Chapter 2
Technical Background, Preliminaries and Assumptions

Putting a computer in front of a child and expecting it to teach him is like putting a book under his pillow, only more expensive.
Joseph Weizenbaum,
German/American Computer Scientist

Although it is beneficial, in order to understand our later material, to know many details about modern computer architecture, it would be unrealistic to explain all these subtle details here. Thus, we refer the reader to literature [see 54, 59, 118, 127] for a thorough treatment of this topic. We limit this chapter to the most important terms that are necessary to understand this work. We start with the *rootkit evolution*. This evolution highlights why the technical background that is presented in the following sections helps to understand this work.

2.1 The Rootkit Evolution

On the popular x86 platform the power of a rootkit strongly correlates to the execution environment, i.e., user-mode (ring 3) or kernel-mode (ring 0), for example. Modern x86 processors provide so-called protection rings to distinguish between different privileged execution environments, see Fig. 2.1. An analysis of the rootkit evolution reveals that attackers discovered new and more powerful execution environments on x86 platforms. The following paragraphs summarize different kinds of rootkits, i.e., user-mode, kernel-mode, virtual machine based, system management mode, firmware-based, and peripheral-based rootkits. This overview represents the rootkit evolution and demonstrates how the term rootkit changed in the recent years.

User-mode rootkits utilize simple techniques. The basic idea is to camouflage the rootkit as normal software [129]. For example, the attacker adds the desired malware functionality to a common software tool that is executed in user mode with super-user/*root* privileges. The modified tool replaces the original tool on the target platform. User-mode rootkits are considered as the starting point in the rootkit evolution. The name is derived from the privilege level that is given by the super-user

P. Stewin, *Detecting Peripheral-based Attacks on the Host Memory*,
T-Labs Series in Telecommunication Services, DOI 10.1007/978-3-319-13515-1_2

Fig. 2.1 “Ring -3” environment compared to other rootkit environments on the x86 platform. Please note, ring 3 and ring 0 are implemented in hardware (host CPU). The terms “ring -1”, “ring -2”, and “ring -3” are used to emphasize the power of the corresponding execution environments. They are not implemented in hardware

root. User-mode rootkits can be discovered by special detection software running in kernel-mode.

Kernel-mode rootkits are based on an advanced technique to hide the rootkit using operating system kernel components [60]. Kernel-mode rootkits modify the kernel, or to be more precise, kernel code (for example system calls) or kernel data. Kernel modifications change the kernel behavior to enforce certain stealth capabilities to hide malicious activities [see 129], e.g., a keystroke code logger. The rootkit executed in kernel-mode is immune to techniques that reveal user-mode rootkits.

Much more powerful rootkits to control a computer system are *Virtual Machine Based Rootkits* (VMBR) such as *SubVirt* [77] and *Blue Pill* [108]. A controlling instance that is called hypervisor or *Virtual Machine Monitor* (VMM) is normally used to host guest operating systems in *Virtual Machines* (VMs). A VMBR exploits the VMM environment to host the operating system of the target computer in a virtual machine. Since the operating system kernel is executed on top of the VMM environment, VMBRs can be considered to be run in “ring -1”. Thus, a malicious controlling instance is placed between hardware and operating system. VMBRs are hard to install. Conversely, VMBRs are also hard to detect. Blue Pill can host the target operating system on-the-fly, i.e., without a shutdown or reboot.

Another powerful execution environment for rootkits is the *System Management Mode* (SMM). SMM is a special high privileged processor mode that executes special system software. It can also be exploited to implement so-called SMM-based rootkits. Code executed in SMM runs with the highest host CPU privileges. This means that a SMM-based rootkit runs with more privileges than the operating system kernel and a hypervisor. Hence, SMM-based rootkits can be considered to be executed in protection “ring -2” [145]. In 2008, Embleton et al. [49] and Wecherowski [144] demonstrated how SMM can be used for rootkits. SMM code is stored in firmware, i.e., SMM rootkits can be considered as a special case of firmware rootkits.

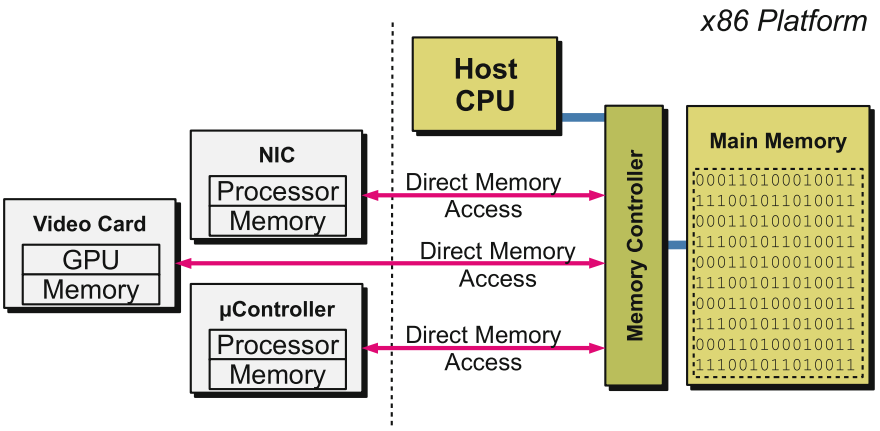

Fig. 2.2 Overview of dedicated isolated hardware potentially exploitable by rootkits. Rootkits hidden in peripherals can directly access the main memory of the computer platform. Hence, they can steal sensitive data, such as the harddisk encryption key, the video telephony session key, online banking credentials, passwords, open files, etc. It is also possible that such rootkits modify data in the main memory

Firmware-based rootkits are also quite powerful. Deploying rootkits in firmware is very difficult, but not impossible. Firmware is special low-level software that is stored on flash memory. The *Basic Input/Output System* (BIOS) is an example of firmware that is stored on flash memory on the x86 platform. A firmware-based rootkit is not deployed on a disk. Thus, it is very difficult to detect and to remove the malicious software. An attacker can use the rootkit to control the computer hardware and to attack the operating system, even if the user reinstalls the operating system. Heasman [56] demonstrated how to implement and detect a BIOS-based rootkit at Black Hat Federal 2006. Heasman [57] continued this research. Further BIOS firmware attacks that can be the basis for a rootkit were presented by Wojtczuk and Tereshkin [149], Loukas [84, 85], and Ortega and Sacco [97, 98]. Brossard [21, 22] also demonstrated that *hardware backdooring is practical*. The author exploits the open source BIOS coreboot[1] and related tools to flash the BIOS and read-only memory of peripherals to attack a computer platform.

Rootkits hidden in firmware can also be implemented using firmware of platform peripherals. Such rootkits are *peripheral-based rootkits*. A potentially exploitable peripheral is the network card [134]. Heasman [55] also discussed how to implement and detect a *Peripheral Component Interconnect* (PCI) based rootkit deployed in expansion *Read Only Memory* (ROM) that is present on the PCI device. Peripherals are well isolated from the actual host system. Hence, the execution environments of peripherals are unconsidered by anti-virus software. This makes peripherals quite attractive for attackers, see Fig. 2.2.

[1] See http://www.coreboot.org/Welcome_to_coreboot [accessed 25 February 2014].

A special micro-controller that executes platform management code on a separate processor offers nice stealth capabilities and can also be used by rootkits. During the Black Hat USA 2009 conference Tereshkin and Wojtczuk [131] presented the idea to use this micro-controller for rootkits. They introduced the term "ring -3" to emphasize the stealth capabilities. Such peripheral-based rootkits are considered to be even more stealthily than SMM-based rootkits. Bulygin [25] demonstrated how to use this special micro-controller based environment to detect SMM-based and VMM-based rootkits. Since peripherals such as network interface cards communicate with the host operating system via the main memory, peripheral-based rootkits can attack the host by illegitimately reading from or writing to the host memory. The mechanism that enables memory access for peripherals is called *Direct Memory Access* (DMA, see Sect. 2.4). Due to this mechanism peripheral-based rootkits are supposed to be absolutely stealthy and undetectable. Such rootkit techniques are the focus of this work. Peripheral-based rootkits can access the host memory to steal passwords, online banking credentials, open documents, etc. that are present in the host's runtime memory via DMA. They can also infiltrate the host with further attack code such as a kernel-based backdoor [47].

Note, in this work we avoid the term "ring -3". No "ring -3" is implemented in hardware. Terms such as "ring -1", "ring -2", and "ring -3" are only used to illustrate the privilege level of the corresponding environment on the x86 platform. The lower the ring the more powerful is the rootkit. In this thesis, we will use the term malware (malicious software) because the attacks that we analyze are not executed on the host CPU. Hence, root privileges are irrelevant. The malware that we focus on has only the goal to operate stealthily in common with original user space rootkits.

2.2 Typical x86-Based System Architecture

The main components of a typical x86 system architecture as depicted in Fig. 2.3. The linkage of *Central Processing Unit* (CPU), *Memory Controller Hub* (MCH), and *Input/output Controller Hub* (ICH) is called the chipset [54]. This chipset solution is also referred to as *3-chip solution*. System memory (*Random Access Memory* or in short RAM) as well as a display adapter are connected to the MCH. The MCH controls access to memory. It can block requests to memory addresses or redirect the request to the ICH, if the destination address belongs to the ICH. Peripheral devices, such as flash memory, *Network Interface Card* (NIC), etc., are integrated into the system using the *Peripheral Component Interconnect express* (PCIe [24]) standard. This standard implements a serial interconnect for peripherals and the chipset. NICs and other add-on cards can be connected to the ICH via PCIe. Flash memory, which stores firmware such as the *Basic Input/Output System* (BIOS [see 54, p. 369]), is also connected to the ICH.

Please note, Intel introduced a so-called *2-chip solution* with the *Intel 5 Series chipset* [121, p. 15]. 2-chip solution means that the MCH functionality moved into the host CPU and is called *Integrated Memory Controller* (IMC [32, p. 14]). The IMC

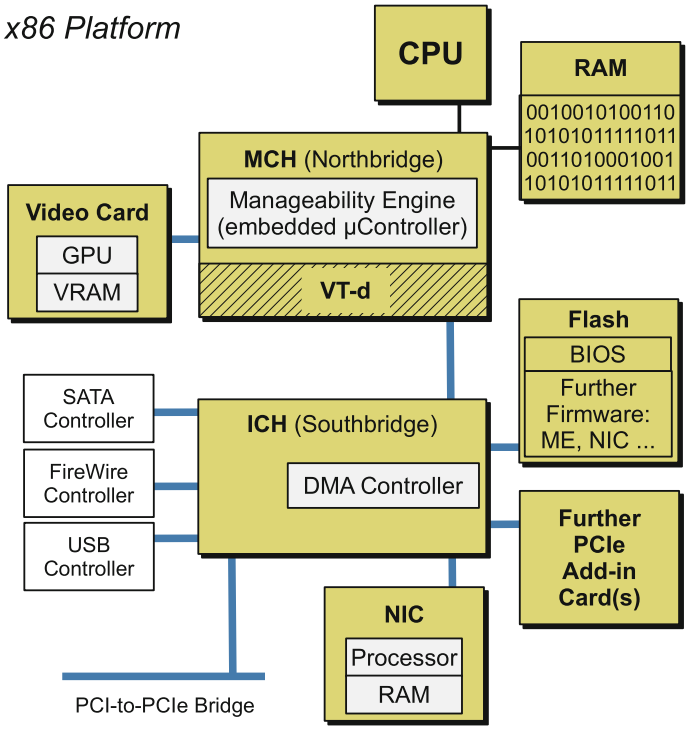

Fig. 2.3 x86 chipset and peripheral components. The chipset components are the *Central Processing Unit* (CPU or host processor), the *Memory Controller Hub* (MCH, also known as northbridge) and the *Input/output Controller Hub* (ICH, also known as southbridge). Peripherals do not belong to the main chipset

is the controlling instance that controls memory accesses just as the former MCH. The ICH was renamed to *Platform Controller Hub* (PCH [68]). The experiments conducted in this thesis are based on the 3-chip solution.

Further controller devices connect other formats, such as *Universal Serial Bus* (USB [8]), *FireWire* (FW [6]), or *Serial Advanced Technology Attachment* (SATA [7]), via PCIe to the system. Legacy PCI devices are connected to the PCIe architecture via a so-called *PCI-to-PCIe bridge* [24]. In laptop computers *Personal Computer Memory Card International Association* (PCMCIA)/*ExpressCard* [139] devices are integrated into the system utilizing PCIe. The host CPU is not necessarily the only processor in the system. The video card, for example, supports a *Graphics Processing Unit* (GPU) to efficiently modify computer graphics. Data to be processed is stored in *Video RAM* (VRAM), that is separated from normal system RAM. Other devices with similar properties are NICs and *Intel's Manageability Engine* (ME [79]) in the platform's MCH. They also utilize separate processors as well as separate RAM to execute firmware.

2.3 Intel x86 Based Host Central Processing Unit

The Intel x86 *Central Processing Unit* (CPU) was announced in 1978 [see 59, Appendix K.3]. Since then, the x86 CPU has been enhanced and nowadays x86 processors consist of several units to support proper features for different computing tasks. Modern extensions are floating point unit, *Single Instruction operating on Multiple Data items* (SIMD [117, p. 524]), *Streaming SIMD Extensions* (SSE [117, p. 748]), x64 [58, p. 351], *Physical Address Extension* (PAE [69, pp. 2–23]), multi-level caches (L1, L2, L3 cache [59, p. 117]), *Performance Monitoring Units* (PMU [see 104, p. 429]) and hardware support for virtualization as described by Grawrock [54]. A modern x86 CPU usually consists of multiple cores [see 59, p. 117]. These cores provide registers of different bit sizes, i.e., from 16 bit up to 512 bit [see 70, Sect. 1.2.1].

To offer protection mechanisms the CPU supports a privilege model via the so-called protection mode. The model provides different privilege levels also known as rings to separate certain software running on top of the hardware. Four rings are available if the processor is in protected mode. Ring 0 is the most privileged ring ring 3 has the fewest privileges. The operating system is executed in ring 0. Thus, it is separated from application software running in ring 3. Ring 1 was considered for device drivers and ring 2 for services, though in practice ring 1 and 2 are not used [54, p. 41].

System Management Mode (SMM [69]) is another processor mode only available for system firmware. That mode was introduced in x86 architectures to implement higher energy-efficiency by, e.g., powering down unused disks and to control system hardware by, e.g., turning on fans and shutting down systems when temperature limits are reached. SMM is triggered by an interrupt, i.e., the *System Management Interrupt* (SMI). SMI handler code is loaded from flash memory by the BIOS into the *System Management RAM* (SMRAM) early in the system initialization. To prevent modifications of the SMI handler code from other processor modes than SMM, the chipset provides a special bit that is called `D_LCK`. The `D_LCK` bit is set to protect the SMI code after loading it into SMRAM. If the `D_LCK` bit is set no alteration of SMRAM content is possible.

When an SMI triggers SMM, the current executed program is interrupted and the processor state will be saved. Afterwards, the processor executes the SMI handler code. When the execution of the handler code has been completed, the saved processor state is restored. After the processor switches back from SMM to the previous processor mode the interrupted program can continue to operate. Note that the previous processor mode has lost CPU cycles/time, since both processor modes cannot be executed simultaneously. SMM can be considered to be a separate execution environment. SMRAM is a separate address space and only accessible when the processor is in SMM. In other words, the OS has no access to SMRAM. Furthermore, privileges in SMM are not restricted, code executed in SMM can call all I/O as well as system instructions.

Hardware virtualization extensions in x86 are called *Intel Virtualization Technology* (Intel VT) on Intel platforms [54]. Virtualization mechanisms are used to run

multiple OSes or applications isolated from each other on a single hardware platform in parallel. A controlling instance called a *hypervisor* or a *Virtual Machine Monitor* (VMM) hosts guest OSes in *Virtual Machines* (VMs). Modern x86 CPUs provide a special instruction set called VT-x. VT-x is part of Intel VT and is intended to support hardware virtualization. This hardware support offers two special CPU operations: VMX root operation and VMX non-root operation. A VMM is run in VMX root operation. VMs running on top of the VMM are executed in VMX non-root operation controlled by the VMM. Both operation modes support their own protection rings, four rings each. Thus, software of the guest system (kernel, drivers, applications, etc.) can be run in the designated privilege level. The protection rings in VMX non-root operation are considered to be unprivileged, since these rings are controlled by the VMM running in VMX root operation. The four rings of the VMX root operation mode are privileged. Usually, the VMM uses only the most privileged ring. This ring is often called "ring -1" to emphasize that it controls the unprivileged rings 0–3.

The x86 micro-architecture also implements a pipelining concept with special execution optimization features, such as branch prediction and out-of-order execution [118, p. 329ff] [127, p. 93ff]. The execution pipeline works with micro-operations, i.e., computations that are implemented as stylized atomic units. Intel architecture instructions are translated into micro-operations [118, p. 331]. For out-of-order execution a so-called *Reorder Buffer* (ROB [118, p. 333]) is required to keep track of renamed registers. Register renaming occurs during out-of-order execution. Registers used in micro-operations are renamed by utilizing the *Register Alias Table* (RAT [118, p. 333]) that is also referred to as the *Register Allocation Table* (RAT [see 127, p. 100]).

PMUs are implemented in the form of *Model-Specific Registers* (MSR [69, Sect. 9.4]) that enable software developers to count micro-architecture related events. This helps programmers to write optimal code for a certain CPU micro-architecture [104]. For example, the MSRs can be configured to count cache misses, RAT stalls, and branch mispredictions that occur when executing code [69, Chaps. 18/19]. The PMU registers that count events are also referred to *Performance Counter* or *Hardware Performance Counter* (HPC). They are only available in ring 0. Another special purpose register that is related to performance measurements is the so-called *Time Stamp Counter* (TSC [69, Sect. 17.12]) register. The TSC register can be used to count CPU cycles after a platform reset. Access to the time stamp counter register as well as to the performance monitoring unit registers from different privilege levels can be controlled via the x86 control register 4 (`CR4`) [see 69, Chap. 2].

A special input/output (I/O) feature to exchange data with peripherals is the concept of I/O-mapped I/O via ports (I/O ports [117, p. 70, 341]) that is provided by the x86 CPU. This concept is complementary to memory mapped I/O (also supported by x86 systems [117, p. 343]) where memory as well as registers of peripherals are mapped into the memory address space of the host CPU. Peripherals also communicate with the host CPU via interrupts to signal that new data is available, for example [117, p. 252]. To communicate with the host system, peripherals can also use the concept of direct memory access. In this case the peripheral does not communicate directly with the host CPU, see Sect. 2.4.

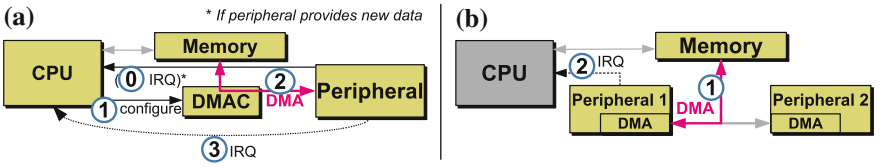

Fig. 2.4 Third-party and first-party DMA. **a** Third-party DMA: The host CPU is required to (*1*) configure (source and destination address) the central DMA controller via I/O ports to (*2*) perform a DMA transfer. The host CPU is (*3*) interrupted when the DMA transfer has been finished [31, p. 454]. Hence, the host CPU is aware of a third-party DMA transfer.—**b** First-party DMA: The peripheral device can (*1*) configure its own DMA engine. The device acts as bus master (see Sect. 2.5) to get control of the system bus to perform a DMA transfer. The device *can* interrupt the host CPU when the device (*2*) has completed the transfer. The transfer also works if the device does not interrupt the host CPU at the end of the DMA transfer. In this case the CPU is unaware of the DMA transfer

2.4 Direct Memory Access

PCIe supports *Direct Memory Access* (DMA) for peripherals, or to be more precise for dedicated hardware such as video cards, NICs, and management controller. DMA enables fast memory access without the involvement of the host CPU. The aim of DMA is to remove the burden from the host CPU. DMA allows peripherals to gain access to the whole host memory bypassing the CPU. The CPU can perform other tasks while DMA transfers occur. Peripherals can have their own engines to perform DMA. This kind of DMA is called first-party DMA [133, p. 428]. Another mechanism is third-party DMA [133, p. 428] where a central *DMA Controller* (DMAC, see Fig. 2.3) is necessary to provide legacy devices (e.g., devices based on the *Industry Standard Architecture* (ISA [116]) format) without DMA engines with fast memory access. It is also integrated in modern platforms [64, p. 128].

Figure 2.4 highlights an important difference regarding stealthy operation between third-party and first-party DMA. When using third-party DMA the host CPU is aware of the DMA transfer, because the peripheral needs the host CPU to configure [see 31, p. 454] the DMAC via I/O ports[2] (see Sect. 2.3). When using first-party DMA the host CPU is not *necessarily* aware of the transfer. Note, a DMAC or a DMA engine can only access host memory addresses, but not host CPU cache, host CPU registers, or the harddisk, for example. The latter implies that data swapped out from runtime memory to the harddisk is not accessible by a DMA engine, either.

[2] See the Linux source code files `arch/x86/include/asm/dma.h` and `arch/x86/include/asm/io.h`, for example.

2.5 Bus Master

A computer platform has several bus systems, such as PCIe and *Front-Side Bus* (FSB). Hence, a platform has different kinds of bus masters depending of the bus systems, see Fig. 2.5. A bus master is a device that is able to initiate data transfers (e.g., from an I/O device to the main memory) via a bus [58, Sect. 7.3]. A device (CPU, I/O controller, etc.) that is connected to a bus is not per se a bus master. The device is merely a *bus agent* [1, p. 13]. If the bus must be arbitrated a bus master can send a bus ownership request to the arbiter [9, Chap. 5]. When the arbiter grants bus ownership to the bus master, this master can initiate bus transactions as long as the bus ownership is granted. Note, this procedure is not relevant for PCIe devices due to its point-to-point property. PCIe requests are not required to be arbitrated and therefore, bus ownership is not required. The bus is not shared as it was formerly the case with the PCIe predecessor PCI.

Nonetheless, the bus master capability of PCIe devices is controlled by a certain bit, that is called *Bus Master Enable* (BME). The BME bit is part of a standard configuration register of the peripheral and is usually set by the corresponding device driver that is executed on the host CPU. The MCH (out of scope of PCIe) still arbitrates requests from several bus interfaces to the main memory [63, p. 27], see Fig. 2.5. The host CPU is also a bus master. It uses the *Front-Side Bus* (FSB) to

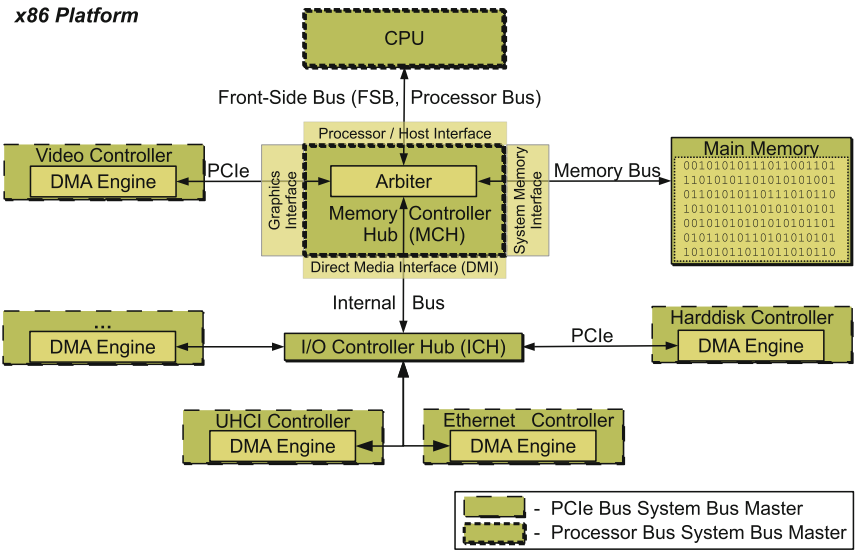

Fig. 2.5 Bus master topology. Bus masters access the memory via different bus systems (e.g., PCIe, FSB). The MCH arbitrates main memory access requests of different bus masters. (based on [23, p. 504][24][58, Section 7.3][63, Section 1.3][64])

fetch data and instructions from the main memory. I/O controller (e.g., ethernet, harddisk controller, etc.) provide separate DMA engines for I/O devices (e.g., USB keyboard/mouse, harddisk, NIC, etc.). This means that when the main memory access request of a peripheral is handled by the MCH, PCIe is not involved at all.

2.6 Input/Output Memory Management Units

Intel introduced a technology called *Intel Virtualization Technology for Directed I/O* (VT-d, [2]) as one of several building blocks to provide hardware supported virtualization for x86 systems. VT-d can be considered as an *Input/Output Memory Management Unit* (I/OMMU) to efficiently assist virtualization requirements, such as reliable isolation of virtual machines running on a virtual machine monitor. VT-d is mainly used in conjunction with virtualization solutions. With VT-d, system software, that means a hypervisor or an OS, can create memory protection domains. For example, isolated subsets of physical memory can be assigned to a virtual machine or to memory of an I/O device driver. An I/O device that is not assigned to a protection domain has no access to physical memory of that domain. These access restrictions are realized using address translation tables. System software configures so-called *DMA Remapping* (DMAR) engines provided by Intel VT-d. Such an engine maps a memory request, for example triggered by an I/O device, to physical memory. VT-d can block a memory request, if the device is not assigned to the protection domain. Please note, an activated I/OMMU can introduce significant performance overhead for the host CPU [13] [88, p. 29] [150]) with the result that the utilization of this technology is often avoided.

To enable system software to configure DMAR engines, the BIOS is required to load corresponding information in the form of *Advanced Configuration Power Interface* (ACPI [44]) tables into the main memory. System software can use this information (e.g., number of DMAR engines) to set up protection domains. Please note, storing the ACPI tables in the main memory raises a serious security threat. These tables are accessible via direct memory access and can be modified as described by Wojtczuk et al. [148] and Sang et al. [111]. System software that is responsible to configure the DMAR engines correctly might fail if this vulnerability is exploited by an attacker.

2.7 Trust and Adversary/Attacker Model

The attacker model provides a description for a stealthy DMA attack scenario. The attacker is able to infiltrate dedicated hardware present in a computer platform with malicious payload *remotely*. This can be carried out via an OS or firmware related zero-day exploit [see 47, for example]. We assume the attacker is able to attack the target platform during runtime. This can not only be done remotely using a

firmware exploit, but also via a remote firmware update mechanism as demonstrated by Duflot [45] and by Triulzi [135], respectively. Alternatively to the described remote exploitation, the attacker can also infiltrate the peripheral before the supposed owner gains and deploys the peripheral on the target platform.

The dedicated hardware supports first-party DMA as described in Sect. 2.4 and accesses the main memory via the memory bus, see Fig. 2.5. We assume that the target computer platform has usual up to date defense mechanisms such as anti-virus software and a host firewall. The platform user does not apply additional hardware such as a hardware firewall to protect the computer platform. We assume that only a stealthy attack can result in a successful attack. Hence, the attacker wants to hide the attack by using the stealth potential of dedicated hardware. Attacks on the main memory (i.e., confidentiality and integrity violations) only originate from peripherals via DMA. The attacker does not implement an attack that requires a cooperation between peripheral and host to increase the probability of a stealthy attack. We further assume that the attacker ensures that an integrity violation (memory write access) does not result in an attack revelation. Additional hardware would decrease the probability of a successful stealthy attack significantly. Most likely, the attacker aims on stealing data, e.g., to conduct industrial espionage or to acquire online banking credentials, etc. To do so, the attacker wants to read data from (confidentiality violation) or write data to (integrity violation) the main memory via DMA.

We consider a computer platform as trustworthy if it conforms to the applied security policy, that means in our case no DMA-based malware is attacking the host platform by reading from or writing to the platform's main memory via DMA. We rely on a minimal *Trusted Computing Base* (TCB [37, p. 66] [99, p. 8]) that consists of the host CPU and the RAM chip hardware as well as the communication path in between (front-side bus, memory controller hub, memory bus). Software (system software as well as application software) executed on the host CPU, is in a trusted state before the platform is under attack. This means that software is loaded as well as started correctly and behaves as expected. We do not count on preventive approaches such as I/OMMUs due to the security issues mentioned in Sect. 2.6.

http://www.springer.com/978-3-319-13514-4

Detecting Peripheral-based Attacks on the Host Memory

Stewin, P.

2015, XV, 108 p. 35 illus., 34 illus. in color., Hardcover
ISBN: 978-3-319-13514-4