Chapter 4
Study of a Stealthy, Direct Memory Access Based Malicious Software

In God We Trust; All Others We Monitor.
Motto of the Air Force Technical Application Center,
Part of the Air Force Intelligence,
Surveillance and Reconnaissance Agency

The arms race between malware developers and the anti-malware community reached a new level. Countermeasures for kernel level [60], hypervisor-based [77], and system management mode based malware [49] were proposed [25, 51, 107]. As a result researchers explored new environments for stealthy malicious software. Malware can be placed on dedicated hardware such as video cards and network interface cards to attack the host platform [see 47, 134, 135]. Such devices bring, among other things, a dedicated processor and dedicated runtime memory. These devices can operate independently from the host system. Anti-virus software cannot detect malicious code stored in separate memory and executed on a different processor. An attacker can use such devices, or more precisely, the direct memory access mechanism to circumvent protection mechanisms built into the operating system by attacking the host runtime memory directly. We call code performing targeted DMA-based stealthy attacks to locate and read or modify target data *DMA malware*. Such data can be cryptographic keys for encrypted harddisks, credentials for online banking accounts, instant messenger chat sessions, and open documents located in the file cache.

In this chapter we characterize DMA attacks and derive the term DMA malware. We explore the term by examining if DMA malware can significantly increase the probability of performing a successful stealthy attack against a computer platform while preserving efficiency and effectiveness. For the evaluation we built our DMA malware DAGGER—a DmA-based keystroke loGGER that exfiltrates captured data to an external entity. We are interested in the efficiency, effectiveness and especially in the stealth properties of DMA malware. We chose to implement a keystroke logger to demonstrate that "short living" data can be captured by DMA malware.

Our implementation is based on Intel's manageability engine that is part of the popular x86 platform. Intel's ME is implemented in business as well as consumer platforms (see Intel vPro platforms [66]) to support different applications, such as the *Intel Active Management Technology* (iAMT [39]) or the *Identity Protection Technology* (IPT [67]). Our DMA malware DAGGER is not executed on the host

P. Stewin, *Detecting Peripheral-based Attacks on the Host Memory*,
T-Labs Series in Telecommunication Services, DOI 10.1007/978-3-319-13515-1_4

processor. It is executed on the processor provided by Intel's ME. No additional hardware is required. DAGGER implements an isolated runtime attack on user input. Additionally, our DMA malware could steal cryptographic keys, target OS kernel structures in an attack, and copy files from the file cache. Although DMA malware cannot by detected by anti-virus software, an attacker still faces certain challenges. DMA malware must be effective, i.e., it should be able to successfully attack various systems. DMA malware must also be efficient, i.e., fast enough to find and process data, even when dealing with virtual memory addresses and randomly placed data. Such malware goes beyond the capability to exploit DMA hardware.

The main contributions of this chapter are:

- **DMA malware definition**: There are different kinds of code that utilizes DMA. To clearly identify if code should be considered harmless, an attack, or DMA malware, we introduce an appropriate definition.
- **DMA malware core functionality**: We present a number of requirements that must be fulfilled by DMA malware in order to mount successful attacks.
- **Evaluation of DMA malware prototype implementations**: To demonstrate that DMA malware increases the probability for successful stealthy attacks while preserving efficiency and effectiveness, we implemented DAGGER. DAGGER is executed on Intel's isolated ME. DAGGER operates stealthily and can attack multiple operating systems. Our implementation is fast and efficient that it can capture keystrokes very early in the platform boot process, that enables DAGGER to capture harddisk encryption passwords under Linux, for example.
- **DMA side effect detection approach**: We present a detection approach that can reveal DMA malware executed in isolated hardware environments. Our work demonstrates that DMA malware produces unexpected side effects that we measure utilizing widely used and cross platform available CPU features.

4.1 DMA Malware Definition

To define the term DMA malware we first characterize different kinds of DMA-based code. This helps to clearly distinguish between simple DMA usage, DMA attacks and DMA malware, whereby the latter has a clear focus on stealthiness. Note, DMA malware goes beyond the capability of controlling a DMA engine. DMA-based code that implements malicious functionality is considered a serious threat. Such code can be operating stealthily during infiltration and runtime. It is also an advantage, e.g., for long-term attacks, if the code can survive platform reboots and power off as well as standby modes. Hence, we can prioritize the following criteria to assess code that utilizes DMA. That is, the DMA-based code:

(C1) implements malware functionality
(C2) needs no physical access to increase the probability of stealthy infiltration
(C3) applies rootkit/stealth capabilities during runtime
(C4) can survive reboot/standby/power off modes

Table 4.1 Fulfillment of criteria C1–C4 of DMA attack examples

Attack presented in	C1	C2	C3	C4	DMA malware
[90] (USB)	–	–	–	✓	–
[15, 17–19, 42, 43, 87, 101] (FireWire)	✓	–	✓	✓	–
[11, 61, 87] (PC card)	✓	–	✓	✓	–
[131] (Intel ME)	–	✓	–	✓	–
[35, 36, 47] (NIC)	✓	✓	✓	✓	✓
[134, 135] (Video card and NIC)	✓	✓	✓	✓	✓
[80] (Video card)	✓	✓	–	–	–

Note, the assessment was done using publicly available material. If we could not decide with the help of available resources whether a criterion is fulfilled, we assume that this criterion is fulfilled.

We use a binary system for our prioritization:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \mathsf{C1} & \mathsf{C2} & \mathsf{C3} & \mathsf{C4} \end{array}$$

This system distinguishes 16 kinds of DMA-based code. We can derive a unique number for each kind. For example, DMA-based code that does not perform malicious actions ($\mathsf{C1} = 0$), leaves no traces on the host ($\mathsf{C3} = 1$), does not need physical access ($\mathsf{C2} = 1$), and cannot survive reboots ($\mathsf{C4} = 0$) is mapped to the binary pattern 0110. This pattern corresponds to class 6 in decimal. The higher the derived number, the more dangerous is the DMA-based code.

Our definition of DMA malware is as follows:

> **Definition**: DMA malware is malicious software executed on dedicated hardware attacking a computer system via a mechanism called direct memory access as well as fulfilling at least the criteria C1, C2, and C3.

When applied to the target platform introduced in Chap. 2, this definition means, that DMA malware is based on first-party DMA and the DMA engine can be configured by the attack code to not involve the host CPU. The attack code is executed on dedicated hardware with its own processor and runtime memory, such as a NIC. Controlling the NIC increases the probability that an attacker can hide data during exfiltration. Table 4.1 applies our binary system to the DMA attacks that are presented in Chap. 3 "Related Work". The table also depicts what related work is DMA malware according to our definition. In this chapter we also aim to develop a DMA malware proof of concept that fulfills at least the criteria C1, C2, and C3.

4.2 DMA Malware Core Functionality

When attacking the host, it is not enough for an attacker to control a DMA engine. The engine enables the attacker to read from and to write to host memory. However, in most cases the target memory address is not known. This section describes the core

functionality of DMA malware, i.e., overcoming address randomization, memory mapping, and search space restriction.

The attacker has to determine memory addresses. The problem is that the memory space allocated for, e.g., kernel data structures is not at the same memory address after a platform reboot. Data structures are placed *randomly in memory* by the OS. This can happen in a natural way when a device driver, for example, allocates memory and gets the next free unallocated memory chunk. The memory address of that chunk is not necessarily the same after a platform reboot. Alternatively, the OS can apply certain randomization algorithms to ensure that data structures are not placed at the same memory position. Of course, an attacker can scan the whole system memory for signatures of the target data, but this is very inefficient when scanning a system with 4 GB physical memory or more.

Operating systems work with *virtual memory addresses* [see 31, Chap. 15]), but DMA works with *physical memory addresses*. The OS creates so-called page tables that are used by the host CPU to map virtual memory addresses to physical ones. The mapping is absolutely necessary to resolve memory address pointers when using DMA. A special host processor control register called `CR3` contains the physical memory address of the page tables. The attacker has no access to the `CR3` register. The visibility of a DMA engine is restricted to host memory only. Without further analysis the attacker has to scan the whole memory address space for relevant data. There are two potential ways in which an attacker can overcome this problem. The first way is to analyze if the OS places the data structures in question in approximately the same memory area. The second possibility is to implement OS memory management mechanisms. That is, the attacker must find a way to access memory page tables created by the OS. With access to the page tables the attacker can then traverse page tables and is able to resolve pointers from one data structure to another. This still requires a known starting point for the search.

4.3 Design and Implementation of DAGGER

We present an overview of a general design for our DmA-based keystroke loGGER DAGGER in the next subsection before we explain the details of the DAGGER implementation in Sect. 4.3.2.

4.3.1 General Design

Our design of DAGGER is depicted in Fig. 4.1. DAGGER is DMA malware. That is, DAGGER has to fulfill the DMA malware definition including at least the criteria C1, C2, and C3. DAGGER consists of three main components:

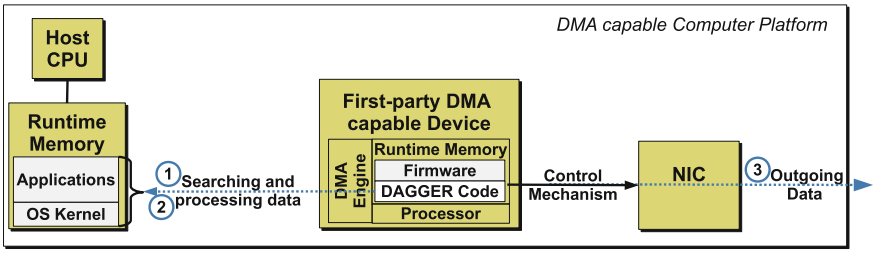

Fig. 4.1 General design of DAGGER. DAGGER is executed on a DMA capable device so that it can (*1*) search and (*2*) process data from host runtime memory. It controls a communication path to exfiltrate information (*3*)

- **Search:** find the address of valuable data in the host memory via DMA.
- **Process data:** read valuable data within the regions identified during the search process.
- **Exfiltration:** exfiltrate information in a way that is invisible to the host.

4.3.2 Implementation Based on Intel's ME Environment

To evaluate DMA malware we chose to implement DAGGER on Intel's ME. Intel's ME provides some useful features for implementing DMA malware that we describe in the following.

The core of Intel's ME is an embedded micro-controller placed in the platform's MCH. This isolated environment contains *Read Only Memory* (ROM), *Static Random Access Memory* (SRAM), DMA hardware to access the host memory [25, 131], and a processor as depicted in Fig. 4.2. The embedded processor of the ME is an ARCtangent-A4 (ARC4). The isolated environment is available regardless of the power state, even in standby or power on/off. It only requires that the chipset is connected with a power source. Applications executed on the embedded micro-controller are implemented in firmware (ME FW) and stored in flash memory together with the BIOS. The most prominent ME firmware example is Intel's Active Management Technology. But depending on the kind of computer platform (business or consumer hardware) the ME can also run other firmware. Other firmware executed by Intel's ME are for instance: Intel's Identity Protection Technology, *Alert Standard Format* [131, p. 46]), *Intel Quiet System Technology* (QST [131, p. 46]) for temperature and fan control, and *Integrated Trusted Platform Module* (iTPM [79, p. 109]).

ME firmware can communicate with the host via a PCI device called *ME Interface* (MEI [79, p. 71]). The MEI can provide the version of the executed ME firmware, for example. The ME environment provides additional PCI devices[1] to support certain

[1] These devices can act as bus masters, see Sect. 2.5.

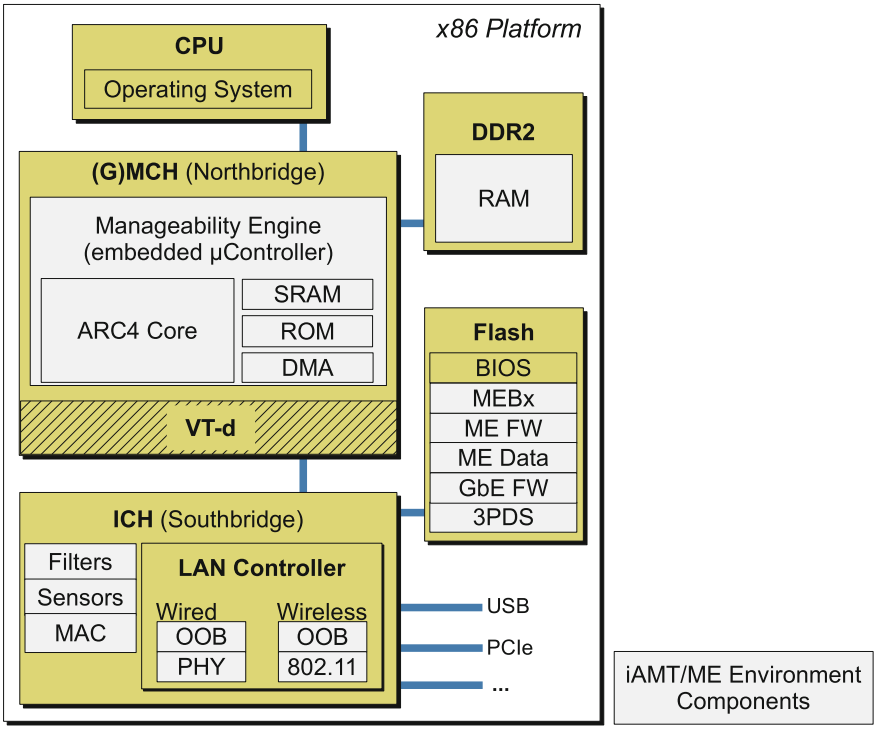

Fig. 4.2 Intel's Manageability Engine environment. Intel's Manageability Engine (ME) environment consists of the Manageability Engine that is included in the MCH. Furthermore the environment consists of an isolated part of the RAM as well as isolated portions of persistent flash memory. The ICH also contains ME environment components, especially components that implement the out-of-band channel

AMT features such as text console and disk redirection. A serial port is emulated to implement text console redirection [see 79, Chap. 5]. Text output that is sent to this port is forwarded to a remote console via the network. With this capability an administrator can remotely control the BIOS. To implement disk redirection a local disk is emulated by the ME environment [see 79, Chap. 5]. An administrator can remotely mount storage media (e.g., a CDROM with an operating system installer to recover the operating system of the AMT enabled platform) via the locally emulated disk.

During the platform power-on procedure the ME firmware image is loaded into ME RAM. The ME firmware itself runs on the micro-controller internal ARC4 processor and it also uses some system RAM as depicted in Fig. 4.2 to store runtime data. This runtime storage is provided by a certain memory area that is invisible to the main CPU and the OS. The separation is enforced by the chipset [79].

The ME environment introduces *Out-Of-Band* (OOB) communication, i.e., a special network traffic channel used by iAMT. The iAMT enabled computer platform is managed by a remote management console using OOB. OOB is also available

regardless of the power state. OOB can be considered to be a separate network connection, running on the same hardware. The ICH implements necessary components to support the ME environment with the OOB feature. The firmware filters network traffic intended for, e.g., iAMT and redirects the packets to the ME. The host is unaware of the redirected ME network traffic. This kind of traffic is identified by TCP port numbers.

4.3.3 Attack Implementation Details for Linux and Windows Targets

We implemented two keystroke logger prototypes to attack two targets, Linux and Windows based OSes. We decided to find and monitor the keyboard buffer address of 32 bit versions of the target OSes. In comparison to 64 bit versions, 32 bit versions have to deal with a more complicated memory management. For example, the attacker has to consider *Physical Address Extensions* (PAE [105, p. 769]) or certain memory offsets when mapping memory addresses. The following subsections describe, how we implemented the DMA malware core functionality as described in Sect. 4.2. The prototypes capture short living keystroke codes within their *monitoring phase*. Each prototype handles the *search phase* for the target buffer differently. This has at least two reasons. One reason is to evaluate as many aspects as possible of DMA malware. The other reason is that OSes have different memory management properties. We use a vulnerability described by Tereshkin and Wojtczuk [131] to infiltrate the ME environment during runtime. To call our code we hook a ME firmware function that we identified as the library function `memset`. Tereshkin and Wojtczuk [131] assumed that they hooked a timer interrupt handler, but they actually hooked the ME firmware function `memcpy`. We hook `memset` since we determined that it is called more often.

Our Linux variant is based on a signature scan as depicted in Fig. 4.3. We analyzed the available Linux source code to derive a signature of our target, the physical address

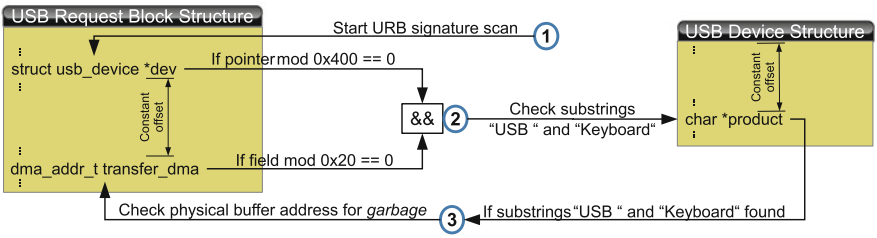

Fig. 4.3 USB request block signature scan (simplified). The scan (*1*) begins to search for a pointer to the USB device structure. A candidate for such a pointer is aligned to a `0x400` boundary. The value of the structure field `transfer_dma` must be aligned to a `0x20` boundary. If both conditions are true, the product string in the USB device structure is (*2*) checked for the substrings "USB" and "Keyboard" In the last step the signature scan (*3*) checks if the keyboard buffer contains *garbage*, that is, invalid keystroke codes

of the keyboard buffer. The buffer address is part of the *USB Request Block* (URB) structure that is defined in the file `include/linux/usb.h` of the Linux source code. The demanded structure field is called `transfer_dma`. The memory offsets differ from kernel version to kernel version. We solved that problem by exploiting the *Grand Unified Bootloader* (GRUB) that places an identifier at a constant physical memory address. We implemented a function that reads the identifier via DMA and parses the kernel version number to derive corresponding offsets. Afterwards our prototype runs through the search phase, that is, the signature scan.

Since our Linux prototype targets kernel data structures we can restrict the search space to the first gigabyte of system RAM. Standard Linux systems have a memory split of 1 GB/3 GB, that means, 1 GB for kernel space and 3 GB for user space. We were able to further restrict the search space by empirically analyzing in which memory area the kernel places the data structures needed by our signature scan. We determined that this memory area is between `0x33000000` and `0x36000000` for the Ubuntu Linux kernel version 3.0.0 after a fresh platform boot. The address of the keyboard buffer does not change after standby or hibernate mode. With this approach we overcome the problem of inefficiently scanning the whole system memory for the randomly placed signature. Mapping virtual addresses to physical ones is a minor issue when attacking the Linux kernel. Normally, in 32 bit versions a kernel virtual address (or more precisely kernel logical address [see 31, Chap. 15]) is mapped to its physical address by subtracting a constant offset. In 64 bit Linux versions such an offset is not needed. Hence, there is no need to know the content of the `CR3` processor register.

The search strategy for Windows-based target platforms works different. To be able to perform the search using the search path as described below, virtual addresses must be mapped to physical ones. This mapping is done using page tables created by the Windows kernel. The memory address of those page tables is loaded into the `CR3` register, which an attacker cannot access via DMA. It turned out after some empirical tests with a simple driver, that the physical address of the page tables for the *system process* takes one of the following two values for Windows Vista/7 systems: `0x122000` or `0x185000`. The system process is the first process created during Windows startup. With this knowledge DAGGER can access the page tables created by the kernel and overcomes the problem of mapping virtual addresses to physical ones. DAGGER implements a page table traversing algorithm that takes account of PAE.

Our Windows malware searches for a structure called `DeviceExtension` that is maintained by the USB keyboard driver `kbdhid.sys`. This structure contains a buffer that stores the codes of the last pressed keys. The source code for `kbdhid.sys` is not publicly available. The most convenient way to get internal information of that driver was to use *IDA Pro*,[2] *Windows Debugger* (WinDbg) tools, and debug symbols provided by Microsoft[3] in the form of `pdb` files. To finally

[2] See http://www.hex-rays.com/products/ida/index.shtml [accessed 25 February 2014].

[3] See http://msdn.microsoft.com/en-us/windows/hardware/gg462988 [accessed 25 February 2014].

determine the location of the buffer in the `DeviceExtension` structure, our research starts early in the boot process [see 105, Chap. 13]. We analyzed further internal Windows structures. To find a starting point for the search, we analyzed the *Kernel Processor Control Region* (KPCR [105, p. 62ff]), or more precisely `KiInitialPCR`, the KPCR for the processor 0. We also examined the *Object Manager Namespace Directory* (OMND, part of the Windows object manager). We determined that `KiInitialPCR` is well suited to derive a path to the `DeviceExtension` structure as depicted in Fig. 4.4. `KiInitialPCR` is not located at a constant memory address. DAGGER has to apply another step before it can start with the search as depicted in Fig. 4.4.

The memory position of `KiInitialPCR` is determined by a function called `OslpLoadAllModules` of the `winload.exe` binary as depicted in Fig. 4.5. This binary is loaded by the Windows boot manager `bootmgr` that in turn is loaded by *Master Boot Record* (MBR) code, etc. The function loads the *Hardware Abstraction Layer* (HAL) library `hal.dll` as well as the Windows kernel image in a more or less random manner. The kernel image contains `KiInitialPCR` at a constant

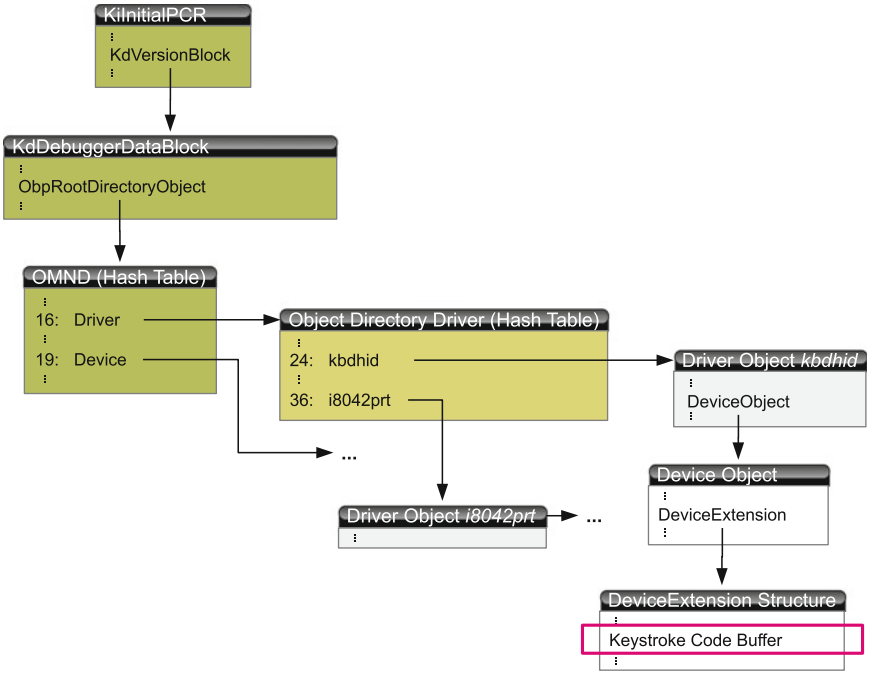

Fig. 4.4 Find `DeviceExtension` structure (simplified). With `KiInitialPCR` as a starting point, DAGGER finds the OMND, that provides via hash tables a path to the driver object `kbdhid`. This object contains a pointer to a device object. The device object provides the `DeviceExtension` structure, which contains the keystroke code buffer

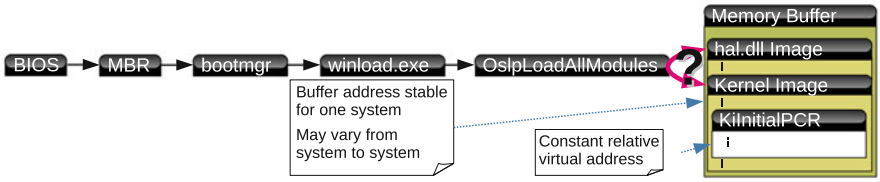

Fig. 4.5 Find `KiInitialPCR` (simplified). `OslpLoadAllModules` determines the exact position of the Windows kernel image and the HAL

relative address. The disassembled code of `OslpLoadAllModules` is similar to an *Address Space Layout Randomization* (ASLR [105, p. 757]) mechanism.

The memory buffer for the kernel image and the HAL is allocated by `OslpLoadAllModules` via a function called `BlImgAllocateImageBuffer`. The latter function returns stable address values for a Windows system. These values may vary on different systems. For every possible return value of the function `BlImgAllocateImageBuffer` there are 64 theoretically possible different 4 KB aligned virtual addresses. These addresses need to be checked in order to find the kernel image base address. The disassembly of `OslpLoadAllModules` revealed that the randomization seed for the address randomization has a 5 bit value. This implies 32 possible addresses for each (of two) possible load order cases, i.e., first kernel image and then `hal.dll` or vice versa. As long as `KiInitialPCR` has a constant relative virtual address within the kernel image, the same number of virtual addresses to be checked also applies for a direct `KiInitialPCR` search without any need to deal with the kernel image. To ensure that DAGGER found the correct `KiInitialPCR` we implemented a `KiInitialPCR` signature check. When DAGGER has identified the correct `KiInitialPCR`, it continues to look for the keyboard buffer using the search path described in Fig. 4.4.

We use ethernet controller to exfiltrate the captured keystroke codes. To be more precise, we use the OOB features of the Intel ME environment. Unfortunately, there is no documentation that explains how to use this feature. Hence, we had to analyze the firmware to figure out how to exfiltrate keystroke codes using the OOB channel. We were able to find the transmit ring buffer that is used to send network packets in the ME runtime memory. Furthermore, we were also able to find the firmware code that is responsible for sending the next network packet from the transmit ring buffer. To exfiltrate the captured data we prepare network packets, e.g., DHCP discover packets as depicted in Fig. 4.6, that contain the logged keystroke code. Then, we copy the prepared network packet to the transmit buffer. Afterwards, we trigger sending the packet by the NIC to an external platform. Please note, the transmitted packets can easily be found when analyzing the network traffic with an external platform. To improve the stealthiness of the design we [124, 125] implemented a covert timing channel that is based on a so-called Jitterbug [see 115].

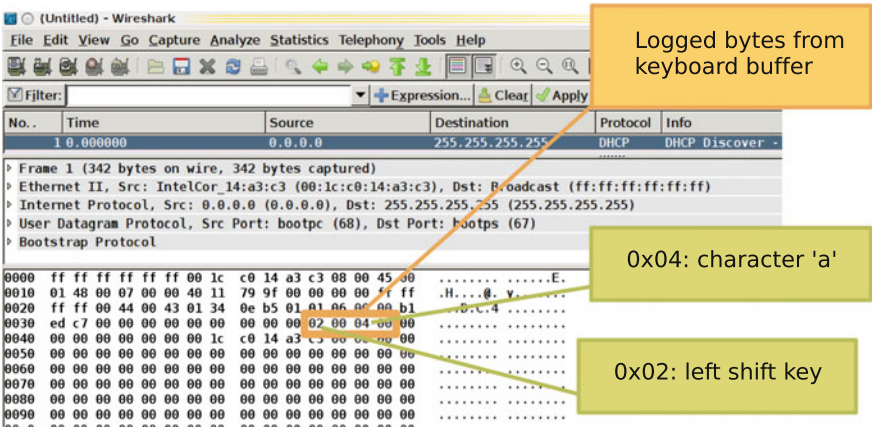

Fig. 4.6 Network packet containing bytes from keyboard buffer. The wireshark instance is executed on an external platform. The network packet that has been parsed by wireshark contains 4 bytes that represent the logged keystroke code data

4.4 Evaluation

We used an x86 platform with a Q35 chipset, 2 GB RAM, a 4-core 3 GHz CPU, and iAMT firmware (version 3.2.1) to evaluate DAGGER with four different 32 bit OS kernels: Windows Vista Business (Service Pack 2), Windows 7 Professional (Service Pack 1) and Ubuntu Linux kernel version 2.6.32 as well as kernel version 3.0.0.

4.4.1 DMA Malware Fulfillment

We designed and implemented our DAGGER prototypes according to the DMA malware definition described in Sect. 4.1. (C1) is clearly fulfilled since DAGGER implements working keystroke logger functionality. DAGGER needs no physical access for the infiltration process (C2). We infiltrate the ME environment using a software-based exploit during runtime. DAGGER exploits dedicated hardware to implement rootkit properties (C3). We ran host performance overhead tests (memory: MEM, network: NET, and CPU), since host and ME environment share the NIC as well as a RAM chip. Parallel NIC and RAM accesses must be arbitrated and could therefore cause delays. Our measurement results depicted in Fig. 4.7 reveal no significant overhead. The highest overhead that we could detect is approximately 1.5 % when accessing the host memory during the search phase. It is extremely unlikely that this minimal overhead would reveal DAGGER.

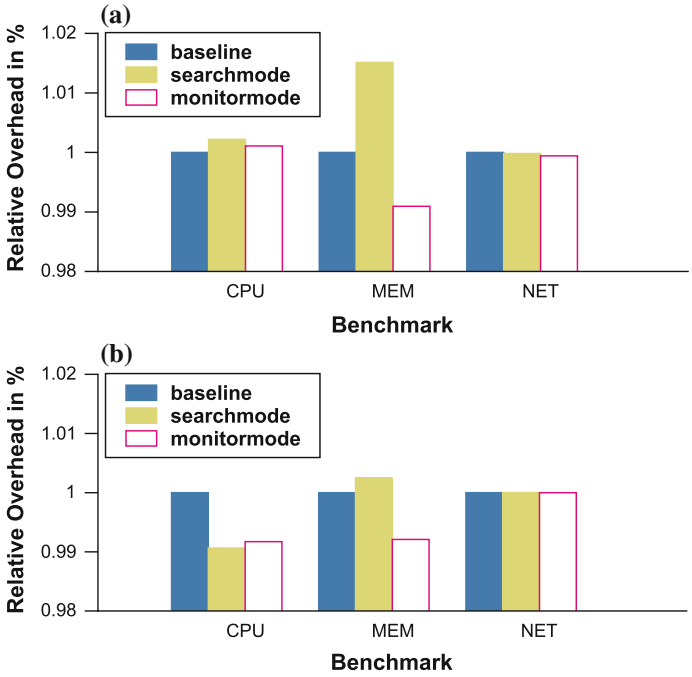

Fig. 4.7 Host performance CPU, MEM, and NET overhead tests. **a** Linux 3.0.0 performance overhead test results. **b** Windows 7 performance overhead test results. We used time stamp counters to measure overhead time. We measured the time it takes to copy a 100 MB test file over the network (NET) and within RAM (MEM) as well as the time it needs to compute a SHA1 hash sum over this test file ten times in parallel to stress all four CPU cores (CPU). Each benchmark was performed three times: without keystroke logger (*baseline*), keystroke logger in search mode, and keystroke logger in monitoring mode. For the monitoring mode we configured the keystroke logger to constantly send network packets of approximately 1,000 packets per minute. This is equal to 500 keystroke and 500 key release events. We repeated each test 1,000 times. A bar in the gure represents the mean of 1,000 runs

The search times summarized in Fig. 4.8 are very short and the very aggressive memory stress test we performed does not represent the memory utilization of a normal computer system. DAGGER has solely read-only operations to ensure stealthiness. The popular network sniffer *Wireshark*[4] was not able to detect any DAGGER traffic on Linux and Windows systems. Host firewalls cannot block such traffic either. Even if anti-virus software knew DAGGER's signature it would be unable to access DAGGER's memory to apply the signature scan successfully. Nevertheless, we also run a software called *Mamutu*,[5] that is, amongst other things, specialized in detecting keylogger behavior. Even specialized software could not find any indication of DAGGER. Regarding criterion C4 we successfully checked if

[4] See http://www.wireshark.org/ [accessed 25 February 2014].

[5] See http://www.emsisoft.com/en/software/mamutu/ [accessed 25 February 2014].

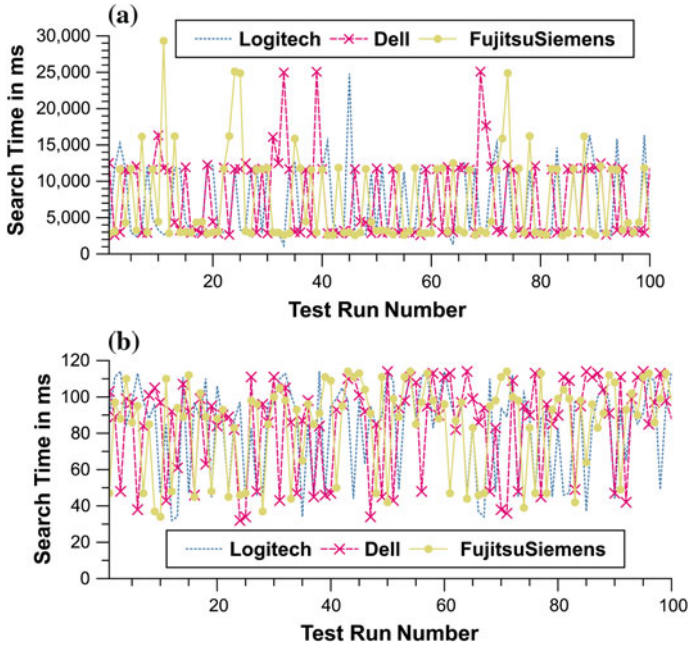

Fig. 4.8 Search time measurement results. **a** Linux 3.0.0 several keyboards **b** Windows 7 several keyboards. The test results with several keyboards under Linux reveal a best case for search times of around 1,000 ms and a worst case of almost 30,000 ms as depicted in (**a**). The median for all keyboards is at 3,281 ms. Useful for comparison: scanning the whole memory area determined for Linux (see Sect. 4.3.2) search takes approximately 13,000 ms. The worst case of 30,000 ms is due to an erroneous DMA transfer that we do not handle directly. This causes DAGGER to repeat the search phase. On Windows 7 the best search time is approximately 50 ms and the worst time is around 120 ms, see (**b**). The median for all keyboards is at 93 ms. Hence, the search strategy we implemented for Windows targets performs much better than the signature scan based strategy for Linux

DAGGER's attack code is fully functional after a platform reboot, after standby and after power off state. We determined that this depends on an iAMT BIOS option. Our code cannot survive a cold boot that happens if this option is not set.

4.4.2 Effectiveness and Efficiency

DAGGER is efficient, since it can permanently catch short living data from the keyboard buffer. To demonstrate that DAGGER is also effective we tested DAGGER with different Windows and Linux versions as well as several keyboards. The measured search times summarized in Fig. 4.8 confirm that DAGGER is quite efficient. We repeated the measurements for each kernel and for each keyboard 100 times. We took a measurement after a platform (re)boot to change the target address for

each test run. The Linux measurement results imply that we could further restrict the search space. We could start the search near the lowest address we encountered most often during our tests. Search times of around 2,500 ms are due to target addresses near `0x33c00000`. Thus, we could skip almost 2,500 ms if we start the search at `0x33c00000`. Furthermore, we could skip the search area address range between `0x34000000` and `0x36000000`. Almost no targets were found in this area. A lot of targets were found near `0x36e0000`, i.e., search times of around 12,500 ms that could also be saved. This increases the probability to miss keyboard buffer addresses. That is, we can get better search times at the expense of effectiveness. The best case search times are sufficient to capture hard disk encryption passwords, for example. We tested this successfully with a Linux system. The Windows kernel can swap out memory pages to the hard disk—Linux does not. Swapped memory pages cannot be found by DMA malware. Hence, we also did a test for Windows to check if swapping has any effect on DAGGER as depicted in Fig. 4.9b.

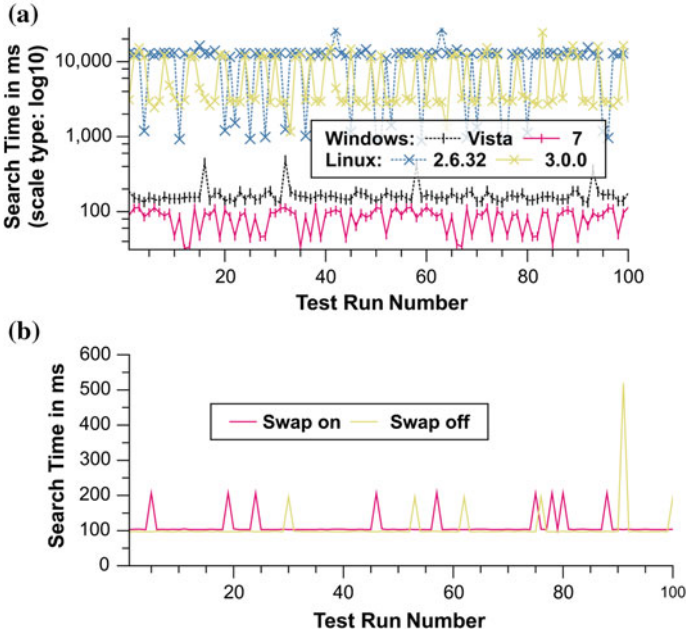

Fig. 4.9 Search time measurement results. **a** Several operating systems. **b** Windows 7 swap on/off. The plot in (**a**) compares different target kernels. DAGGER performs slightly better on Windows 7 than on Windows Vista. Linux 2.6.32 places the target memory structure closer to `0x33000000` than Linux 3.0.0. Thus, DAGGER has more hits around 1,000 ms when attacking Linux 2.6.32. The results in (**b**) confirm that swapping has no effect on the efficiency and effectiveness of DAGGER. A platform reboot was only applied to change the swapping behavior. The peaks are due to restarts of the search phase

4.4.3 ME Firmware Condition

To be really stealthy DAGGER ensures that the ME firmware is still up and running correctly. iAMT provides a web server for remote platform management [see 79, p. 215] that is still usable. The server responds correctly on the local platform on Linux and Windows. Firmware tools utilizing the MEI (see Sect. 4.3.2) also work when DAGGER is active. We successfully tested the *AMT Status Tool* (part of the *Local Manageability Service* driver) and the *Manageability Connector Tool* (part of the *Manageability Developer Toolkit 7.0*) under Windows. Under Linux we successfully tested the *Intel AMT Open-source Tools and Drivers* (version 5.0.0.30), or more precisely the *ME Status* and the *ZTCLocalAgent* tool. Note, we determined that DAGGER still runs even after having disabled the iAMT firmware in the BIOS. It appears that the ME environment cannot be disabled entirely via any BIOS options.

4.4.4 I/OMMU

To test an I/OMMU (see Sect. 2.6) as a countermeasure against DAGGER we enabled Intel VT-d in the BIOS. As far as we know Windows does not support I/OMMUs directly. We could successfully attack Windows Vista and Windows 7 although the I/OMMU was activated. Linux supports I/OMMU configuration with additional effort. We also enabled VT-d in the BIOS and we activated I/OMMU support via the kernel command line. With these additional steps we were able to prevent the Linux version of DAGGER from reading short living keystroke codes from OS memory. This protection is not activated by default. In the next section we discuss, among other things, further issues regarding the I/OMMU.

4.5 Countermeasures Considerations

To scan for DMA malware using software executed on the host CPU is quite difficult. For example, current AV software does not scan the runtime memory of peripherals or the host CPU cannot access the runtime memory due to certain isolation mechanisms. The worst case for a scanning approach is that the DMA malware changed the behavior of the scan software, which would deliver incorrect results. Checking firmware images at load time, as proposed by the TCG [136], does not prevent runtime attacks. Furthermore, it is unclear if all ROM components are accessible by the host.

4.5.1 I/OMMU Issues

In the case of DMA attacks an appropriate configuration of the I/OMMU (see Sect. 2.6) is proposed as a preventive countermeasure, for example by Duflot et al. [47, p. 48]. It is required that system software configures the I/OMMU. An incorrect configuration cannot be excluded [83, p. 2].

It is assumed that the I/OMMU is secure. Unfortunately this is not always the case. Sang et al. [111] demonstrated that an I/OMMU configuration can be tricked with legacy PCI devices. Wojtczuk et al. [148] revealed that an I/OMMU can be attacked by modifying the number of DMA remapping engines provided by the BIOS (see Sect. 2.6). This is done before the I/OMMU is configured by system software. The environment we used for DAGGER is able to carry out such an attack. This threat can only be mitigated by executing special hardware dependent code called `SINIT`. However, on at least one previous occasion the manufacturer of the chipset failed to release `SINIT` code at the launch of the chipset [147, p. 22]. This code is needed to initialize a well known and trustworthy environment for, e.g., a hypervisor. It checks the DMA remapping engines and can therefore prevent an attack as presented Wojtczuk et al. [148].

`SINIT` belongs to and increases the size of the trusted computing base. Previous work demonstrated that `SINIT` code can have exploitable security vulnerabilities that can be used to trick I/OMMU mechanisms [see 148]. Recently, Wojtczuk and Rutkowska [148] presented another attack that can be used to circumvent I/OMMU mechanisms as well. To prevent the attacks presented by Wojtczuk and Rutkowska [146, 148], a `SINIT` as well as a BIOS update must be applied. Wojtczuk et al. [147] presented another I/OMMU attack. Note, `SINIT` is normally triggered on hypervisor-based platforms. Platforms running a normal OS cannot necessarily count on the I/OMMU. It should also be mentioned that `SINIT` requires the activation of additional platform features, namely the *Trusted eXecution Technology* and the TPM [54]. This means that users that do not want to activate the TPM for example cannot rely on the I/OMMU. Note, the TPM is an opt-in device [see 54, p. 212] and is turned off by default.

For a comprehensive protection against DMA malware it is absolutely necessary to correctly configure the I/OMMU. However, the I/OMMU can only be considered secure if the above mechanisms to protect the whole platform are secure. This is a difficult task. Hence, alternative approaches were considered by Li et al. [83] and Duflot et al. [46]. Li et al. [83] state that their approach requires extending the firmware, does not work correctly if peripherals cause heavy PCIe traffic, and the verifier component needs to know the exact hardware configuration. The approach presented by Duflot et al. [46] is highly NIC adapter-specific and not applicable to isolated environments such as Intel's ME. It is worth noting that malware such as our implementation controls the NIC without any NIC firmware modifications, i.e., exfiltration cannot be detected by the approach described by Duflot et al. [46]. Furthermore, this approach has significant performance issues for the host CPU (100 % utilization of one CPU core).

Memory access policies enforced by I/OMMUs can be insufficient or can even prevent the use of some other features in some application scenarios. Consider hardware supported malware scanners such as *CoPilot* [100] and *DeepWatch* [25]. The I/OMMU can be configured to stop CoPilot and DeepWatch from working or to allow such systems to access the host memory to scan it for malicious software. In the latter case DMA malware could make use of the execution environment of CoPilot or DeepWatch to attack the host. DAGGER, for example, uses the DeepWatch environment, i.e., Intel's ME. Since iAMT version 5, Intel supports a verified launch for the firmware to be executed on Intel's ME [see 79, p. 271]. The firmware is checked during load time. The result of the load time check is provided to system software. As far as we know the result is not used in practice. The mechanism cannot prevent runtime attacks as applied by our PoC. This means, DAGGER confirms that our assumption that an attacker already infiltrated the target system, e.g., via a zero-day exploit (see Sect. 2.7), can also hold even if such additional security mechanisms are in place. An appropriate configuration of the I/OMMU is a first step against DMA malware. However, without resolving the mentioned issues a successful deployment cannot be guaranteed.

4.5.2 Detection Approach Based on DMA Side Effects

A possible detection approach is based on DMA side effects that we observed in a first experiment with our own DMA malware prototype DAGGER. Our detection mechanism is based on multiple widely used and cross platform CPU features.

So far we developed, implemented, and evaluated our mechanism that is able to detect rogue DMA usage that is not initiated and unexpected for the host system. DMA usage is initiated by the host CPU when a peripheral has to process data on behalf of the host CPU. Sending a network packet using the network interface card is an example. Expected DMA usage originates from peripherals and is intended for software running on the host CPU such as the operating system. Receiving a network packet is an example for intended DMA usage. Our method is able to detect a general side effect pattern. Thus, we believe it is suited to detect other kinds of DMA malware besides the prototype we implemented. Our investigation into detecting malicious DMA usage is based on the knowledge that both, the main CPU and platform peripherals, can request to access the main system memory at the same time. The memory controller hub arbitrates parallel memory access requests, see Fig. 2.5. The interesting question for us was if this parallel memory access introduced any measurable side effects. If side effects are present and measurable then we can use these to detect malicious behavior.

We booted a Linux kernel and started just a root shell to ensure that the system workload was minimized. Only one CPU core was online. We performed a memory stress three times: without keystroke logger (baseline), keystroke logger in search mode, and keystroke logger in monitor mode, see also Sect. 4.3.3. For the tests we used a 100 MB file that we copied from one location to another within a

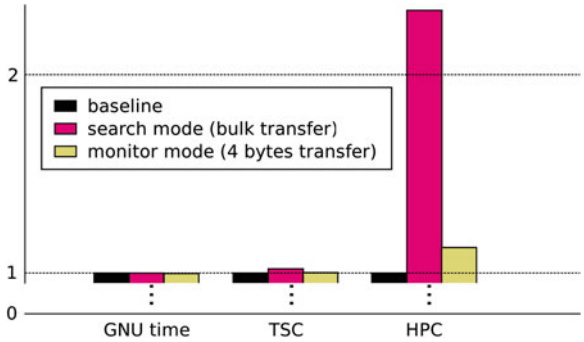

Fig. 4.10 Memory stress measurements. Search phase and monitor phase are depicted relative to the baseline

RAM-based file system. We repeated the tests 1,000 times and calculated the means. The results are depicted in Fig. 4.10. The diagram reveals how we refined our strategy with different and more specialized measurement tools.

GNU Time Measurements First we tried the common system tool GNU time to determine a delay. GNU time measures system resource usages of a process, in our case the memory stress test tool. As shown in Fig. 4.10 on the left hand side the means of the test runs are nearly the same. We concluded that the measurement resolution of GNU time is insufficient to reveal delays in our experiment.

Time Stamp Counter (TSC) Measurements We repeated our measurements with a more accurate hardware-based measurement tool, the TSC [see 69, Sect. 17.12]. The TSC counts clock ticks, see Sect. 2.3. The results are presented in the middle in Fig. 4.10. We were able to (re)produce an overhead of 2 % when our prototype malware is in search mode. DMA was originally introduced to eliminate the burden on the CPU. That means, to perform memory transfers without the involvement of the host CPU. *Hence, that overhead is surprising and a first piece of evidence that detectable DMA side effects exists.* When our prototype malware is in monitor mode we cannot see noteworthy overhead when using TSC. The critical difference between the two modes is that in search mode the malware copies at least a memory page where it searches for valuable data. However, in monitor mode the malware copies just 4 bytes from the keyboard buffer.

Hardware Performance Counter (HPC) Measurements We repeated the measurements with a third approach using HPCs, a hardware-based performance monitoring tool for code optimization, see Sect. 2.3. These counters are special purpose processor registers on Intel processors [69, Chaps. 18/19] that count certain events such as cache misses, branch prediction misses, and resource stalls. Similar HPC are also available on other platforms such as ARM and SPARC. The Intel platform we

used for our experiments supports 340 events.[6] We evaluated all of them and determined that resource stalls are a particularly effective DMA side effect. HPC events are more precise than TSC measurements for certain events. We assume the number of resource stalls are a direct result of the delays we can measure with TSC. As an example we present the result of a hardware performance counter called `RAT_STALLS:ROB_READ_PORT` (see Sect. 2.3) in Fig. 4.10. Compared to the baseline the overhead is more than double. Without our prototype malware the mean of our measurements was 1,359,898 counted events. With our prototype malware in search mode the mean was 3,161,868 counted events, and in monitor mode it was 1,535,054 counted events. The latter is only slightly higher compared to the baseline. The refined measurements demonstrate the more accurate we measure the better is the visibility of the DMA side effect.

Detection Based on our findings, DMA side effects can be measured. This means we can design a DMA malware detection mechanism. The mechanism works by establishing a measurement baseline and reference values for the TSC/HPC. During runtime, our system monitors the TSC/HPC values and compares them to the reference values. If the values deviate from the reference values DMA malware is detected. We acknowledge that an actual implementation of this delay-based detection approach needs some additional investigation. In Chap. 5 we present a more enhanced detector that is also based on HPC. Furthermore, the artificial memory stress is not required anymore to detect DMA malware with our enhanced method. In this section we discuss the I/OMMU and a detection approach based on DMA side effects as countermeasures.

4.6 Chapter Summary

In this chapter we studied DMA malware, i.e., malware hidden in dedicated hardware. Such malware can circumvent protection mechanisms run on the host CPU by directly accessing the host memory. We implemented and evaluated DAGGER, a DmA-based keystroke loGGER. The dedicated hardware enables our prototype to benefit from rootkit properties. DAGGER operates stealthily. It is undetectable by anti-virus software etc. We can conclude that DAGGER is a representative malware proof of concept when comparing it with other known DMA malware. Hence, we will reuse DAGGER in the next chapters to develop a reliable DMA malware detector.

DMA malware is more than controlling a DMA engine. Our evaluation confirmed that DMA malware is efficient even if obstacles such as memory address randomization are in place. We also demonstrated that DMA malware can be effective, that is, it can attack several OSes. This confirms that DMA malware is stealthy at no costs regarding efficiency and effectiveness.The host has no reliable means to protect itself.

[6] We used the Performance API, that is available at http://icl.cs.utk.edu/papi/software/index.html [accessed 25 February 2014], to work with HPC in the described experiment.

Throughout this chapter we highlighted that the I/OMMU has several issues and the host cannot necessarily count on this preventive countermeasure against DMA malware. Besides possible vulnerabilities and various preconditions that must be fulfilled for a successful I/OMMU deployment, the most obvious issue is that common OSes do not or do insufficiently support the I/OMMU. Hence, DMA malware can attack OSes such as Windows. A general and reliable approach for scanning the dedicated devices for malware does not exist. A reliable and more general DMA malware detection mechanism is needed. Other researchers have also investigated I/OMMU alternatives.

In this chapter we discussed an alternative approach. Our detection approach is based on the observation that parallel memory accesses from the isolated hardware (via DMA) and the main CPU produce measurable side effects. Hence, we can conclude that illegitimate DMA operations are not stealthy anymore. Nonetheless, we have to admit that the experimental setup used for the detection is rather artificial. We conclude that the current setup is insufficient for a detection tool that can be applied in practice. However, we demonstrated that hardware performance counters can be the basis for a reliable detection tool. We revealed that the measurement tool requires a sufficient measurement resolution. Hardware performance counters fulfill this requirement. We will further investigate this point in more detail in the next chapter.

Without an alternative, only dedicated hardware whose inner workings is accessible by the host, i.e., complete RAM and ROM access, should be deployed. This enables the host to check the device for malicious modifications from time to time. A precondition for this is a reasonable measurement strategy and that the scanner gets loaded first. Devices with a dedicated processor, dedicated runtime memory, and a DMA engine are a threat for the host platform. This chapter demonstrates that additional protection mechanisms are needed to ensure a platform's confidentiality, integrity, and especially its trustworthiness.