System Management Mode Design and Security Issues

Loïc Duflot, Olivier Levillain
Benjamin Morin and Olivier Grumelard

French Network and Information Security
Agency (ANSSI)
SGDSN/ANSSI 51 boulevard de la Tour Maubourg 75007 Paris

Premier ministre
Secrétariat général de la défense nationale

Introduction

- The goal of this presentation is to explain why security should be taken into account during the early steps of any new hardware or software technology design.
- We consider the example of System Management Mode, one of the legacy modes of operation of modern x86 and x86-64 CPUs.
- System Management Mode is a highly privileged mode, and running code in SMM is an easy and stealthy way for attackers to own a machine.
    - Can be used to bypass security restrictions on trusted computing enabled machines;
    - Can be used by rootkits to conceal functions.
- At the time System Management Mode was specified, security was not taken into account. When it later became obvious that security was mandatory, security layers were added.
- In this presentation, we show that such an approach is flawed.

Introduction

- We present several design flaws allowing an attacker to run arbitrary code in SMM:
  - First, we give a comprehensive summary of design flaws already discussed in security conferences.
  - Then we show how security layers were added to prevent vulnerabilities from being exploited.
  - We also introduce design flaws never discussed before and analyze whether they are exploitable by attackers or not.
- We try to analyze the current situation, and what can be done at this point:
  - Is it still possible to achieve true SMM security?

- In this presentation, examples are given for x86-32 CPU-based platforms but can be easily transposed to x86-64 architectures.

Outline

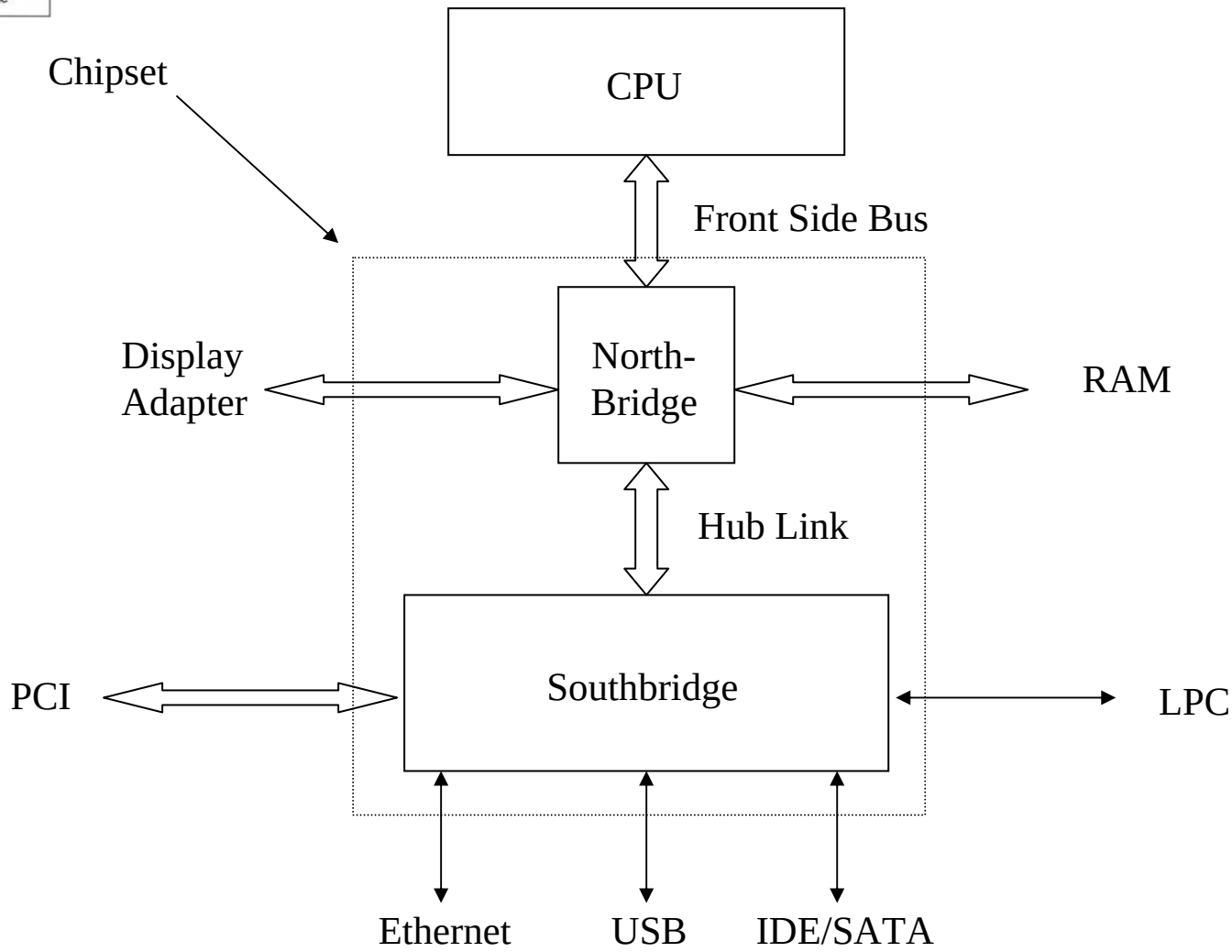
- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

Premier ministre
Secrétariat général de la défense nationale

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

Simplified PC architecture



System Management Mode (SMM)

- Used for maintenance:
  - Motherboard control, power and thermal management.
  - Sometimes called by ACPI.
- 16-bit mode:
  - Free access the whole memory space.
  - No memory protections available at the hardware level (no paging, no segmentation, segment limits are ignored).
  - Free access to I/Os.
- SMM code runs when the operating system is “frozen”:
  - OS does not even notice it is interrupted by software running in SMM.
  - OS is unable to enforce its security policy.
  - **Arbitrary code running in SMM has complete control over the machine.**

Offensive use of SMM code

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

- Bottom line: the machine has to protect SMM code from attackers and thus even from the operating system itself.

Entering SMM

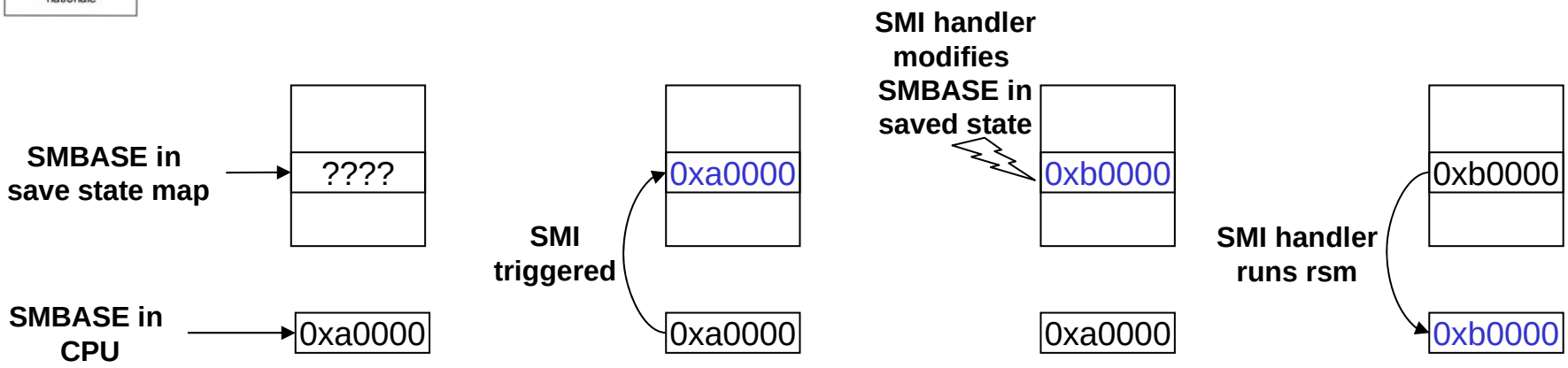
- System Management Interrupts (SMI):
  - Hardware interrupts generated by the chipset (Northbridge).
  - **SMM mode is only entered when CPU receives an SMI.**

- Triggering an SMI:
  - Writing to the Advanced Power Management Control Register (APMC).
    - `outl(something, 0xb2)`
  - Requires input/output privileges.

SMRAM

- SMI handlers are executed from SMRAM.
- SMRAM is a memory area located in RAM.
  - Upon entry in SMM, almost every CPU register is saved in a “memory saved state map” located in SMRAM.
  - SMI handler restores the CPU state from the memory saved state map with rsm assembly language instruction.
- SMRAM location
  - Specified by the SMBASE register of the CPU.
  - SMBASE cannot be accessed at all but is saved in memory saved state map (like other CPU registers).
  - So, it can be changed when CPU state is restored, but only by the SMI handler.

Location of the SMRAM



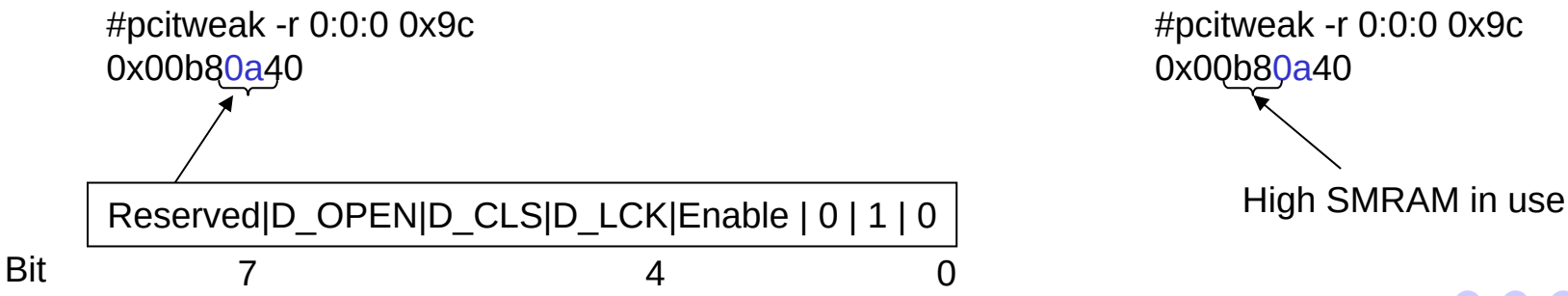
- In practice SMBASE is usually :
  - 0xa0000: legacy SMRAM location.
  - 0xfeda0000 (+/- 0x8000): high SMRAM location.
  - Something else: TSEG (Extended SMRAM).
- The base address of the SMI handler is SMBASE + 0x8000 (fixed offset).

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

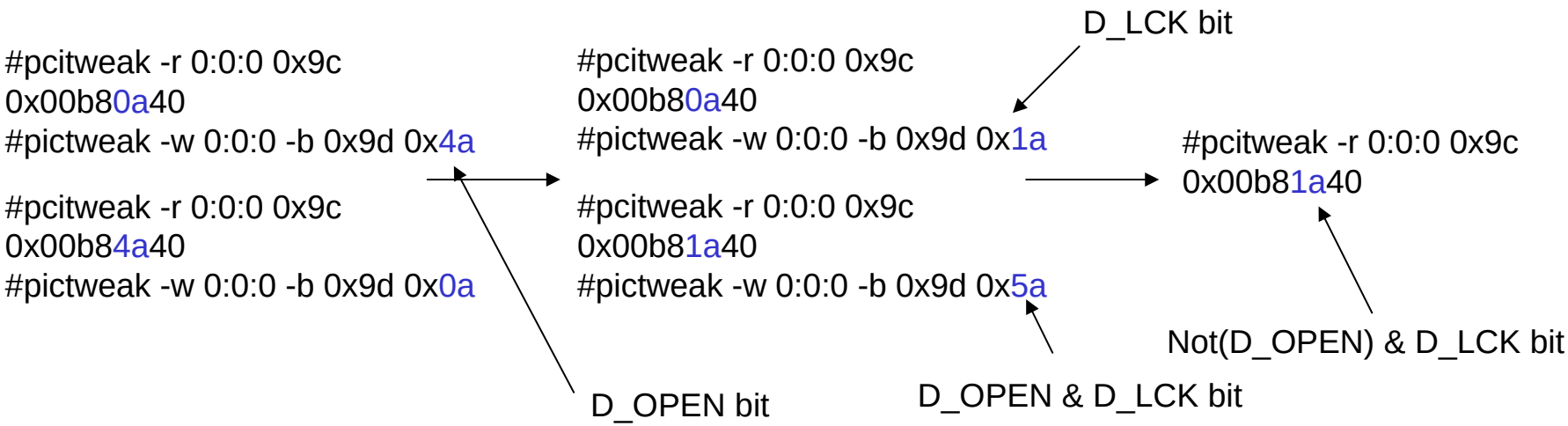
SMRAM Protection

- Initially, no protection at all:
  - SMRAM was just located in RAM at physical address 0x30000.
  - Trivial for a kernel-level rootkit to modify SMRAM content.
- Enhancement :
  - Prevent access to SMRAM when the system is not in SMM.
  - Only the SMI handler can modify the content of SMRAM.
- Access control rule
  - Legacy SMRAM (and later 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 modifications 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).



- Theoretically, this mechanism is an efficient access control mechanism.

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes.
- Reaching SMM security
- Conclusion

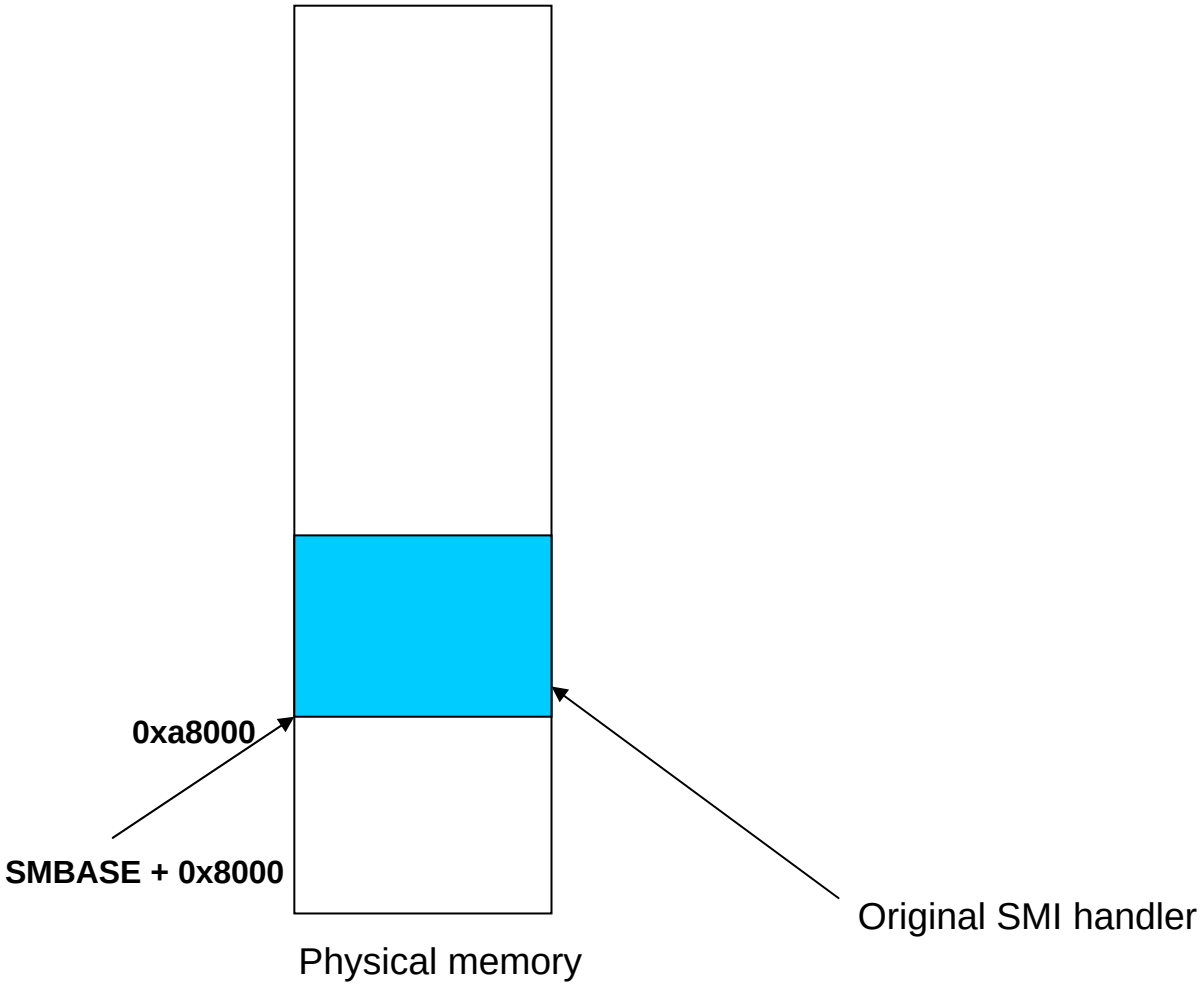
Design flaw 1: bad distribution of security functions between chipset and CPU

- The access control checkpoint is in the chipset:
  - D_LCK and D_OPEN mechanism.
- But only the CPU knows where the SMRAM actually is:
  - SMBASE is only known to the CPU.
- The chipset can only protect the memory location where the SMRAM is supposed to be:
  - The chipset will only protect legacy SMRAM, high SMRAM and TSEG.
- SMM code is run on the CPU:
  - As a consequence, the chipset is not an efficient access control point.

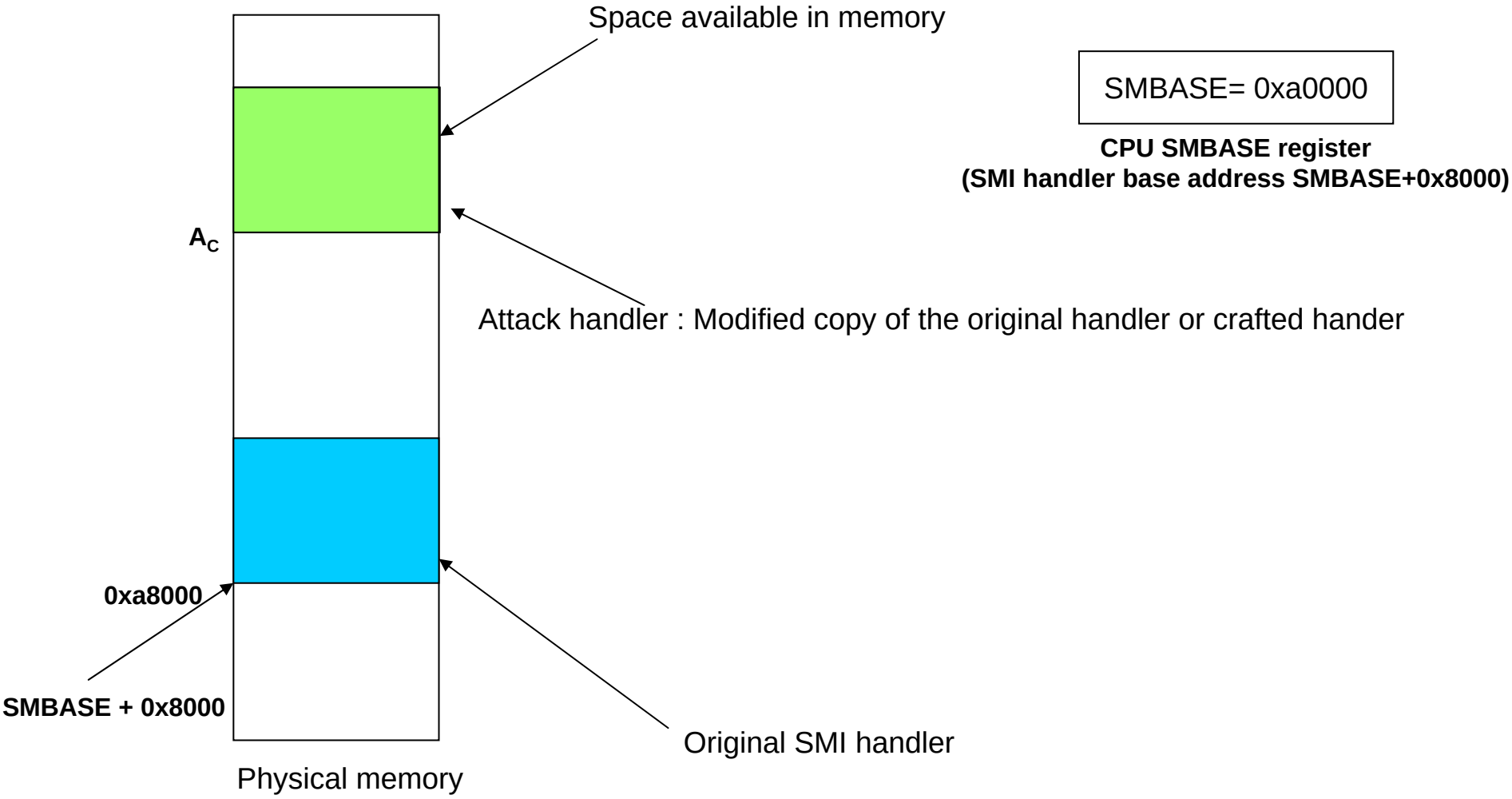
Cache poisoning attacks (2009)

- Code running on the CPU can determine how SMRAM will be cached.
- If the SMRAM is cached in Write Back, the attacker can run code from the cache.
- SMM code is run without the chipset being able to enforce the security policy.
- See:
  - Our Cansecwest 2009 presentation.
  - Joanna Rutkowska and Rafal Wojtczuk (http://theinvisiblethingsblogspot.com/).

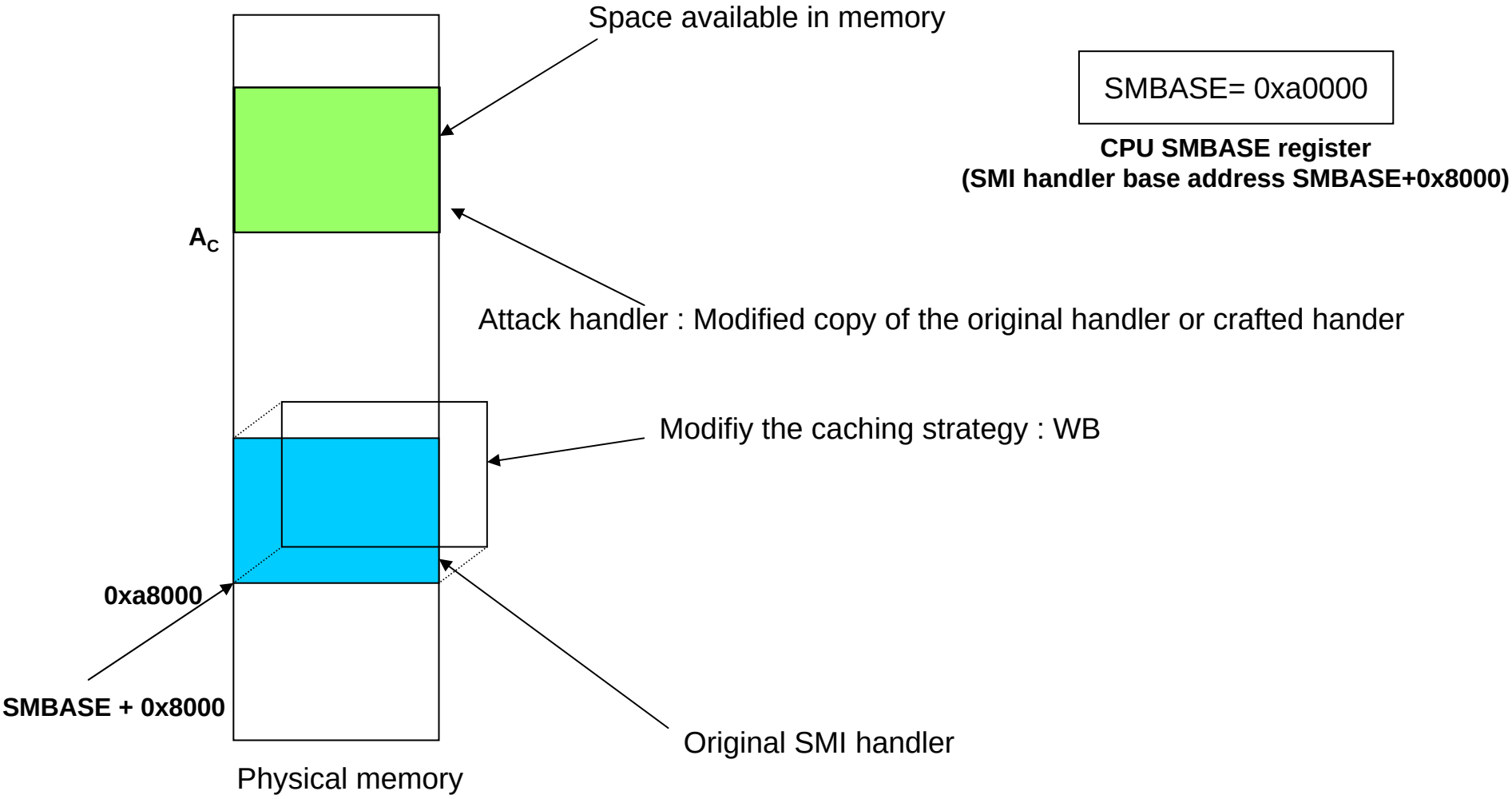
Attack scheme (example for legacy SMRAM)



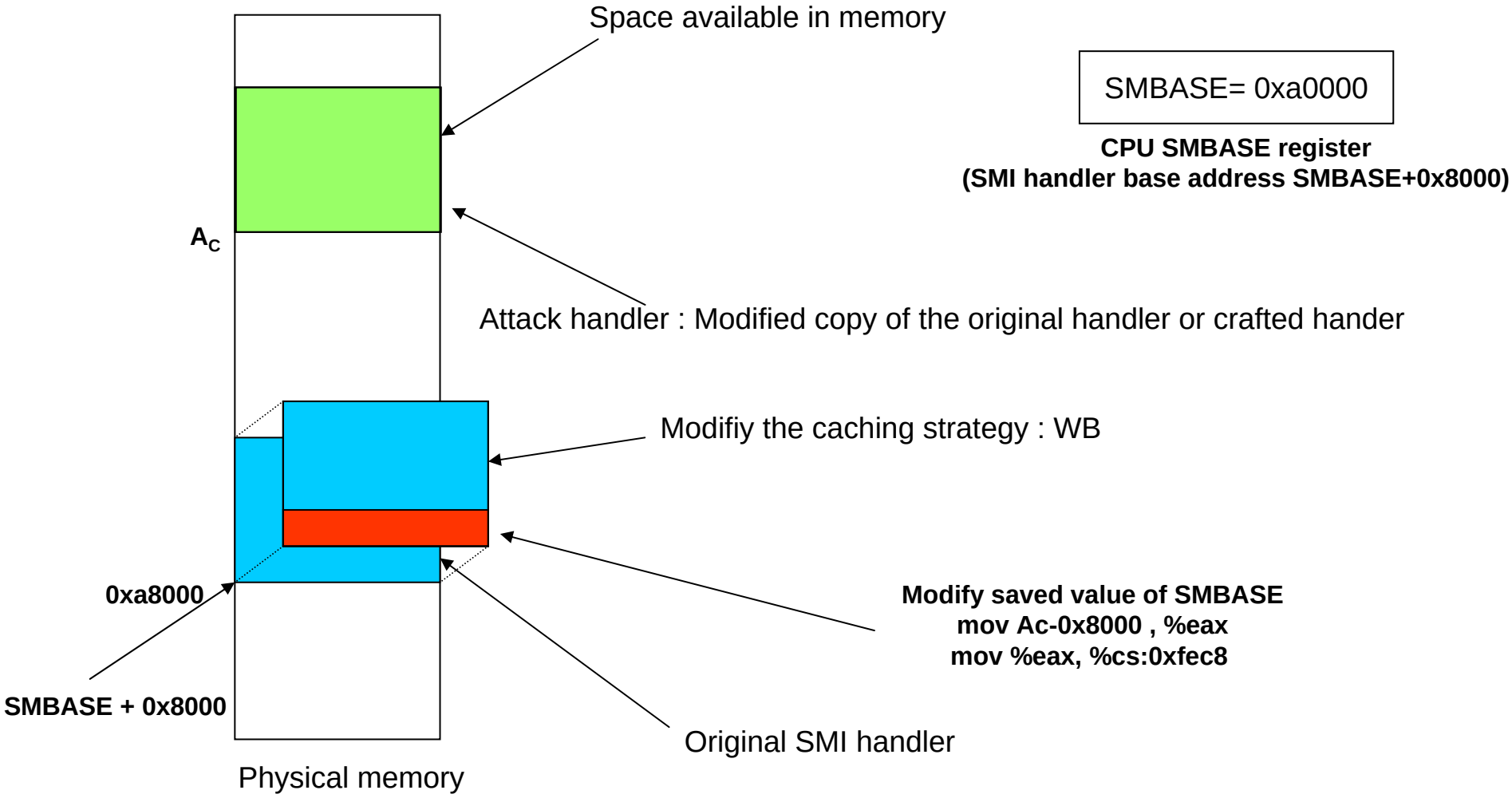
Attack scheme (example for legacy SMRAM)



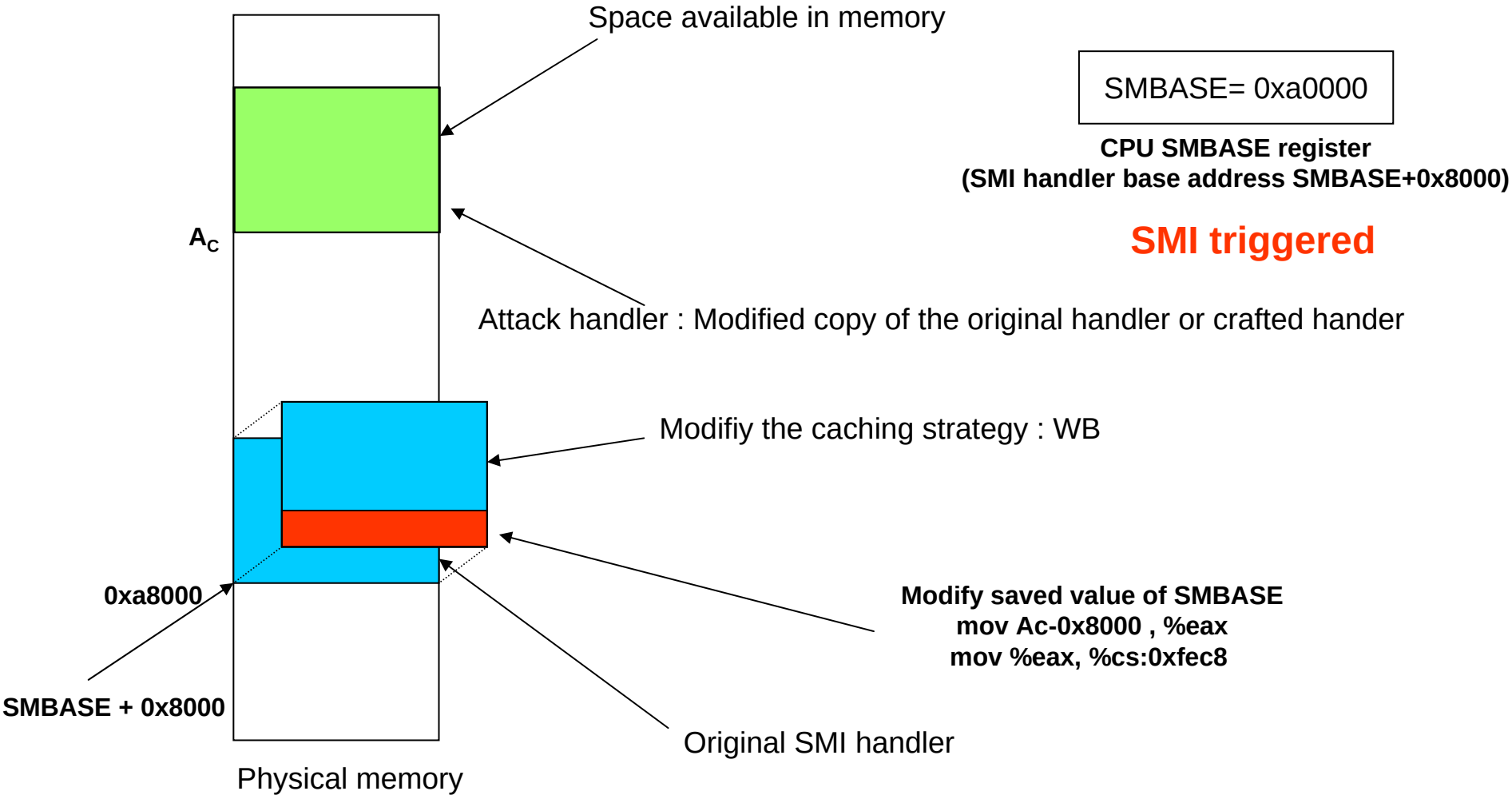
A attack scheme (example for legacy SMRAM)



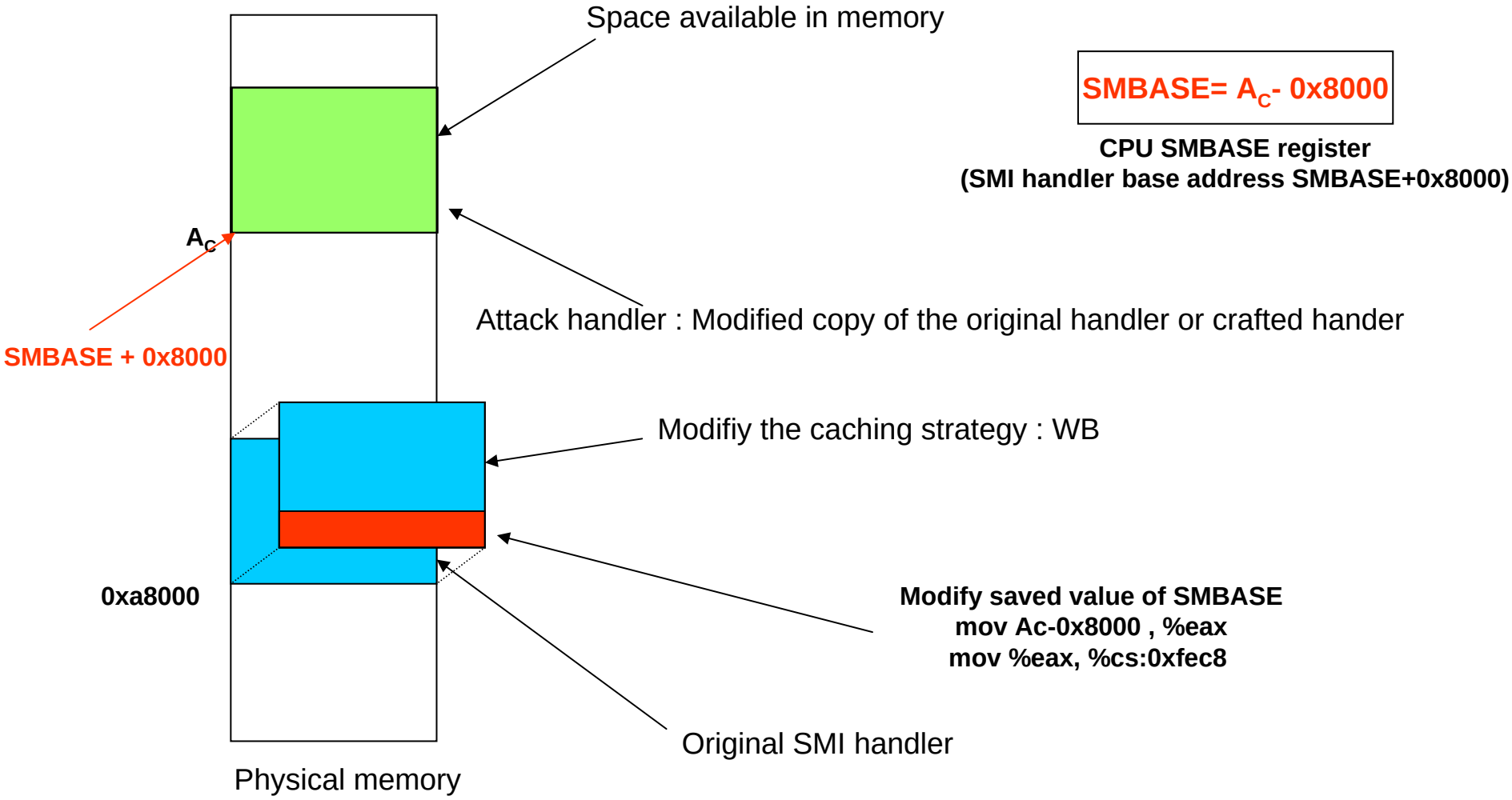
Attack scheme (example for legacy SMRAM)



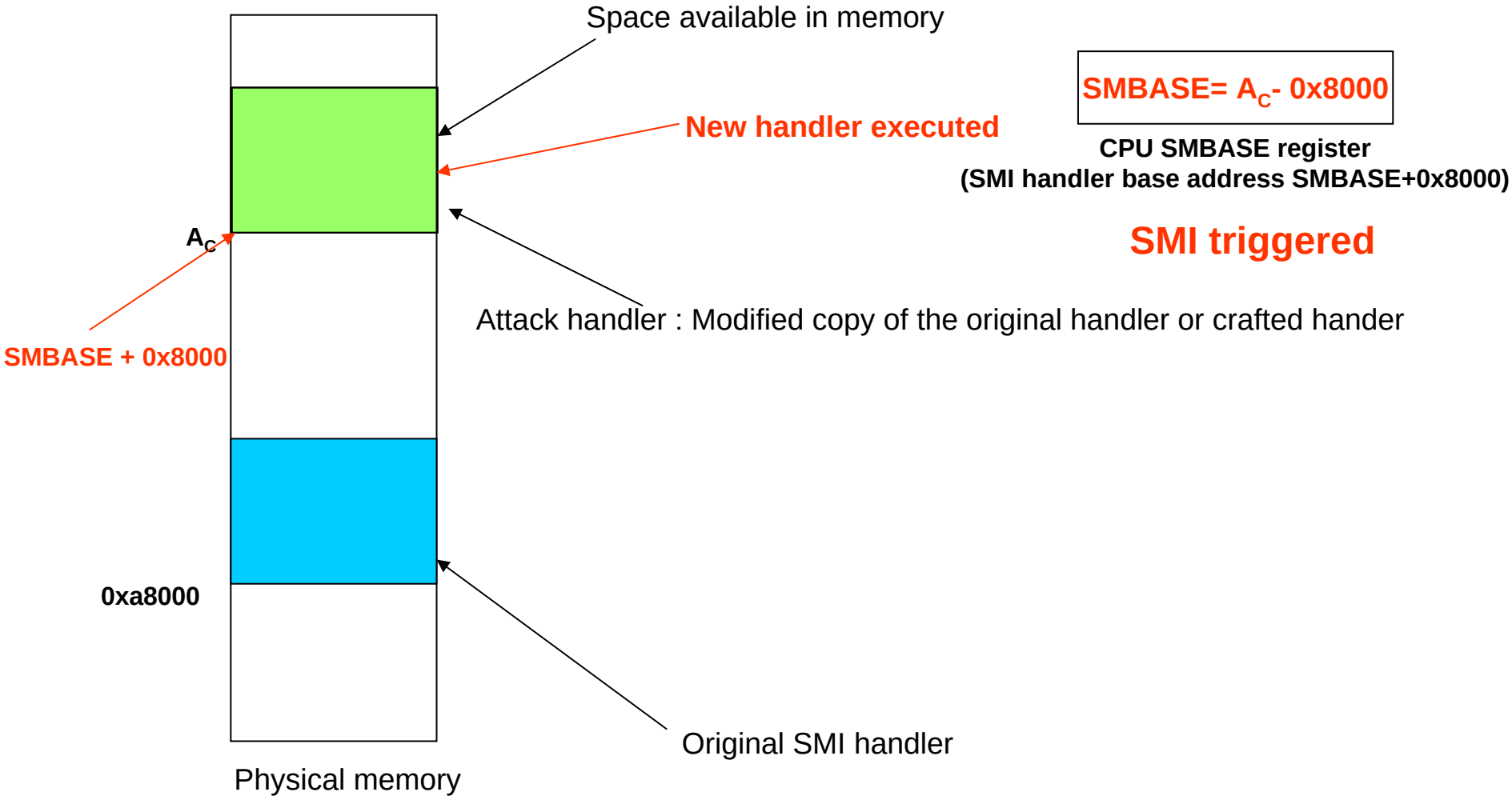
Attack scheme (example for legacy SMRAM)



Attack scheme (example for legacy SMRAM)



Attack scheme (example for legacy SMRAM)



Premier ministre
Secrétariat général de la défense nationale

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

- This can be used by a rootkit to conceal functions in the SMRAM and in the SMI handler.

Why did it work?

- System Management Mode is a CPU mode. The CPU determines which code is bound to be run in SMM.
- D_LCK bit is a chipset security mechanism. It can only protect memory that is accessible from the chipset.
- Only the CPU knows what SMBASE is 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 mechanism seems bound to fail: there have proved to exist at least two different mechanisms to circumvent this particular security mechanism.

Vendor responses

- Intel noticed the problem back in 2006 and chose to modify the way SMRAM caching was handled in recent CPU.
- Intel® added a new register : System Management Range Register (SMRR):
  - SMRR determines the way the SMRAM will be cached.
  - SMRR can only be modified by the SMI handler itself.
  - If the CPU is in SMM, the CPU checks SMRR to determine how SRAM should be cached.
  - If the CPU is not in SMM, the CPU considers that memory is uncachable ; write accesses are ignored and read accesses return a fixed value.
- Requires the BIOSes to take advantage of the feature:
  - should have been done in 2009.
- This mechanism is indeed efficient against SMRAM cache poisoning attacks.

Is all this consistent?

- SMRR makes the whole D_LCK/ D_OPEN mechanism useless from a security perspective:
  - If SMRR is implemented, the CPU can load the first SMI handler and trigger an SMI.
  - Then, the SMI handler can configure the SMRR so that SMRAM is not accessed outside of SMM.
- Achieves the same goal as the D_LCK/D_OPEN mechanism for CPU-related memory accesses:
  - D_LCK / D_OPEN is only useful at the chipset level to prevent DMA accesses to SMRAM.
  - However, Vt-d and I/OMMUs could do the job.
- Why wasn't the SMRR feature added and supported before?

Design flaw 2: not specifying priorities between chipset mechanisms

- Chipsets are basically composed of aggregated functions not necessarily related one to the other.
- Several memory translation mechanisms and access control mechanisms are implemented in modern chipsets.
- Memory control registers can change physical memory mappings and access control are on physical addresses.
- It is not clear (i.e. not specified), which mechanism has priority over which.
- Example:
    - Graphics Aperture (GART) and D_LCK/D_OPEN.
    - On some machines, GART translation occurs before SMRAM access control.

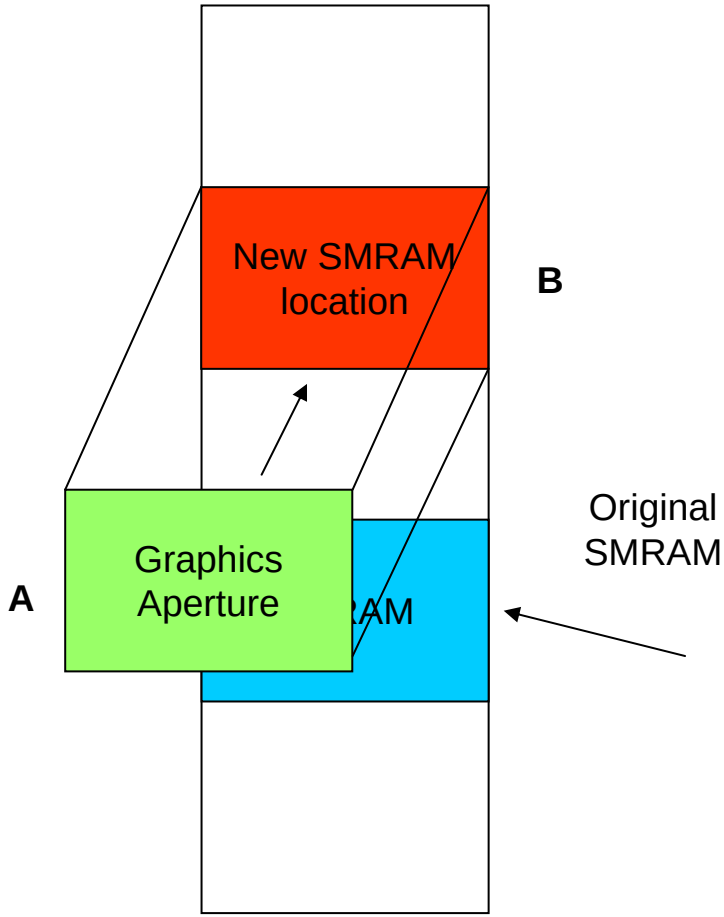
Translation mechanisms attacks (2006-2008)

- 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).

- Possible solutions: BIOS updates that would correctly configure chipset translation mechanisms and prevent attackers from modifying translation tables.

Attack principle overview

- The graphics aperture allows physical memory to physical memory mappings. It is possible to use the graphics aperture to redirect access to memory zone A to memory zone B.
- If graphics aperture base address is identical to SMRAM base address, calls to the SMRAM will first reach the graphics aperture and be redirected to B.
- B is not covered by the chipset access control mechanism and attackers can freely read and write to B.
- The attacker can thus define a new SMI handler under her control.

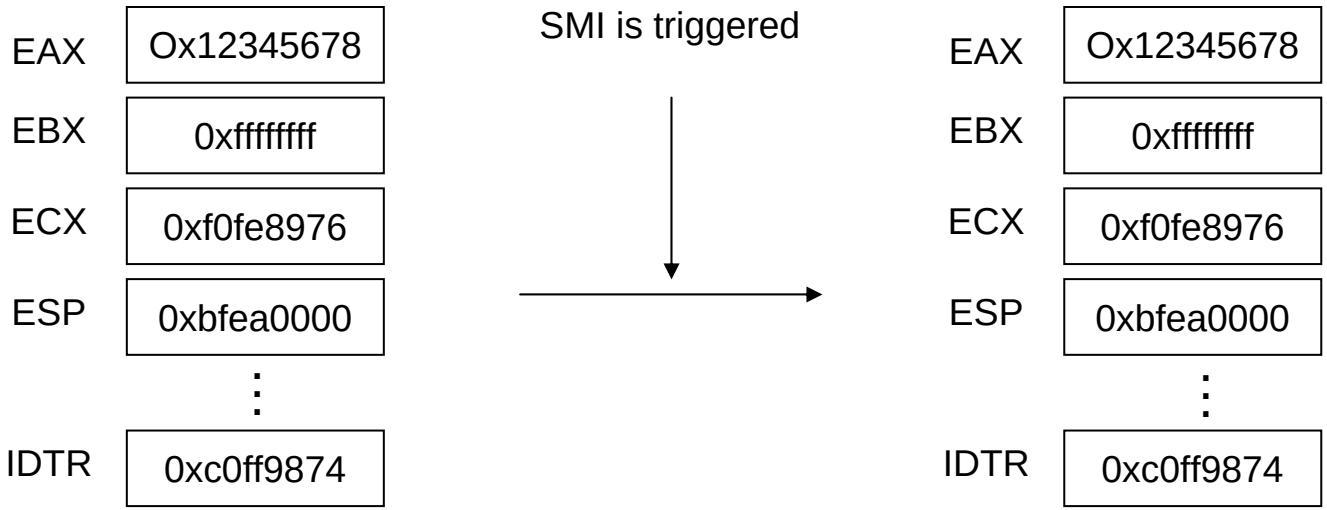


Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
    - Security features
    - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

Design flaw 3
not resetting CPU registers on SMI

- Some registers are set to a well known value when the CPU enters SMM (instruction pointer EIP, debug control register DR7).
- Most registers are not (for performance reasons).
- These registers retain the value they had before the SMI is triggered.

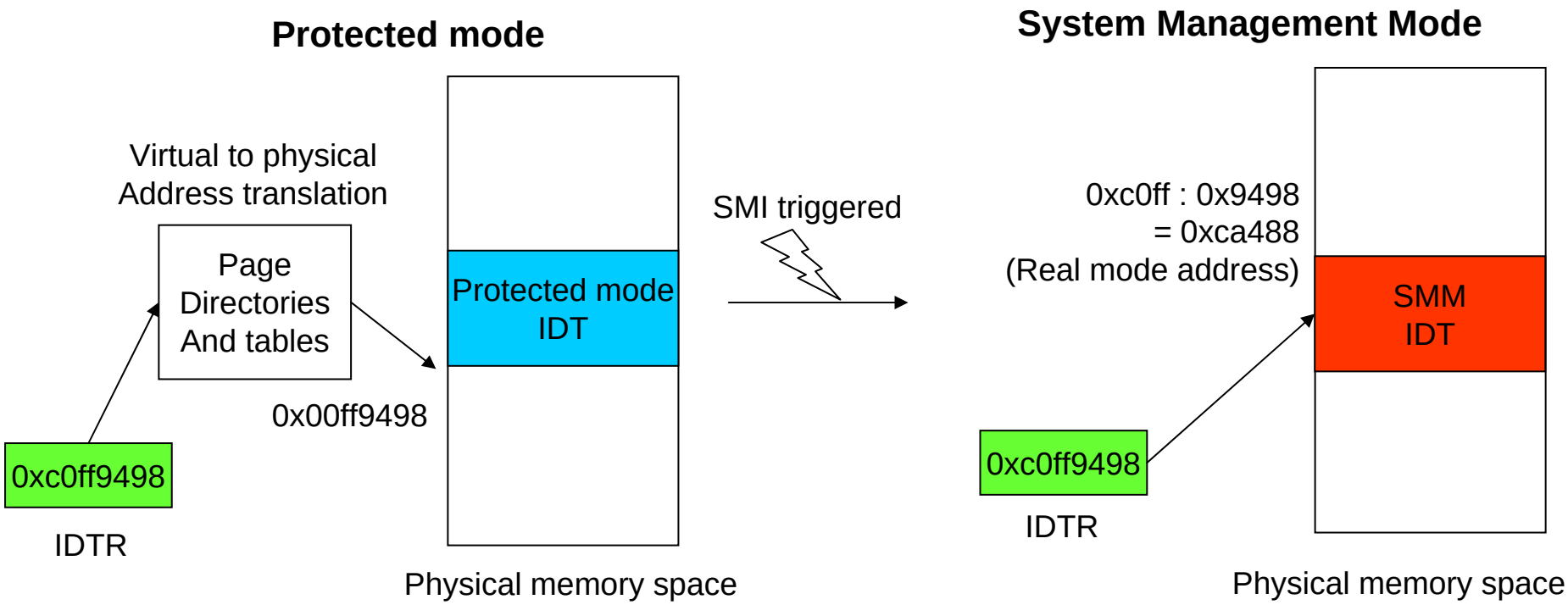


State before SMI (protected mode)

State inside SMM

Example of IDTR

- IDTR stores the interrupt table address
  - virtual address in protected mode
  - real address in 16 bit mode
  - IDTR retains its value when SMM is entered.



IDTR problem (1/2)

- The documentation states :

Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and soft-ware interrupts will result in unpredictable processor behavior.

- In practice, the content of IDTR is used to locate the interrupt table the system is using when an interrupt occurs:
  - Interrupts occurring in SMM can be trapped by handlers under the control of an attacker.
  - The attacker has to forge a 16-bit interrupt table at the correct address.

IDTR problem (2/2)

- Software interrupts are not cleared while in SMM
  - Breakpoints, bounds, exceptions.
  - NMI, A20, SMI interrupts and hardware triggered interrupts are masked (IF flag cleared).
- Arbitrary code can be run in SMM
  - The attacker has to cause the SMI handler to trigger an exception.
  - The handler defined by the attacker will be used.
- Attack feasibility?
  - Some SMI handlers never set an interrupt table.
  - Some SMI handlers probably do, but there is still a window of opportunity before the handler actually sets IDTR.
  - Can we be sure that the SMI handler cannot be forced to trigger an interrupt?

Offensive use of the IDT table to modify the content of the SMRAM

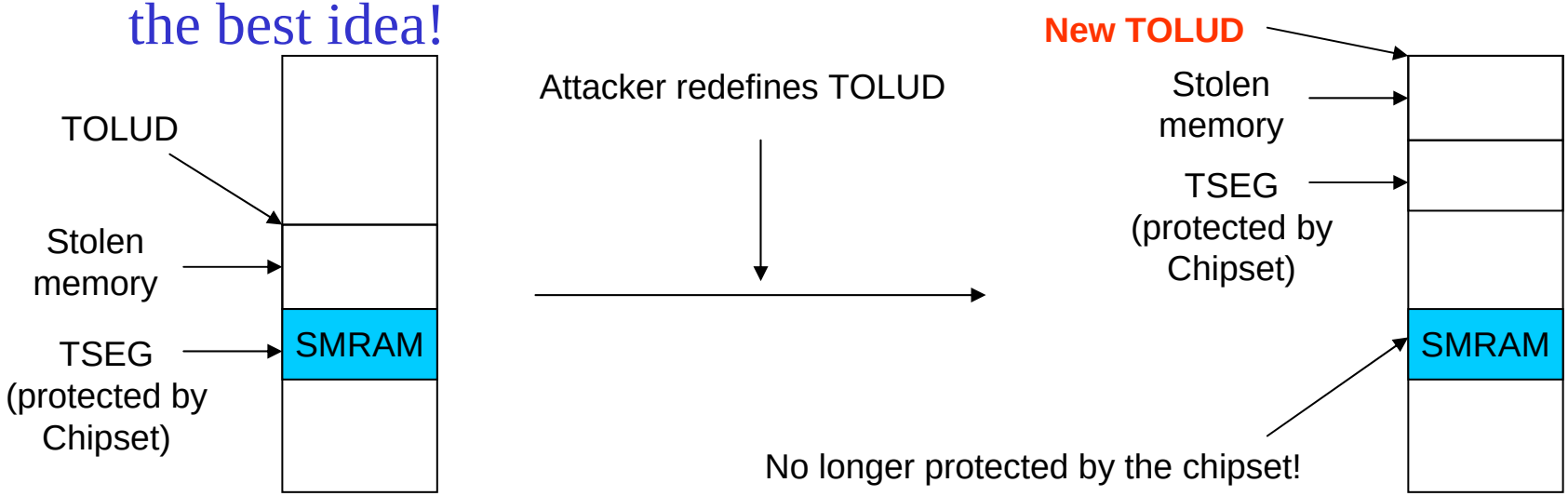
- We assume the attacker can run code in ring 0.
- First the attacker sets IDTR: easy
  - sidt new_idtr_address_and_size
- Then she triggers an SMI : easy
  - outl(0xd, APMC)
- Finally, the attacker manages to cause the SMI handler to trigger a software interrupt or an exception before the SMI handler actually sets the IDTR register to a correct value (if it does): challenging (depends on the SMI handler).
  - Not so many software triggered interrupts in 16 bit mode (but there are a few).

Use of data registers

- EAX, EBX and other data registers also retain their value when SMM is entered.
- This is very convenient for an attacker:
  - If the attacker has already been able to modify the SMI handler to include a backdoor,
  - the attacker can use those data registers to send commands to the backdoor.
  - The backdoor will easily send data back to the attacker (using the data registers in the save state map for instance).
- Sending commands to the backdoor does not require any hardware privilege (setting data registers of the CPU) can easily be done for instance by a javascript application in a browser.

Design flaw 4: TSEG location

- Initially (2006-2007) TSEG was defined as the memory area located just before stolen memory.
- TOLUD (Top of Low Usable DRAM) value, TSEG and Stolen memory size are known to the attacker (as they are defined by chipset registers).
- Not defining TSEG base address in a register was probably not the best idea!



Is the flaw exploitable?

- Assuming the SMRAM is stored in TSEG,
- The attacker could change TOLUD value:
  - outl(new_TOLUD, TOLUD_register)
- This would cause TSEG location to change.
  - TSEG_top = TOLUD- Stolen_memory-size
- The chipset would then protect the new TSEG location, leaving the SMRAM exposed and freely accessible to the attacker.

- Corrected in recent chipsets (2008), TSEG base address is defined in an independent chipset register protected by D_LCK.

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
    - Security features
    - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

SMI handler design Flaw 1: stepping outside of SMRAM

- Only code that is running inside legacy SMRAM, high SMRAM and TSEG is protected.
- If the SMI handler « steps » outside of the SMRAM, then it is potentially vulnerable.

- Stepping outside of the SMRAM could be:
  - Jumping to or calling BIOS or Video BIOS functions;
  - Using an interrupt table outside of SMRAM (traps to interrupt vectors outside of SMRAM).

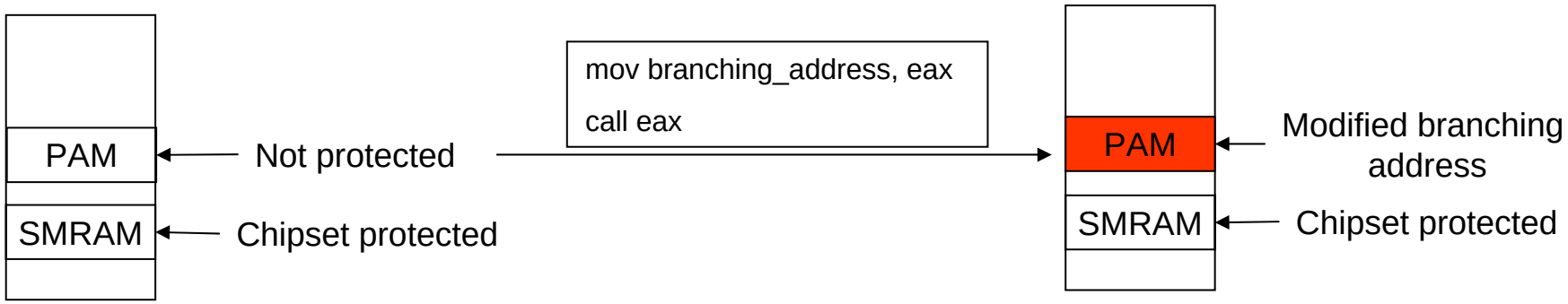
Flaw exploitation

- Either the function called outside of SMRAM is in RAM:
  - The attacker can modify it before an SMI is triggered.

- Or the function is in ROM:
  - The attacker can use the cache poisoning attack to have a modified version of the function (residing in cache) executed.

SMI handler design Flaw 2: fetching a branch address outside of MRAM

- Some SMI handlers fetch branching addresses outside of SMRAM:
  - They do not actually call out of SMRAM.
- Example : PAM (Programmable Attribute Memory location)
  - Used to define platform specific attributes.
  - Can be modified from protected mode.
  - This branching address can be modified by an attacker before the SMI is actually triggered.



Premier ministre
Secrétariat général de la défense nationale

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

Summary of design problems

Problems	Vulnerability window	Solution	Proved exploitable
Cache attacks	Up to 2008	SMRR	Yes
GART, translation mechanisms (Q35)	Up to 2008	BIOS updates GART obsolete	Yes
IDTR	All platforms (potentially)	Set IDTR as soon as possible	Not so far
TSEG location	Up to 2008	TSEG location chipset locked	Yes
Stepping out SMRAM	Depends on platform	Not calling out of SMRAM	Yes
Fetching pointers outside SMRAM	Depends on platform	Not fetching addresses outside SMRAM	Yes

Solutions: recommendations in the current model

- Apply hardware corrections:
  - Use SMRR in the BIOS when possible.
- Protect sufficiently SMI handler code:
  - By initializing the IDTR as soon as possible.
  - By sanitizing all data / pointers coming from outside the SMRAM.
- Add a hardware reset of the registers:
  - In particular, IDTR, to avoid “undetermined behavior”.

Solutions: model change

- Solution 1 (preferred): Power and system management should be managed by the operating system:
  - As the BIOS sends power management information through ACPI tables, it could provide routines to the OS.
  - OS can add its own handlers and check ACPI tables behavior is correct with its policy.
  - The System Management tables could be modified and checked similarly by the OS.
  - The mechanisms provided by the Protected Mode could be used by the OS to rule privileges inside SMM.

Solutions: model change

- Solution 2: run SMM code in the chipset (for instance on the manageability engine).
  - Today's chipsets have enough memory space and computing power to achieve this mission.
  - Requires OEMs to rewrite SMM code.
  - Code running in the chipset will not be accessible to attackers.
  - The downside here is SMM could not be used by manufacturer to provide other services, like a BIOS update manager, through SMM.

Outline

- Introduction
- A (short) description of SMM
- Known SMM design issues:
  - Security features
  - Known exploitable flaws
- Other SMM model design issues or flaws
- SMI handler possible design mistakes
- Reaching SMM security
- Conclusion

Overall conclusion

- In this presentation, we presented several design issues that show that as security was not taking into account at the time SMM was first specified, attempts to add security features at a later time to prevent attackers to modify SMRAM content are bound to fail.
- A correct approach would be to move to a safer model where power management is under the control of the operating system (closer to the ACPI model).

- Not all the design flaws presented today have been proved to be exploitable but we strongly believe that they are (depending on the platform).

Premier ministre
Secrétariat général de la défense nationale

Thank you for your attention
Any questions?

Special thanks to:
Olivier Grumelard (SGDN/ANSSI)
Olivier Levillain (SGDN/ANSSI)
Benjamin Morin (SGDN/ANSSI)

Contact address:
loic.duflot@ssi.gouv.fr

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Premier ministre
Secrétariat général de la défense nationale

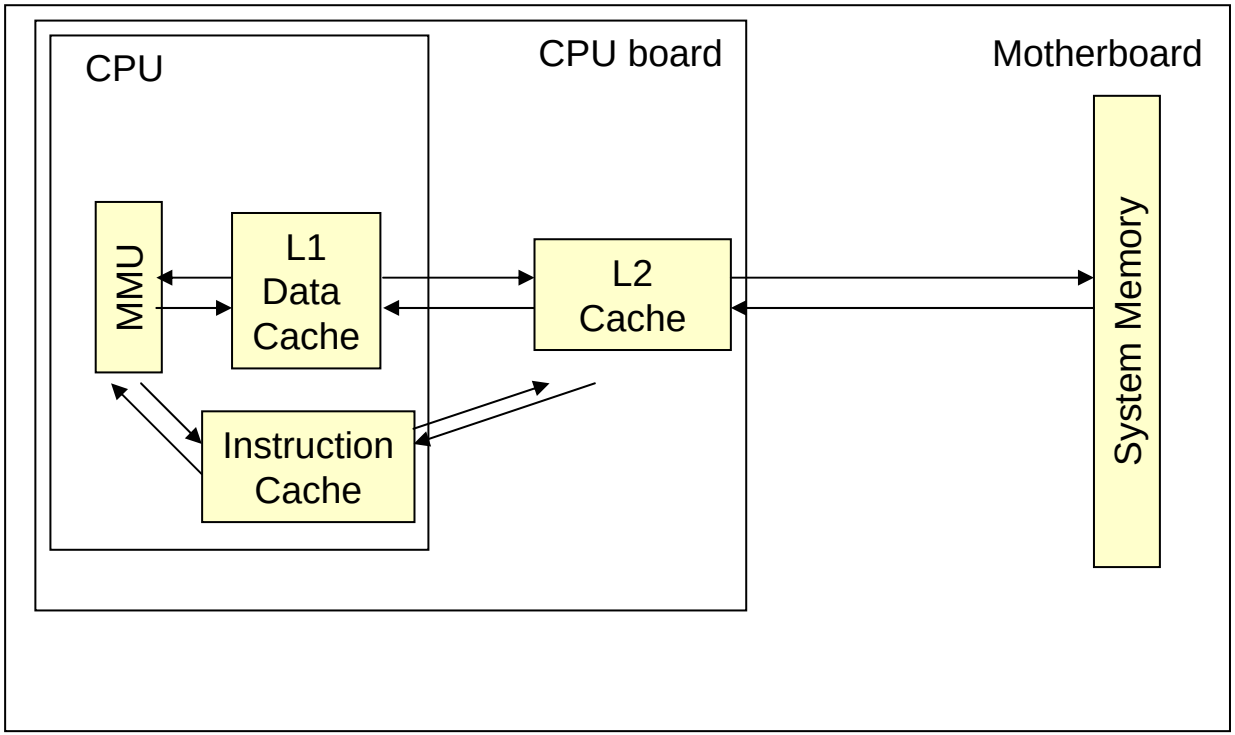
Backup slides

Cache poisoning attacks (2009)

- Code running on the CPU can determine how SMRAM will be cached.
- If the SMRAM is cached in Write Back, the attacker can run code from the cache.
- SMM code is run without the chipset being able to enforce the security policy.
- See:
  - Our Cansecwest 2009 presentation.
  - Joanna Rutkowska and Rafal Wojtczuk (http://theinvisiblethingsblogspot.com/).

Cache hierarchy

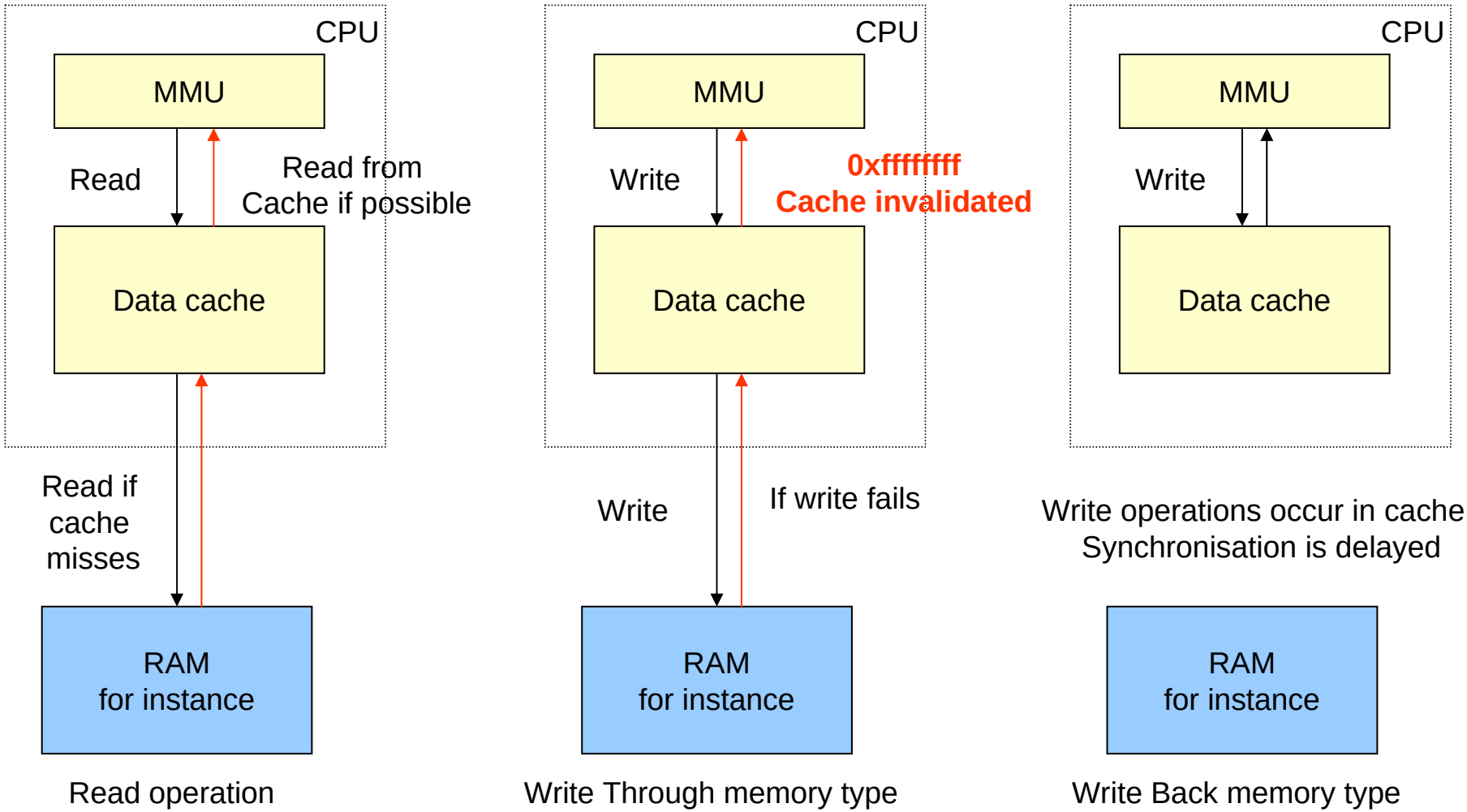
- To speed up memory accesses, caching is used.
- Description of the cache hierarchy of a x86 processor (example):



Memory caching

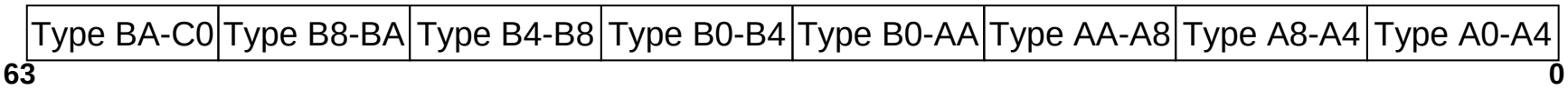
- There is a need to synchronise CPU caches with memory. Different synchronisation schemes can be specified, for instance:
  - WB: write back.
  - WT: write through .
  - UC: not cacheable.
- What memory zones are cached or not in L1 is specified in the memory management unit of the CPU (responsible for the translation between virtual and physical addresses).
- Two different mechanisms to specify the cache strategy: Page directories and tables (the hard way) and MTRR (the easy way).

Cached memory types

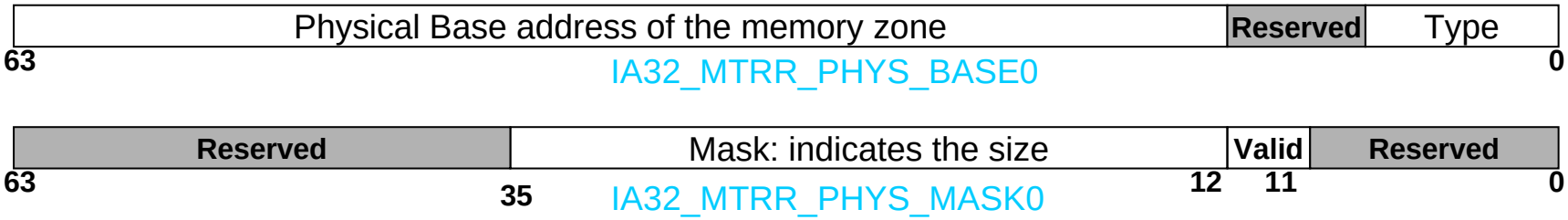


Use of MTRRs

- 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 low legacy memory addresses 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).

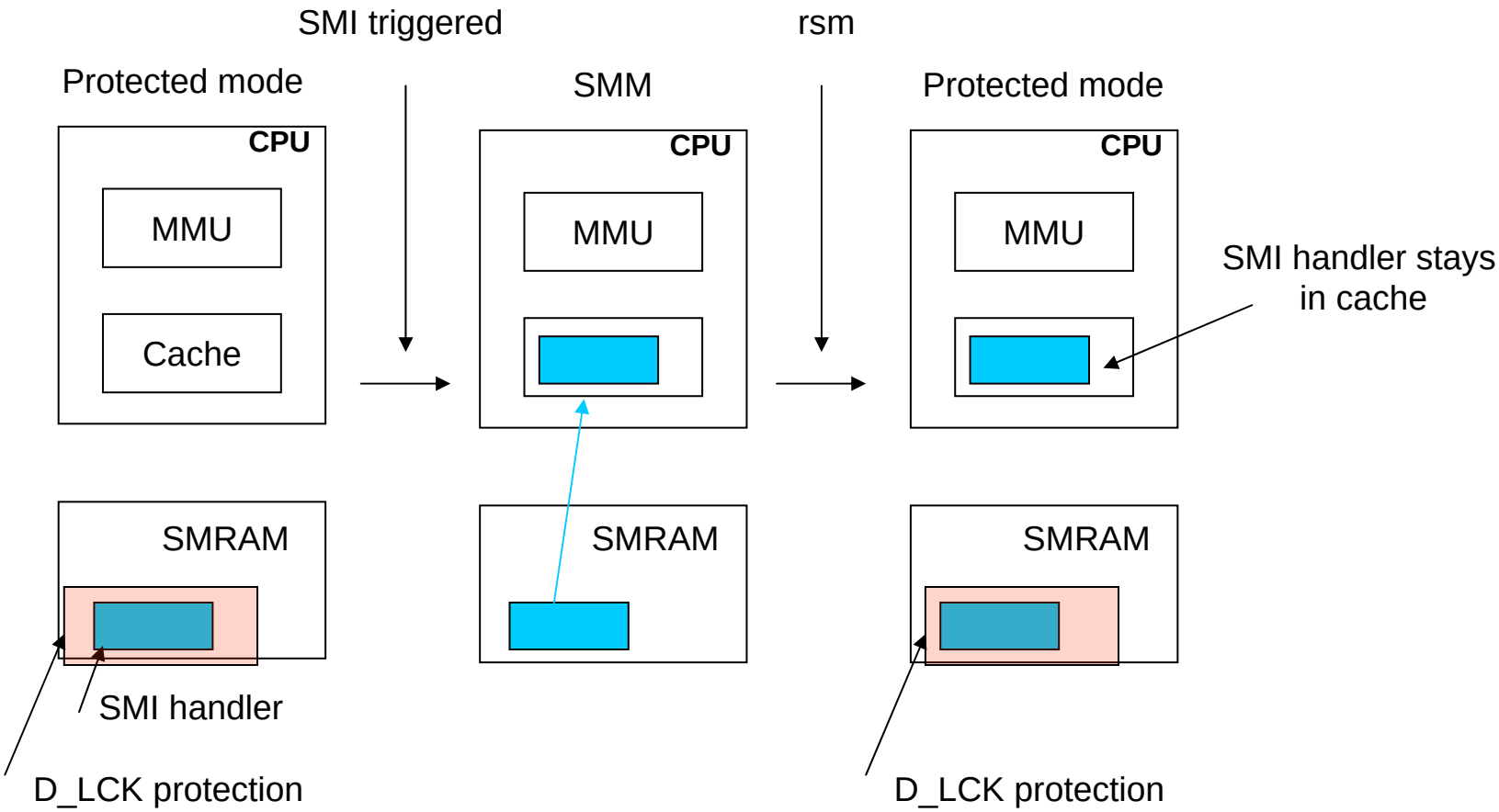
- Isn’t there a coherency problem here?

SMRAM and caching

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

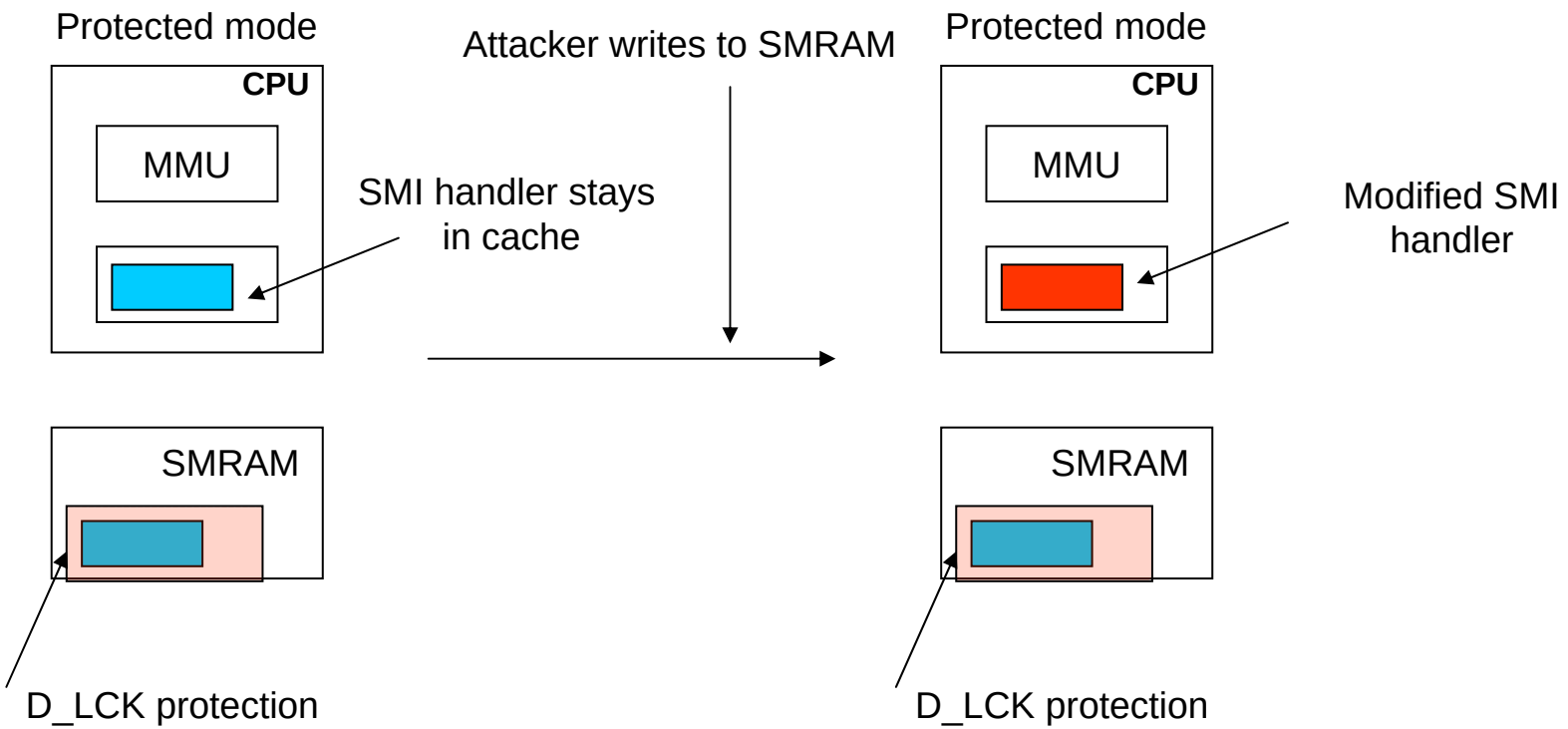
- Let's assume that the SMRAM memory address zone is cached in WB by the CPU. If the SMI handler routine is executed it will be moved to the CPU instruction and data caches.
- If SMM handlers do not flush the cache when they give the hand back to the operating system, it is likely that the SMI handlers 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

- Assume that a rootkit wants to conceal some functions within the SMRAM but the D_LCK bit is set and SMRAM cannot be written.
- 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 if not necessary

- 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);
```

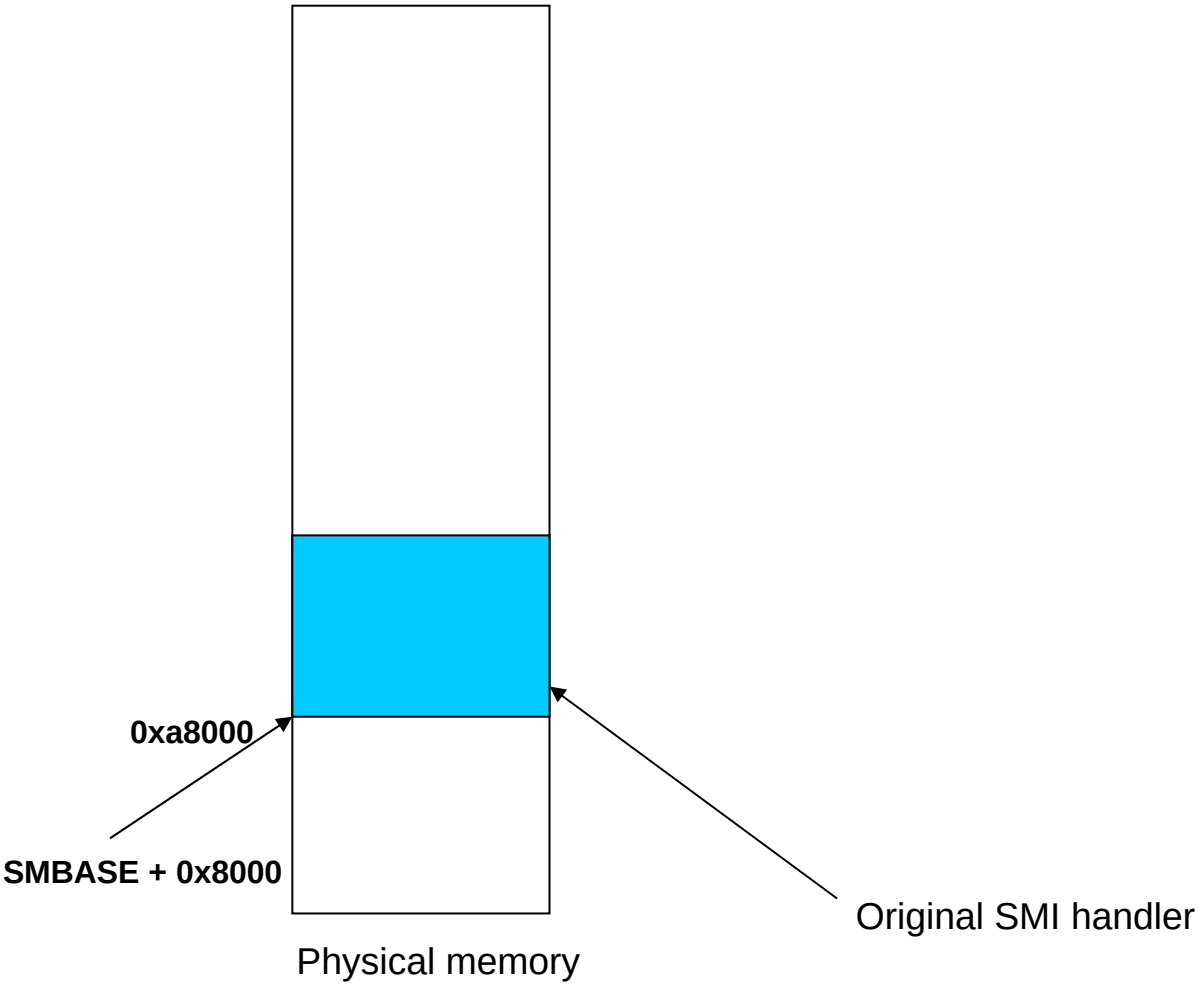
- Objection: But wait! Only the data cache is modified, not the instruction cache, so the modification should have no effect.
  - True, but the instruction caches are flushed during mode transitions as running 16-bit instructions in a 32-bit (or 64-bit) mode should not be advised, so instructions are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

No way!

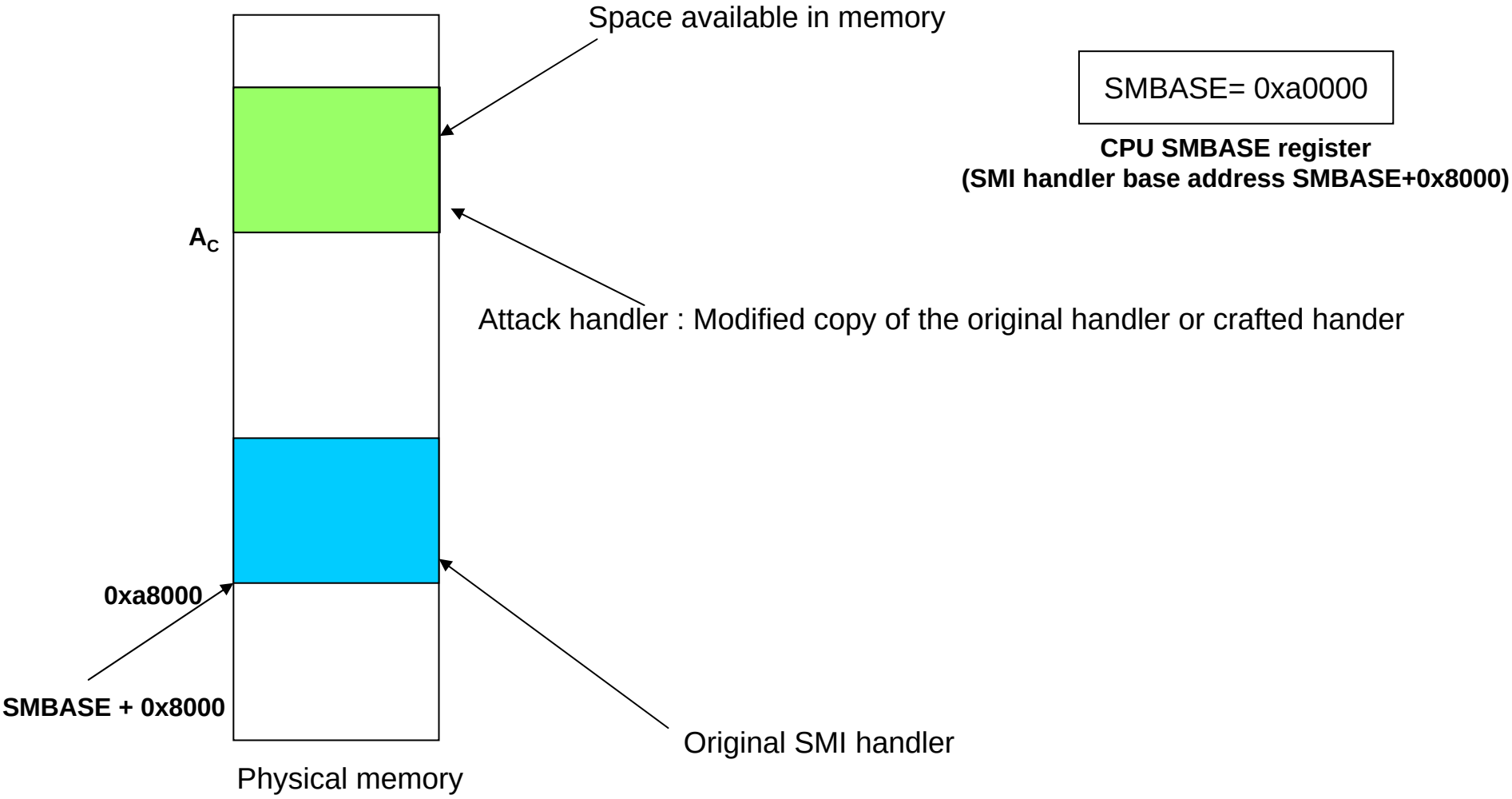
- The SMI handler can flush the cache before exiting!
  - No SMI handler that I have seen does that.
  - Anyway if the handler did so it would not be a major problem.
- Data caches are small, the whole SMRAM won't probably fit in it!
  - True.
- Data are not bound to stay for too long in the data cache 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.

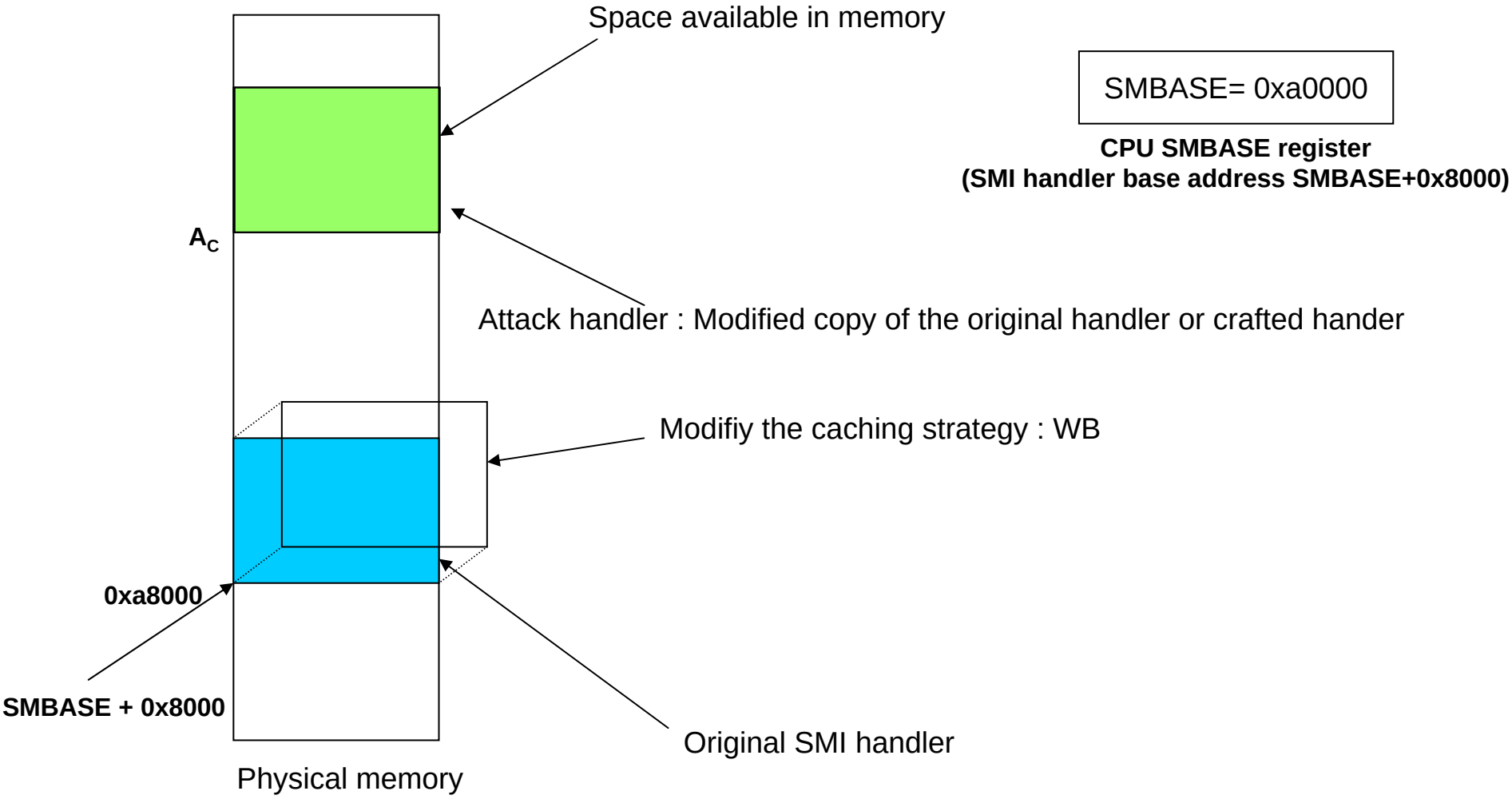
A more efficient scheme (example for legacy SMRAM)



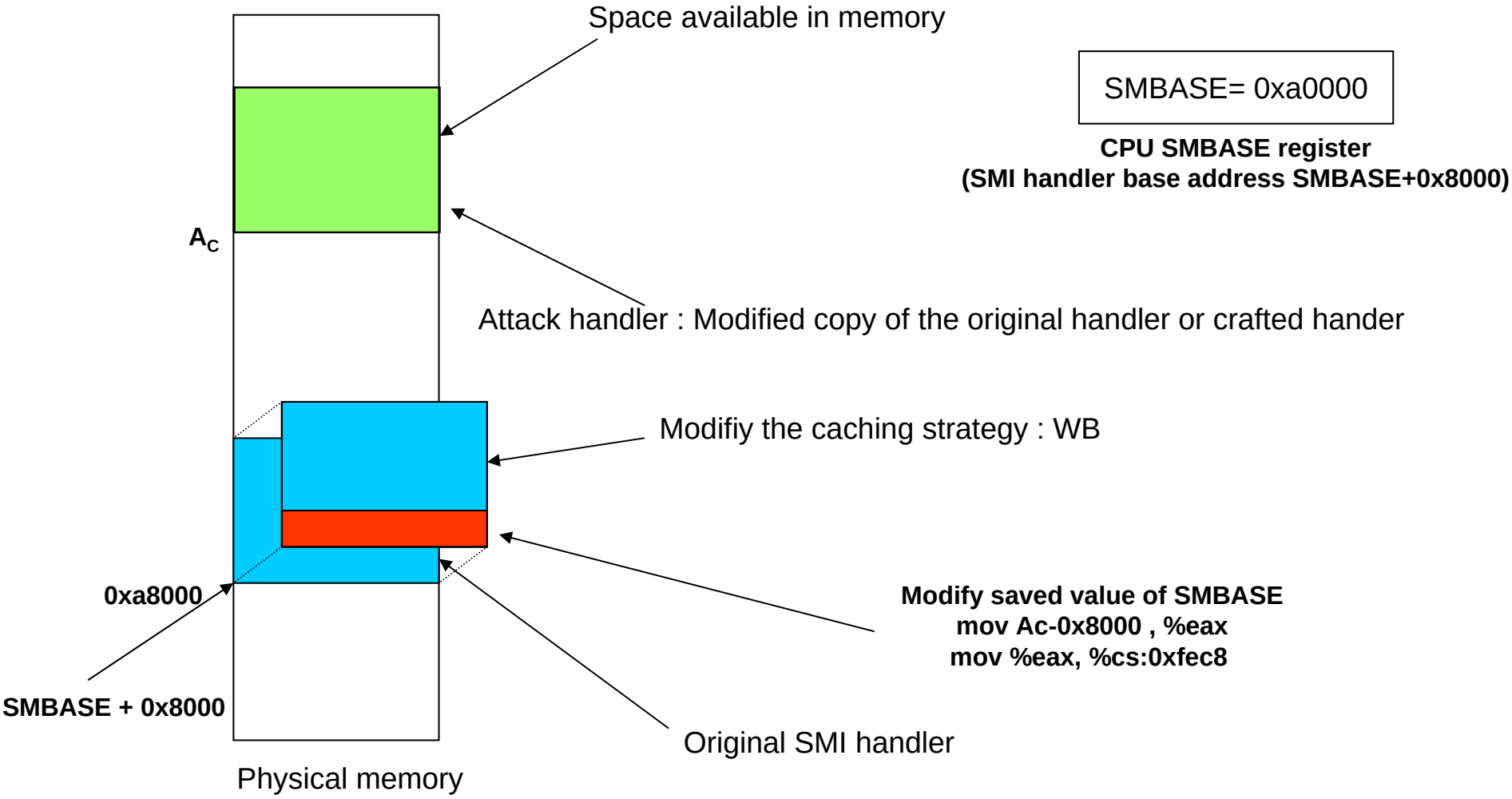
A more efficient scheme (example for legacy SMRAM)



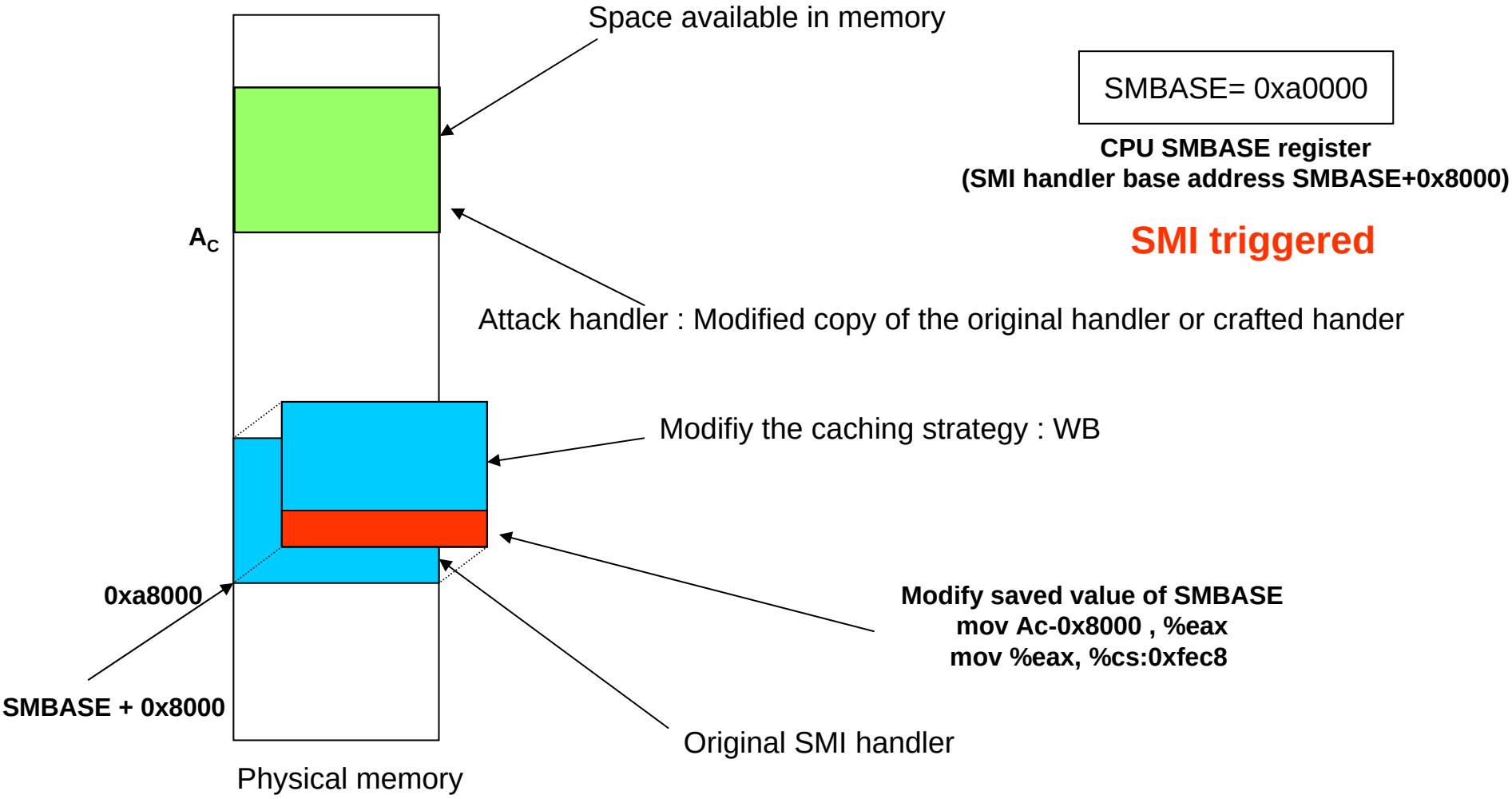
A more efficient scheme (example for legacy SMRAM)



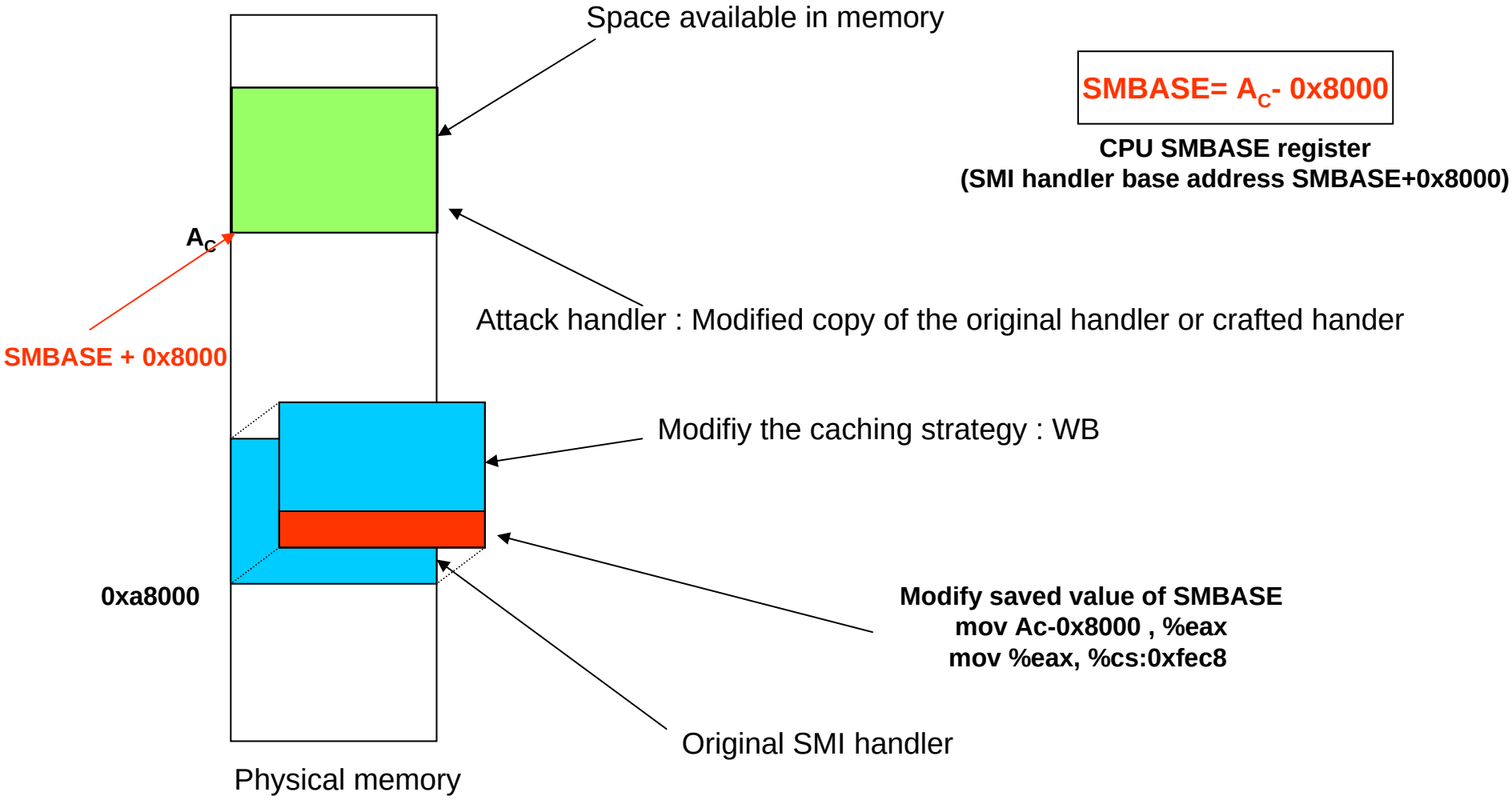
A more efficient scheme (example for legacy SMRAM)



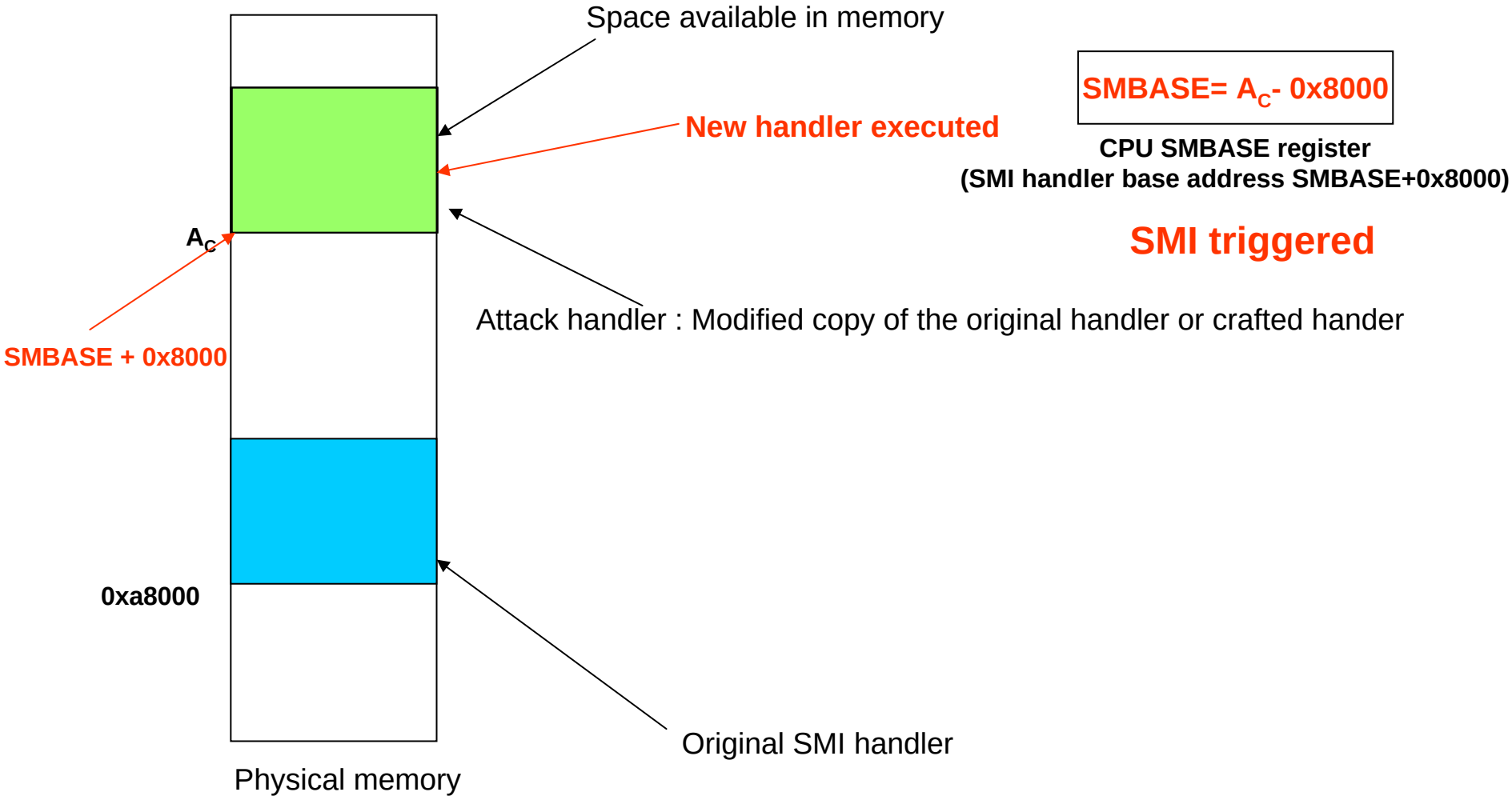
A more efficient scheme (example for legacy SMRAM)



A more efficient scheme (example for legacy SMRAM)



A more efficient scheme (example for legacy SMRAM)

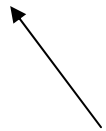


A more efficient scheme

- 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.
  - 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 a SMI. The M is executed. Upon execution of the “rsm” assembly language instruction, the SMBASE register value will 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 SMI handler copy “C” will be executed from that space in 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.

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

- This can be used by a rootkit to conceal functions in the SMRAM and in the SMI handler.

Stealth properties

- If the attack handler C invalidates cache lines where M is stored (using the “clflush” assembly language instruction for instance), relocation occur without any write back cycle to SMRAM.
- 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.

Premier ministre
Secrétariat général de la défense nationale

Why did it work?

- System Management Mode is a CPU mode. The CPU determines which code is bound to be run in SMM.
- D_LCK bit is a chipset security mechanism. It can only protect memory that is accessible from the chipset.
- Only the CPU knows what SMBASE is 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 mechanism seems bound to fail: there have proved to exist at least two different mechanisms to circumvent this particular security mechanism.

Difficulties

- Exploiting the flaw 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 that can prove to be difficult, explanations later).
- On multi-CPU architectures each CPU may use a different SMRAM location and thus a different SMI handler. Which of them will be called when an SMI is triggered is not specified in the specification.
    - 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