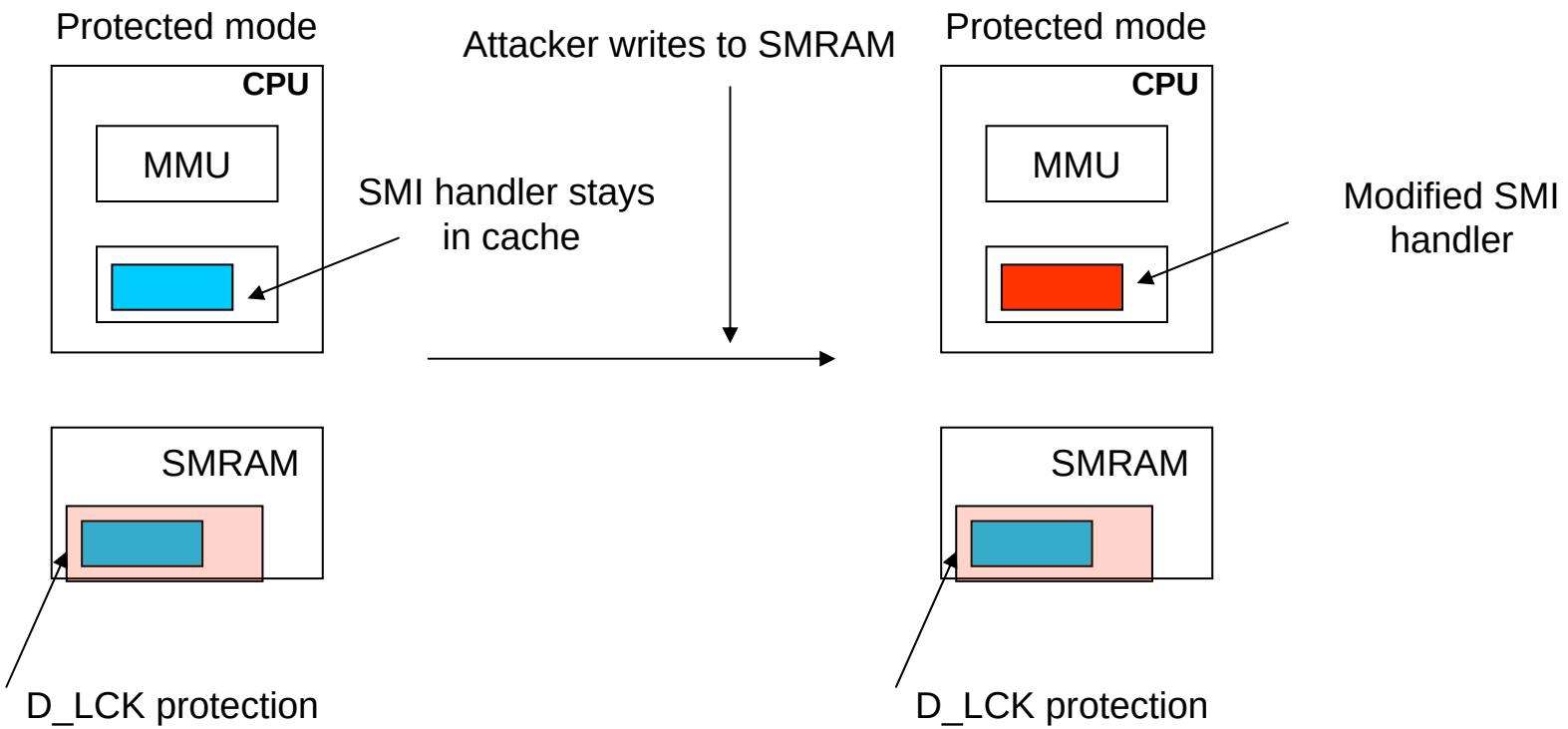
- Let's assume that the SMRAM memory zone is cached in WB by the CPU. If the SMI handler is executed, it will be "copied" into the CPU instruction and data caches.
- If SMM handlers do not flush caches when they give the hand back to the operating system, it is likely that the SMI handlers will linger (for a very small time) in the data cache of the CPU.

Basic idea: SMI handler stays in cache



Basic idea: attacker writes to the SMRAM

- When the CPU is not in SMM, the CPU cannot write in SMRAM. But if the SMRAM is cached in Write Back mode, the CPU only writes to the cached version and not in memory.



Scheme to circumvent the D_LCK bit: cache poisoning

- We assume that a rootkit wants to conceal some functions within the SMRAM but the D_LCK bit is set and that as a consequence SMRAM cannot be accessed except in SMM.
- The attacker has to modify the caching strategy of the SMRAM location (example if SMBASE=0xa0000).

```
__asm__ volatile(
            "movl $0x06060606, %eax\n"
            "movl $0x0, %edx\n"
            "movl $0x259, %ecx\n"
            "wrmsr\n"
);
```

May be skipped in practice

- Trigger an SMI. The SMI handler will be cached by the CPU.

```
outl(0x0000000f, 0xb2);
```

- Modify the memory content at SMRAM address (only the cached image is modified).

```
vidmem  =  mmap(NULL, 0x8000, PROT_READ | PROT_WRITE, MAP_SHARED,
             fd, 0xa8000);
memcpy(vidmem, handler, endhandler-handler);
```

Scheme to circumvent the D_LCK bit: cache poisoning (2/2)

- Trigger a SMI. This time the modified image is executed from the cache.

```
outl(0x0000000f, 0xb2);
```

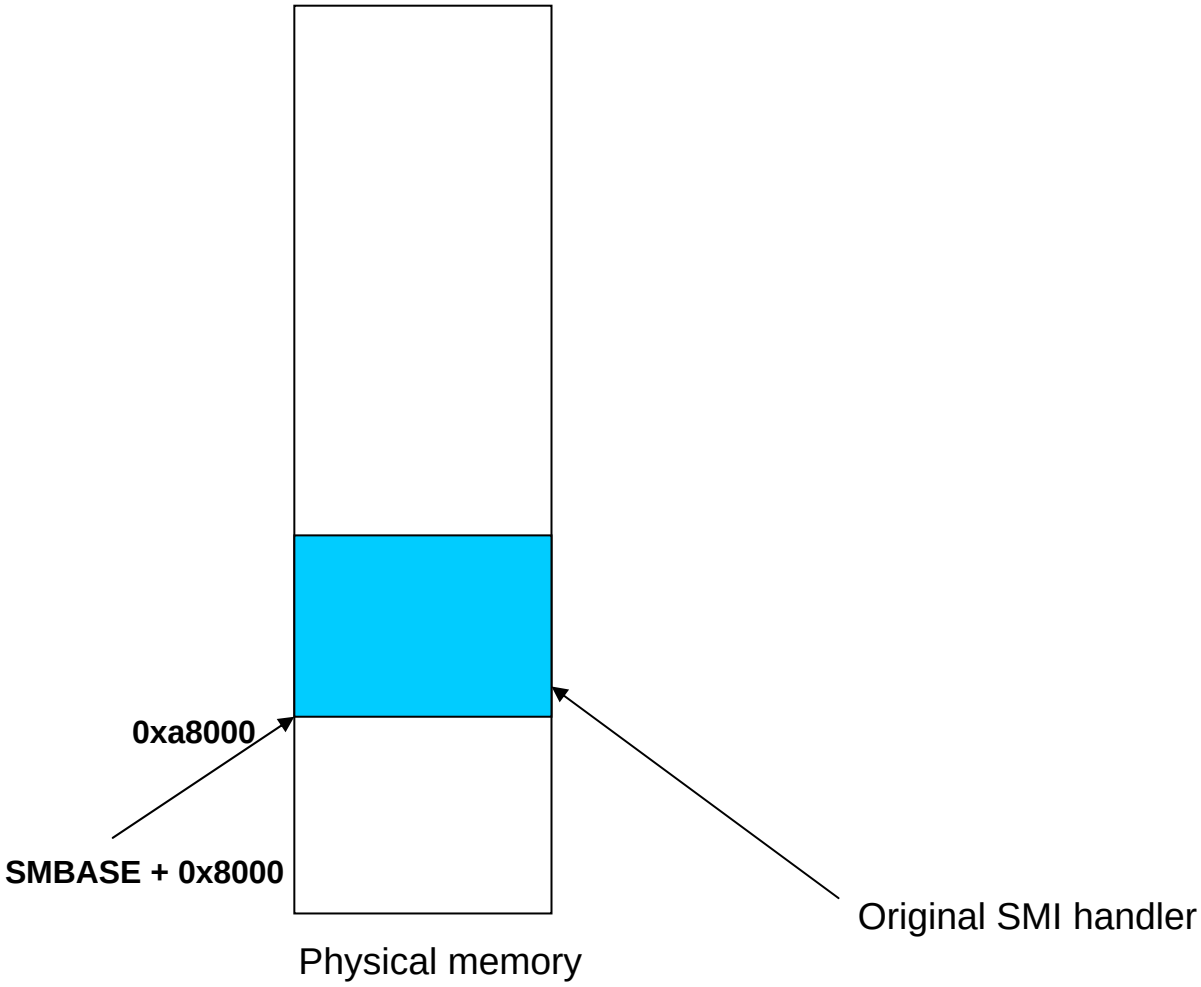
- Objection: But wait! Only the data cache is modified, not the instruction cache, so the modification should have no effect.
  - True, but the instruction caches will probably be flushed during mode transitions as running 16-bit instructions in a 32-bit (or 64-bit) mode should not be advised. In that case, instructions are reloaded from… the L1 data cache.
- Conclusion: modification of the SMI handler succeeded even though the D_CLK bit is set.

No way!

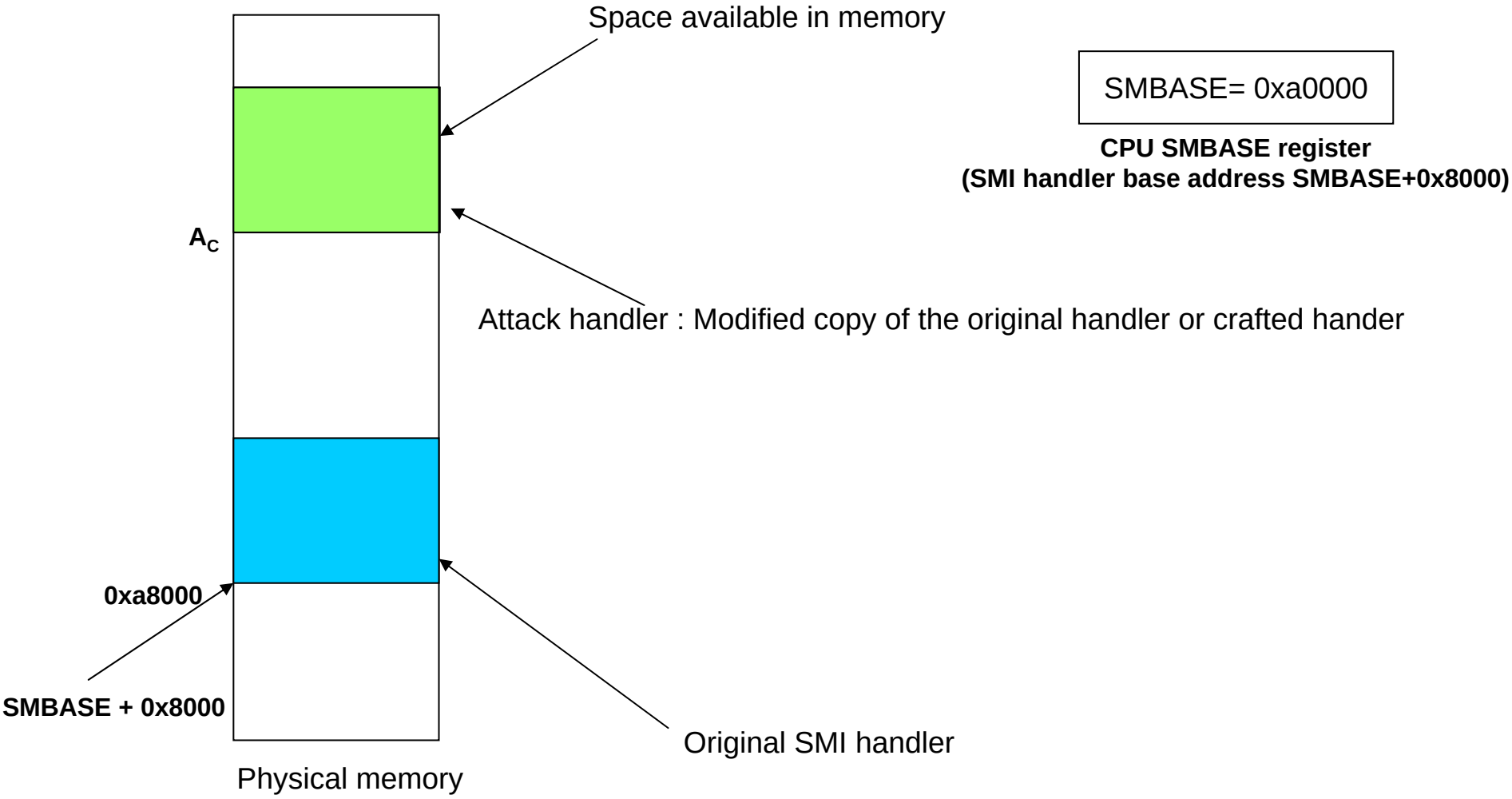
- The SMI handler can flush the cache before exiting!
  - No SMI handler that I have seen does that.
  - Anyway if the handler did so it would not be a major problem as cache flushing only ensures SMI handler confidentiality (not integrity).
- Data caches are small, the whole SMRAM won't probably fit in it!
  - True.
- Data are not bound to stay for too long in the data cache so the attacker needs a way to either:
  - Ensure that the SMI handler stays permanently in the data cache (periodic accesses to the handler for instance).
  - Or find a way for the SMI handler to permanently stay in memory (not only in cache).

- Overall, we need to rethink our attack scheme for it to be usable.

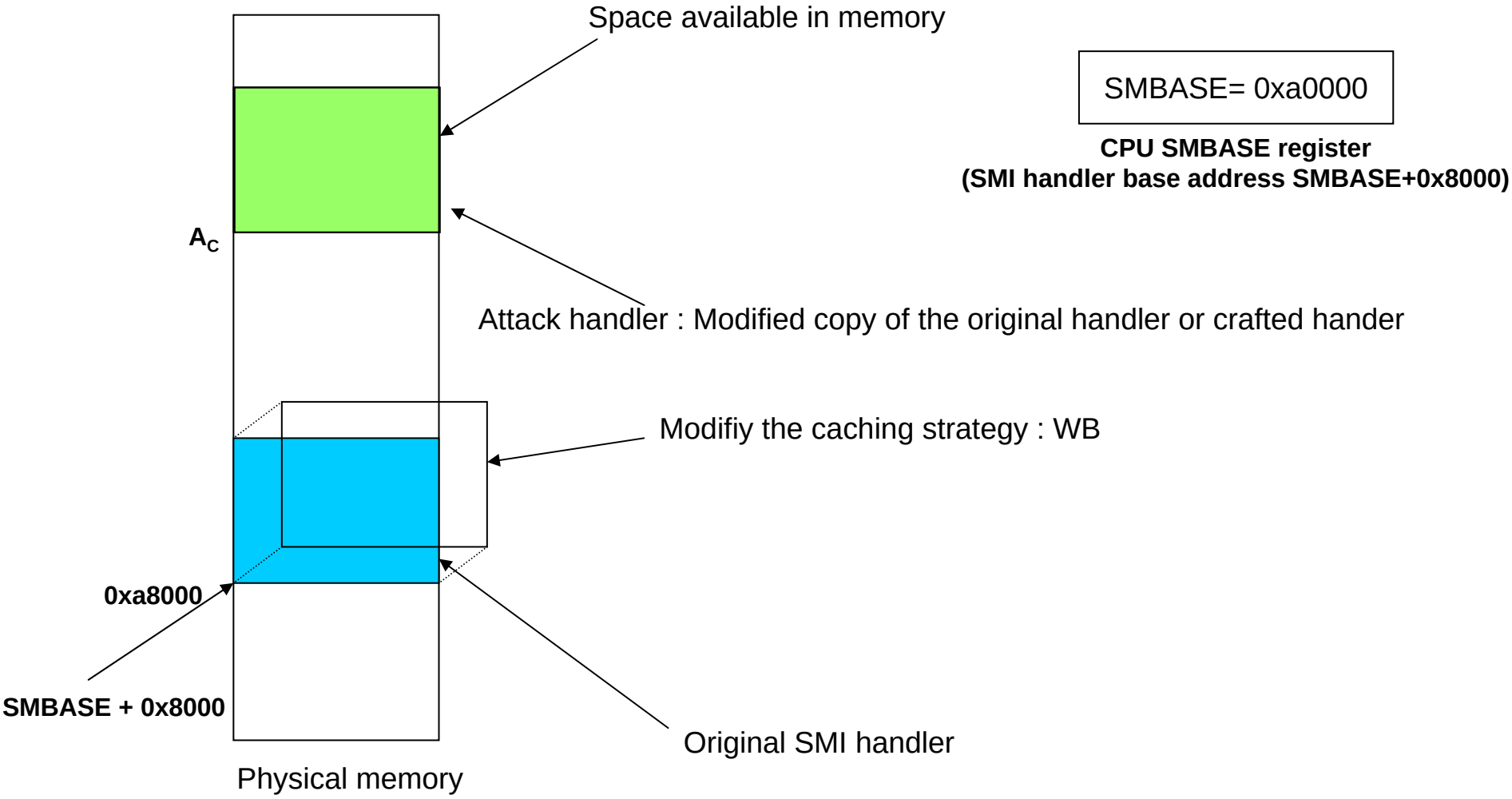
A more efficient scheme (example for legacy SMRAM)



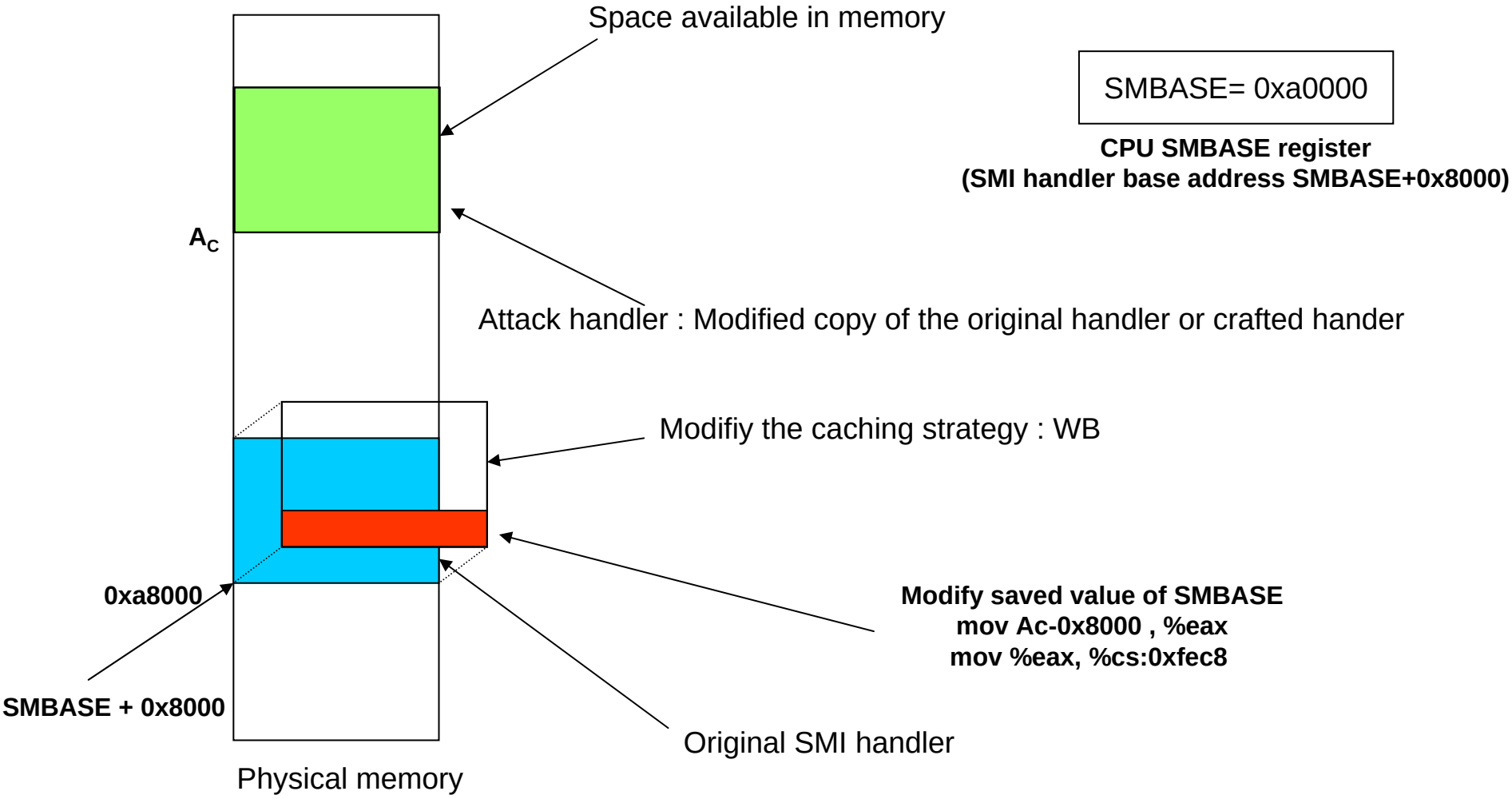
A more efficient scheme (example for legacy SMRAM)



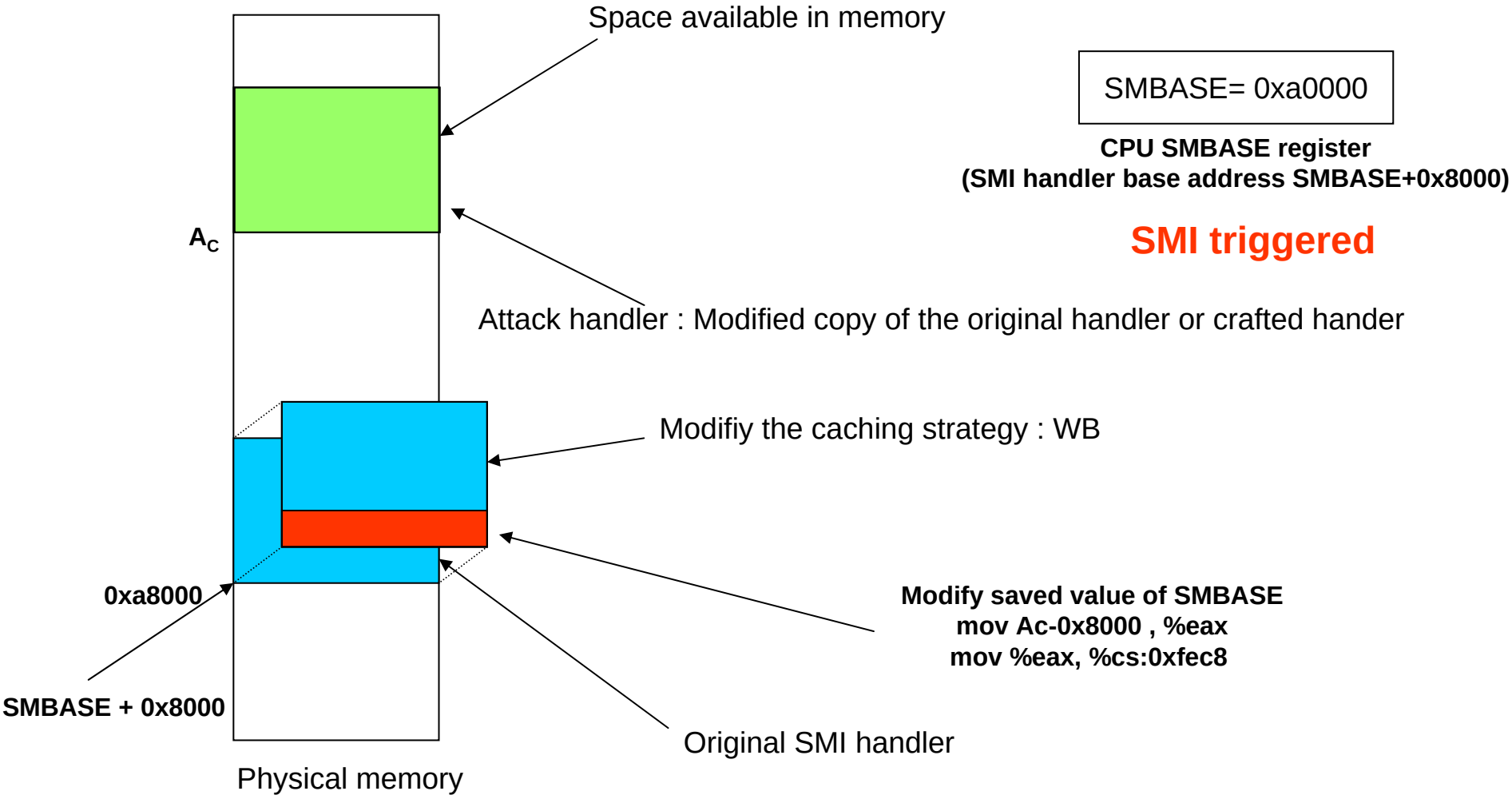
A more efficient scheme (example for legacy SMRAM)



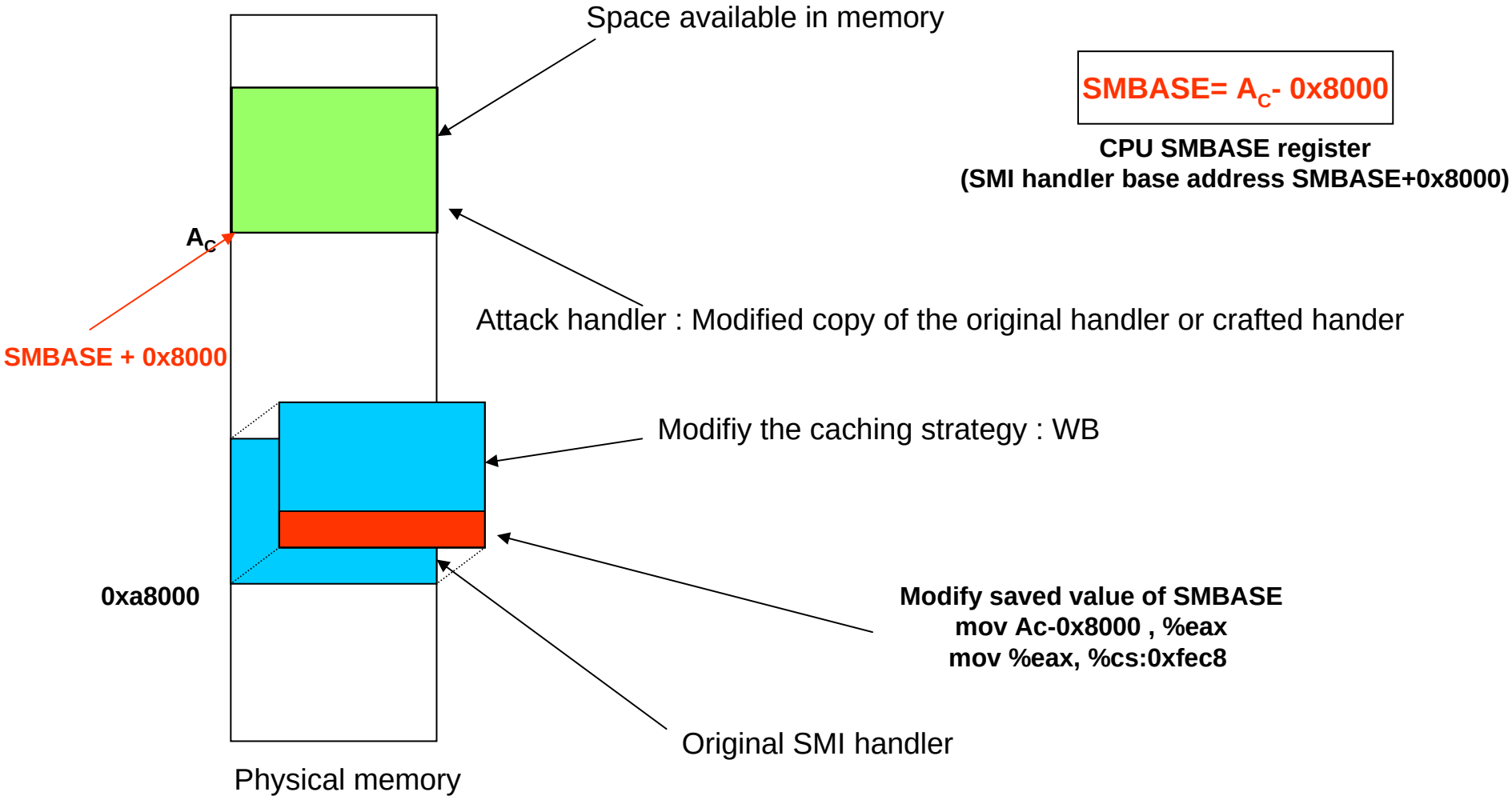
A more efficient scheme (example for legacy SMRAM)



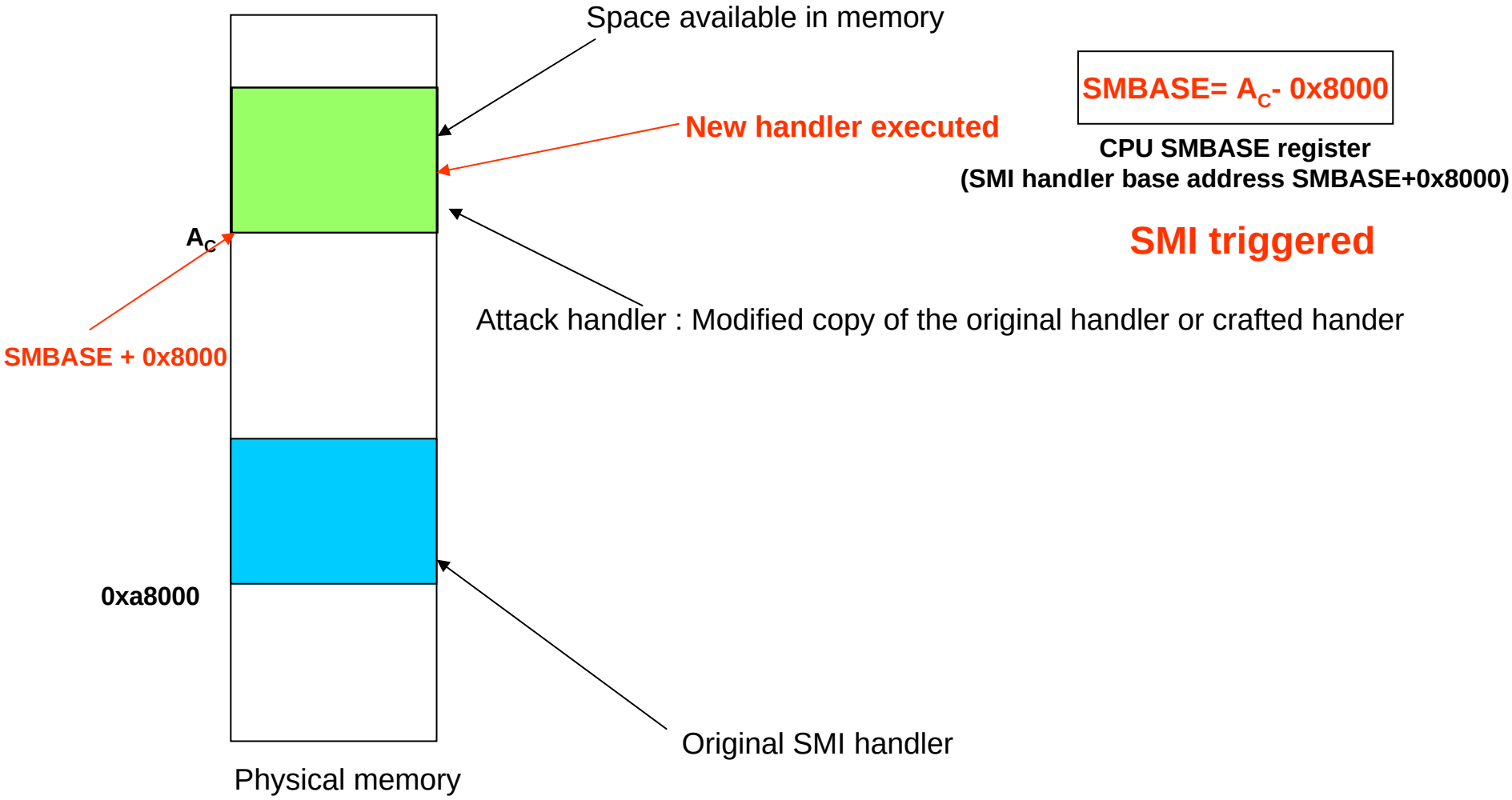
A more efficient scheme (example for legacy SMRAM)



A more efficient scheme (example for legacy SMRAM)



A more efficient scheme (example for legacy SMRAM)



A more efficient scheme

- Modify the caching strategy of the SMRAM (same as before).
- Copy the “attack” SMI handler in a free (unused) RAM location. We will call C this copy and A_C the address of the copy.
  - For instance by copying it from the data cache after an SMI has been triggered.
  - Or a valid SMRAM handler can be crafted.

```
fd = open(MEMDEVICE, O_RDWR);
vidmem  =  mmap(NULL, 0x8000, PROT_READ | PROT_WRITE, MAP_SHARED,
           fd, 0x38000);
close(fd);
memcpy(vidmem, handler, endhandler-handler);
```

- Modify the original handler O (in cache). The handler should modify the SMBASE value in the save state of the CPU so that SMBASE = A_C – 0x8000. The modification is small (less than 20 bytes and fits in the cache).
  - By crafting a new handler that will do that.
  - By hooking the original handler.
- We call M this modified handler.

That’s the only time when the cache is actually needed

A more efficient scheme (2/2)

- Trigger an SMI. M is executed. Upon execution of the “rsm” assembly language instruction, the SMBASE register will be set to A_C – 0x8000.

- When the next SMI is triggered, the CPU will determine that the new SMRAM location is SMBASE +0x8000 = A_C, and the “attack” SMI handler “C” will be executed from memory.

- The thing is, this area is not at all protected by the D_LCK bit. So the attacker can freely modify the new SMI handler C, even though the D_LCK bit is set.

Stealth

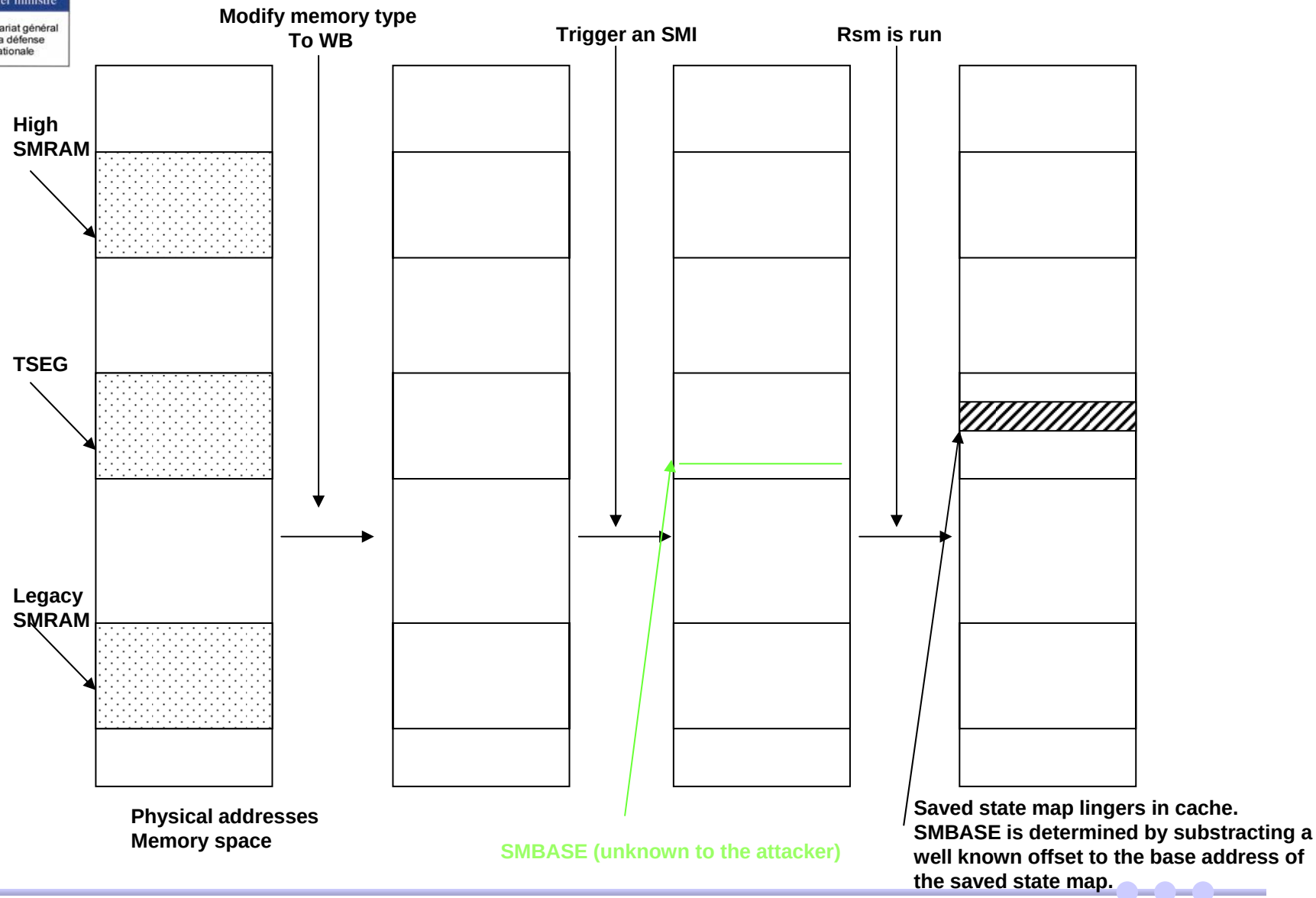
- If a write back cycle occurs, M will be written to SMRAM, and O will be overwritten.
- But if the attacker invalidates cache lines where M is stored (using the “clflush” or “invd” assembly language instructions for instance), relocation occurs without any write back cycle to SMRAM being issued.
- This way, O is not modified at all during the course of the relocation.

- OEM SMI handler code did not change but is not used any more should an SMI be triggered.

Difficulties

- The scheme requires modification of the CPU caching strategy and thus requires ring 0 privileges (only useful to kernel level rootkits).
- It requires the attacker to locate the SMI handler (i.e. determine SMBASE, explanations later).
- On multi-CPU architectures each CPU may use a different SMRAM and thus a different SMI handler. Which of them will be called when an SMI is triggered is not specified in the specifications.
    - The attacker will have either to modify all of them,
    - or to modify only one of them and wait until this particular one is run.

Guessing SMBASE: methodology



Summary

- It is possible for an attacker with sufficient privileges (i.e. a kernel rootkit given the fact that the attack requires modifying MMU structures and CPU cache strategy), to modify the content of the SMRAM even though the D_LCK bit is set.

- We tried our scheme on different machines from different manufacturers (laptops, desktops, servers using either TSEG, high or legacy SMRAM) and it worked against all of them.

Premier ministre
Secrétariat général de la défense nationale

Why did it work?

- System Management Mode is a CPU mode. The CPU determines which code runs in SMM.
- The “D_LCK bit” is a chipset security mechanism. This mechanism can only protect memory that is accessible from the chipset.
- Only the CPU knows what is stored inside SMBASE i.e. where SMRAM really is. The chipset can only protect the memory zone where it “thinks” SMRAM is.
- The chipset only knows the CPU is in SMM because the CPU is telling it is.

- Is all this coherent? The security model seems to be flawed: two different mechanisms to circumvent this particular security mechanism have been proposed so far.

Outline

- Introduction
- A (short) description of SMM
- Offensive use of SMM
  - Potential uses: privilege escalation schemes, rootkits
  - Limitations
- Circumventing the D_LCK bit
  - Memory caching
  - Cache poisoning
  - Applications and demo
- Impact
- Countermeasures and conclusion

Impact of the scheme

- The scheme we proposed can be used by an attacker to overload any chipset protected area.
    - System Management RAM.
    - System ROM (BIOS for instance).
- PCI integrity scanners or chipset-based integrity scanners such as DeepWatch (proposed by Yuryi Buligin during Blackhat 2008) can be fooled :
    - DeepWatch is a generic chipset-based integrity scanner that can be used to monitor SMRAM integrity.
    - Deepwatch can check whether the SMRAM is modified.
    - But DeepWatch cannot know what SMBASE is so SMRAM relocation using our scheme will be invisible to DeepWatch.
    - DeepWatch will keep on checking the integrity of the memory location where it “thinks” SMRAM is when the attacker has defined another handler somewhere else in memory.

Impact on HyperGuard

- The goal of HyperGuard is to include security functions in the SMI handler.
- Joint work (under progress) between “the Invisible Thing” and “Phoenix BIOS”.
- Presented during the Blackhat 2008 forum by Joanna Rutkowska and Rafal Wotjczuk.
- The scheme we propose can be used by an attacker to remove Hyperguard in the relocated SMI handler.
- Efficient even when HyperGuard and DeepWatch are combined!

Outline

- Introduction
- A (short) description of SMM
- Offensive use of SMM
  - Potential uses: privilege escalation schemes, rootkits
  - Limitations
- Circumventing the D_LCK bit
  - Memory caching
  - Cache poisoning
  - Applications and demo
- Impact
- Countermeasures and conclusion

Countermeasures

- The only efficient solution is for CPU designers to modify the CPU.
- Indeed, platform manufacturers could:
  - Ensure that the SMI handler flushes data caches before exiting (each “rsm” instruction should be preceded by a cache flush instruction). But this only ensures SMI handler confidentiality…
  - Or randomly relocate SMRAM at boot time. This is not really efficient (security by obscurity).
- What can the end user do?
  - Prevent the system from being taken over by rootkits (that is the best workaround…).

From the vendors perspective

- Intel® has been contacted (September 2008) and acknowledged (private communication) that the problem was generic.
- However, they stated that they noticed the problem earlier and solved the problem in the Conroe CPU timeframe (2007-2008) with a CPU modification. Patents have been filed.
- But completely solving the problem requires OEMs to take advantage of the new CPU mechanism, which has not been done that much so far (according to Intel® some of them already did).
- Intel® expects machines from different OEMs to be protected first trimester 2009 (in the Intel® Nehalem CPU timeframe).
- But what about the billions (?) of machine already shipped?

Overall conclusion

- As a conclusion, we showed that it was possible for a rootkit to conceal functions in the SMI handler even on recent machines when the D_LCK bit mechanism is correctly used.
- The proof of concept schemes exploit a global design flaw in the repartition of security functions between chipsets and CPUs.

- The only efficient solution against the problem is CPU modification (recent CPU are already modified).

Premier ministre
Secrétariat général de la défense nationale

Thank you for your attention
Any questions?

Spécial thanks to:
Olivier Grumelard (SGDN/DCSSI)
Olivier Levillain (SGDN/DCSSI)
Benjamin Morin (SGDN/DCSSI)
And the Intel Security Incident Response Team

See also (independent research work):
http://theinvisiblethingsblogspot.com/ (Joanna Rutkowska and Rafal Wojtczuk)

Contact address:

loic.duflot@sgdn.gouv.fr