# Understanding DMA Malware

Patrick Stewin[¶] and Iurii Bystrov[§]

Security in Telecommunications — Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
[¶]{patrickx}@sec.t-labs.tu-berlin.de
[§]{l.inc}@mailbox.tu-berlin.de
http://www.sec.t-labs.tu-berlin.de/

**Abstract.** Attackers constantly explore ways to camouflage illicit activities against computer platforms. Stealthy attacks are required in industrial espionage and also by criminals stealing banking credentials. Modern computers contain dedicated hardware such as network and graphics cards. Such devices implement independent execution environments but have direct memory access (DMA) to the host runtime memory. In this work we introduce DMA malware, i. e., malware executed on dedicated hardware to launch stealthy attacks against the host using DMA. DMA malware goes beyond the capability to control DMA hardware. We implemented DAGGER, a keylogger that attacks Linux and Windows platforms. Our evaluation confirms that DMA malware can efficiently attack kernel structures even if memory address randomization is in place. DMA malware is stealthy to a point where the host cannot detect its presense. We evaluate and discuss possible countermeasures and the (in)effectiveness of hardware extensions such as input/output memory management units.

**Keywords:** Dedicated Hardware, Direct Memory Access, I/OMMU, Keylogger, Malware, Manageability Engine, Rootkit, Stealth, vPro, x86

## 1 Introduction

Recently the arms race between malware developers and the anti-malware community reached a new level. Countermeasures for kernel level [16], hypervisor based [20], and system management mode based malware [12] were proposed [13, 26, 5]. 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 [30, 31, 11]. Such devices bring, among other things, their own processor and 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 a mechanism called *Direct Memory Access* (DMA), to easily circumvent protection mechanisms built into the *Operating System* (OS) by attacking host runtime memory directly. We

call code performing targeted DMA based stealthy attacks to find 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 paper we classify DMA attacks and derive the term DMA malware. We explore the term in more detail 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 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* (ME) that is part of the popular x86 platform. Intel's ME is implemented in business as well as consumer platforms to support different applications, such as the *Intel Active Management Technology* (iAMT) [21] or the *Identity Protection Technology* (IPT) [19] (see Intel vPro platforms [18], for example).

Our DMA malware DAGGER is not executed on the host processor. It is executed on the processor provided by Intel's ME. No additional hardware is required. DAGGER implements a sophisticated 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 work 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 prove 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 so 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.

*Paper Organization.* Section 2 introduces necessary background. Our assumptions and attacker model are presented in Section 3. A classification of DMA code and a definition for DMA malware is given in Section 4. In Section 5 we present DMA malware core functionality. The design and implementation of our DMA malware is presented in Section 6. Section 7 describes the evaluation of DAGGER, Section 8 considers countermeasures and discusses in particular I/OMMU issues, and Section 9 presents related work. We conclude in Section 10.

## 2 Technical Background and Preliminaries

The target platform for our evaluation is a modern Intel x86 based system. This section introduces the most important terms regarding the target platform.

### 2.1 Typical x86 System Architecture

The main components of a typical x86 system architecture as depicted in Figure 1 are a *Central Processing Unit* (CPU or host processor), a *Memory Controller Hub* (MCH, also known as northbridge) and an *Input/output Controller Hub* (ICH, also known as southbridge). The combination of CPU, MCH, ICH is called the chipset [14]. 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) 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. Further controller devices connect other formats, such as *Universal Serial Bus* (USB), *FireWire*, or *Serial Advanced Technology Attachment* (SATA), via PCIe to the system. Legacy PCI devices are connected to the PCIe architecture via a so called *PCI-to-PCIe bridge* [4]. In Laptop computers *Personal Computer Memory Card International Association* (PCMCIA)/*ExpressCard* 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 in the platform's MCH. They also utilize separate processors as well as separate RAM to execute firmware.

### 2.2 Direct Memory Access

PCIe supports DMA for peripherals for fast memory access without the involvement of the host CPU. The aim of DMA is to remove the burden from the host

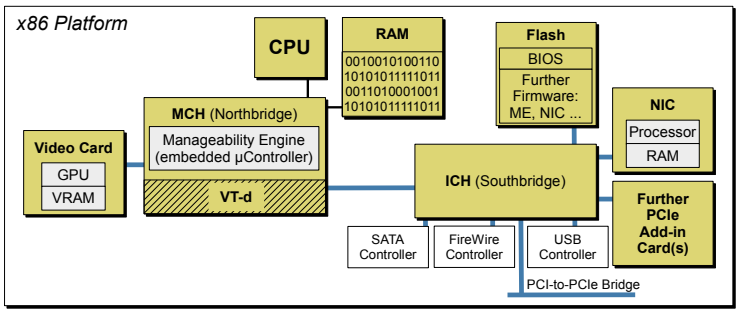


**Fig. 1.** x86 Chipset and Peripheral Components

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 [29, p.428]. Another mechanism is third-party DMA [29, p.428] where a central *DMA Controller* (DMAC) is necessary to provide legacy devices without DMA engines with fast memory access. It is also integrated in modern platforms [17, p.128].

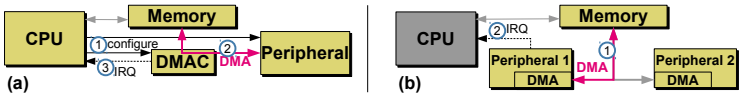


**Fig. 2. (a)** Third-party DMA: The host CPU (1) configures (source and destination address) the central DMA controller to (2) perform a DMA tranfser. The DMA controller (3) interrupts the host CPU when the DMA transfer has been finished [15, p.700]. Hence, the host CPU is aware of a third-party DMA transfer. — **(b)** First-party DMA: The peripheral device can (1) configure its own DMA engine. The device acts as bus master 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 completely unaware of the DMA transfer.

Figure 2 highlights an important difference regarding stealthy operation between third- and first-party DMA. When using third-party DMA the host CPU is aware of the DMA transfer, 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.

### 2.3 Input/Output Memory Management Units

Intel introduced a technology called *Intel Virtualization Technology for Directed I/O* (VT-d) [1] 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 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.

## 3 Assumptions and Attacker Model

The attacker model describes the setting of 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 [11], for example. The dedicated hardware supports DMA as described in Section 2. We assume that this 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 completely stealthy attack can result in a successful attack. Hence, the attacker wants to hide the attack by using the stealth potential of dedicated hardware. 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 aquire online banking credentials, etc.

## 4 DMA Malware Definition

To determine a definition for the term DMA malware we first classify 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 implementing malicious functionality is considered as 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 for our classification system. 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

With this prioritization we can derive a binary based classification:

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

This classification system covers 16 classes of DMA based code. We can derive a unique number for each class. 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 classified with the binary pattern 0110, that is class 6 in decimal. The higher the class, the more dangerous is the DMA based code. Note, we use this classification system to compare related work in Section 9.

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 Section 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.

## 5 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 and to write to host memory. However, in most cases the target memory address is not known.

**Overcoming Address Randomization.** 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.

**Memory Mapping.** Operating systems work with *virtual memory addresses* [6, Chapter 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.

**Search Space Restriction.** Without further investigations the attacker has to scan the whole memory address space for valuable 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. Note, this still requires a known starting point for the search.

## 6 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 Subsection 6.2.

### 6.1 General Design

Our design of DAGGER is depicted in Figure 3. 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. **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.

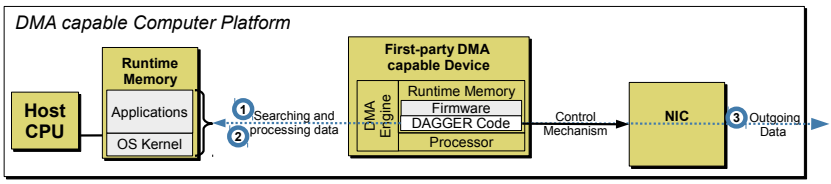


**Fig. 3.** General Design: DAGGER is executed on a DMA capable device so that it can (1) search and (2) process data from host runtime memory. It (3) controls a communication path to exfiltrate information.

### 6.2 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 paragraphs.

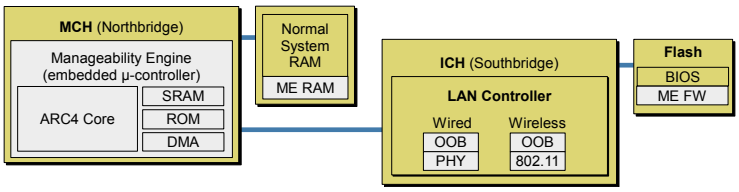


**Fig. 4.** Intel's Manageability Engine Environment

*Embedded µ-Controller.* The core of Intel's ME is an embedded µ-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 [5, 28], and a processor as depicted in Figure 4. The embedded processor of the ME is an ARCtangent-A4 (ARC4). The isolated execution 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.

*ME Firmware.* Applications executed on the embedded µ-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 [21]. 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 [19], *Alert Standard Format* [28, p.46], *Intel Quiet System Technology* for temperature and fan control [28, p.46], and *Integrated Trusted Platform Module* [21, p.109]. ME firmware can communicate with the host via a PCI device interface called

*ME Interface* (MEI) [21, p.71]. The MEI can provide the version of the executed ME firmware, for example.

*Separate Memory.* During the initial platform power-on procedure the ME firmware image is loaded into ME RAM. The firmware itself runs on the $\mu$-controller internal ARC4 processor and it also uses some system RAM as depicted in Figure 4 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 [21].

*Out-of-Band Network Channel.* 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. This kind of traffic is identified by TCP port numbers.

### 6.3 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) [25, 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 Section 5. 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 in [28] 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`. The authors of [28] assumed to hook a timer interrupt handler. But actually they hooked the ME firmware function `memcpy`. We hook `memset` since we determined that it is called more often.

**Linux.** Our Linux variant is based on a signature scan as depicted in Figure 5. We analyzed the available Linux source code to derive a signature of our target, the physical address of the keyboard buffer. The address of the buffer 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 a kernel identifer 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.

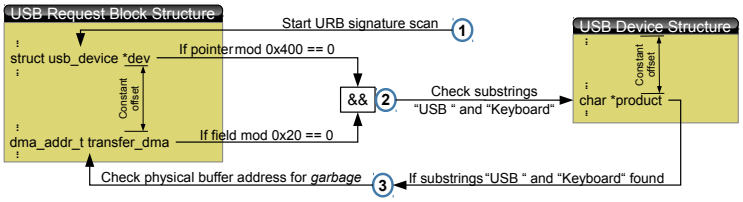


**Fig. 5.** USB Request Block Signature Scan (simplified): The scan (1) begins to search for a pointer to the USB device structure. A candidate for such a pointer is aligned to a `0x400` boundary. The structure field `transfer_dma` must be aligned to a `0x20` boundary. If both conditions are true, the 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.

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 determinded 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 adddresses 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 [6, Chapter 15]) is mapped to its physical address by substracting 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.

**Windows.** 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 sample 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*[1], *Windows Debugger* (WinDbg) tools, and debug symbols provided by Microsoft[2] in form of `pdb` files.

To finally determine the location of the buffer in the `DeviceExtension` structure, our research starts quite early in the Windows boot process [25, Chapter 13]. We analyzed further internal Windows structures. To find a starting point for the search, we analyzed the *Kernel Processor Control Region* (KPCR [25, 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 figured out that `KiInitialPCR` is well suited to derive a path to the `DeviceExtension` structure as depicted in Figure 6.

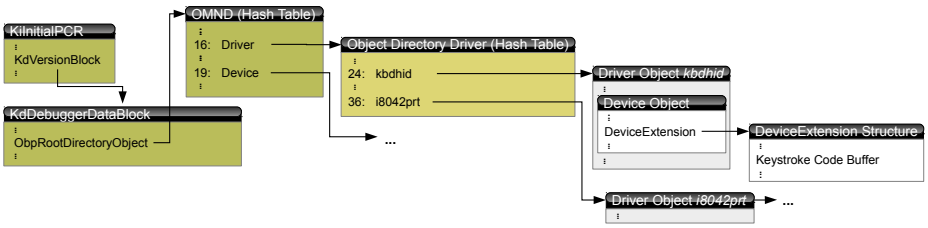


**Fig. 6.** Find `DeviceExtension` Structure (simplified): With `KiInitialPCR` as a starting point, DAGGER finds the OMND, that provides via hash tables a path to the driver object `kbdhid`. This object contains a pointer to a device object. The device object provides the `DeviceExtension` structure, which contains the keystroke code buffer.

`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 Figure 6.

The memory position of `KiInitialPCR` is determined by a function called `OslpLoadAllModules` of the `winload.exe` binary as depicted in Figure 7. This binary is loaded by the Windows boot manger `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 relative address. The disassembled code of `OslpLoadAllModules` is reminiscent of an *Address Space Layout Randomization* (ASLR) mechanism [25, p.757].

[1] See `http://www.hex-rays.com/products/ida/index.shtml`
[2] See `http://msdn.microsoft.com/en-us/windows/hardware/gg462988`

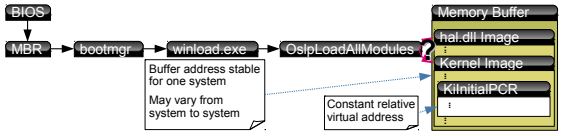


**Fig. 7.** Find `KiInitialPCR` (simplified): `OslpLoadAllModules` determines the exact position of the Windows kernel image and the HAL.

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 found the correct `KiInitialPCR`, DAGGER continues to look for the keyboard buffer using the search path described in Figure 6.

## 7 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. The DAGGER attack binary code has a size of approximately 33 KB for Linux and 31 KB for Windows.

**DMA Malware Fulfillment.** We designed and implemented our DAGGER prototypes according to the DMA malware definition described in Section 4. (C1) is clearly fulfilled since it 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 Figure 8 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. The search times summarized in Figure 9 are very short and the very aggressive memory stress test we performed does not represent the memory utilization of a normal computer system.

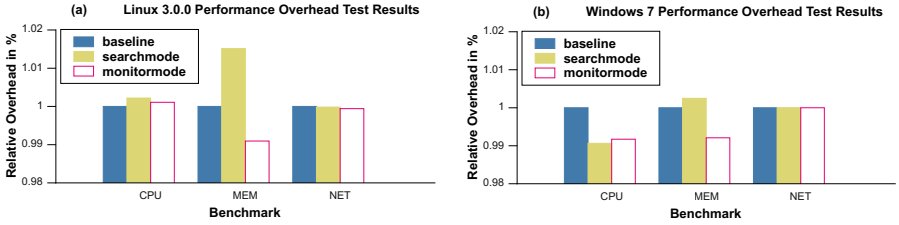


**Fig. 8.** Host Performance CPU, MEM, and NET Overhead Tests: We used *Time Stamp Counters* [6, p.186] to measure overhead time. We measured the time it takes to copy a 100 MB test file over the network (NET) and within RAM (MEM) as well as the time it needs to compute a SHA1 hashsum over this test 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 1000 packets per minute. This is equal to 500 keystroke and 500 key release events. We repeated each test 1000 times. A bar in the figure represents the mean of 1000 runs.

DAGGER has solely read-only operations to ensure stealthiness. The popular network sniffer *Wireshark*[3] 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. Nethertheless, we also run a software called *Mamutu*[4], 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 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.

**Effectiveness and Efficiency.** DAGGER is efficient, since it can permanently catch short living data from the keyboard buffer. To prove that DAGGER is also effective we tested DAGGER with different Windows and Linux versions as well as several keyboards (Logitech, Dell, FujitsuSiemens). The measured search

[3] See `http://www.wireshark.org/`
[4] See `http://www.emsisoft.com/en/software/mamutu/`

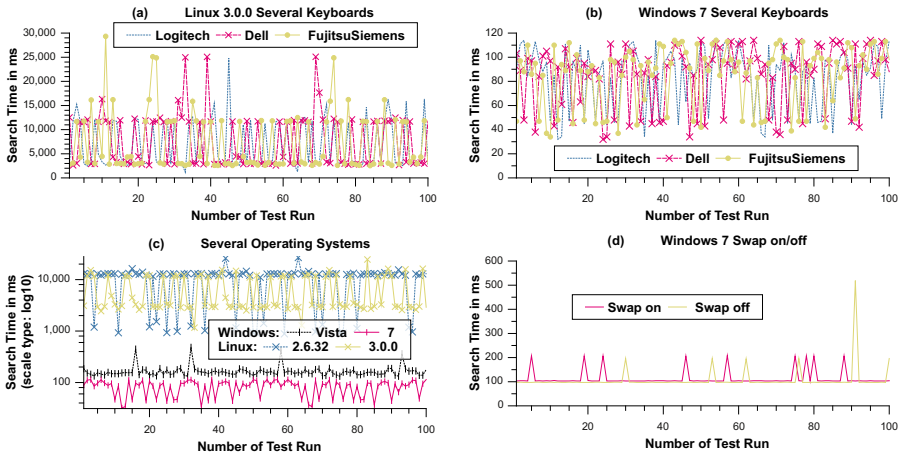


**Fig. 9.** Search Time Measurement Results: The test results with several keyboards under Linux reveal a best case for search times of around 1000 ms and a worst case of almost 30,000 ms as depicted in (a). The median for all keyboards is at 3281 ms. Useful for comparison: scanning the whole memory area determined for Linux (see Section 6.2) 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 keyboard is at 93 ms. Hence, the search strategy we implemented for Windows targets performs much better than the signature scan based strategy for Linux. The plot in (c) compares different target kernels. DAGGER performs slightly better for Windows 7 than for 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 1000 ms when attacking Linux 2.6.32. The results in (d) 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 search phase repeats.

times summarized in Figure 9 prove 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 2500 ms are due to target addresses near `0x33c00000`. Thus, we could skip almost 2500 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 (similar to the Windows attack) 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 Figure 9 (d).

**ME Firmware Condition.** To be really stealthy DAGGER ensures that the ME firmware is still up and running correctly. iAMT provides a webserver for remote platform management [21, p.215] that is still usable. The server responds correctly on the local platform on Linux and Windows. Firmware tools utilzing the MEI (see Section 6.2) 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 Opensource 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 when we deactivated the iAMT firmware in the BIOS. It appears that the ME environment cannot be disabled entirely via any BIOS options.

**I/OMMU.** To test an I/OMMU 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 experimentally supports I/OMMU configuration with additonal 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 and the code is still experimental. In the next section we discuss, among other things, further issues regarding the I/OMMU.

## 8 Countermeasures

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 *Trusted Computing Group* [32], does not prevent runtime attacks. Furthermore, it is unclear if all ROM components are accessible by the host.

**I/OMMU Issues.** In the case of DMA attacks an appropriate configuration of the I/OMMU (see Section 2.3) is proposed as a preventive countermeasure, for example in [11, p.48]. It is required that system software configures the I/OMMU. An incorrect configuration cannot be excluded [22, p.2].

It is assumed that the I/OMMU is secure. Unfortunately this is not always the case. The authors of [27] demonstrated that an I/OMMU configuration can be tricked with legacy PCI devices. In [35] is revealed that an I/OMMU can be attacked by modifying the number of DMA remapping engines provided by the BIOS (see Section 2.3). 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 [34, 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 in [35]. `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 [35]. Recently, the authors of [33] presented another attack that can be used to circumvent I/OMMU mechanisms, too. To prevent the attacks presented in [35, 33], a `SINIT` as well as a BIOS update must be applied. Another I/OMMU attack was presented in [34]. 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 to activate additional platform features, namely the *Trusted eXecution Technology* and the *Trusted Platform Module* [14]. That means, users that do not want to activate the TPM for example cannot count on the I/OMMU either. Note, the TPM is an opt-in device [14, 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 in [22] and [10]. The authors of [22] 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 in [10] 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 in [10]. 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 [24] and DeepWatch [5]. 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 [21, 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 implementation. This means, DAGGER proves that our assumption that an attacker already infiltrated the target system, e. g., via a zero-day exploit (see Section 3), can also hold even if such additional security mechanisms are in place.

On the one hand an appropriate configuration of the I/OMMU is a first step against DMA malware. On the other hand, without resolving the mentioned issues a successful deployment cannot be guaranteed.

## 9 Related Work

The discussion of related work is based on the classification system and its criteria C1 – C4 that we introduced in Section 4.[5]

Since 2004 several DMA attacks using additional hardware such as USB devices [23], special PCMCIA cards [2], and Firewire devices [8, 9, 3] were presented. According to C2 those approaches cannot be considered as DMA malware. According to our classification system of Section 4 the attacks presented in [8, 9, 3, 2] are classified in class 11 (1011). The attack presented in [23] is in class 1 (0001) since it reveals itself.

In [28] it was demonstrated that Intel's ME can be used to write to host memory. The authors of [28] described a vulnerability that allows to inject code into the ME environment. The code of [28] did not implement any malware behavior. It reveals itself by writing to a known hard coded host memory address. Hence, this approach is in class 5 (0101). Furthermore, it did not demonstrate how to read from host memory and how to use the OOB network channel.

On the contrary the attacks described in [30, 31, 11], and [7] fulfill all criteria for DMA malware. More precisely, they are classified in class 15 (1111). The authors of [30, 31] presented a stealthy secure shell that offers memory inspection using DMA. A combination of NIC and video card is used to hide the shell. The shell is installed by reflashing firmware remotely. Our DAGGER prototype does not require to infilrate code into two peripherals and it does not require to reflash firmware. The attack presented in [11] exploits a vulnerability in the firmware of a NIC during runtime. The compromised NIC is used to attack the host system by adding a backdoor. The authors of [11] described how the host could access the NIC internal memory. This offers a possibility to detect the DMA malware using code executed on the host CPU. As far as we know no anti-virus like software makes use of this. It should be mentioned that the host access to the NIC internal memory is not a common feature. Normally, the runtime memory of the Intel ME environment used for our DAGGER implementation is not accessible by the host. The work of [7] is quite similar to [11]. Both attacks use the same NIC.

[5] Note, all classifications were 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.

The malware described in [7] aims to implement rootkit capabilities. Their work is still in progress.

## 10 Conclusion

In this work we studied DMA malware, i.e., malware hidden on dedicated hardware. Such malware can circumvent protection mechanisms run on the host CPU by directly accessing 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.

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

Currently, the host has no reliable means to protect itself. Throughout this work 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 not sufficiently support the I/OMMU. Hence, currently, DMA malware can easily attack OSes such as Windows. A general and reliable approach for scanning the dedicated devices for malware does not exist. Future work is needed to develop a reliable and more general DMA malware detection mechanism. Until such a solution is developed, only dedicated hardware that is fully 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 detector gets loaded first.

We conclude that dedicated hardware with a separate processor, runtime memory, and a DMA engine are a serious threat for the host platform. DMA malware executed on such devices is quite effective and efficient. DMA malware clearly demonstrates that additional protection mechanisms are needed to ensure a platform's confidentiality, integrity, and especially its trustworthiness.

**Acknowledgments.** The authors would like to thank Dmitry Nedospasov, Jean-Pierre Seifert and especially Collin Mulliner for useful discussions and their inevitable help to complete this paper. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

## References

1. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel

Virtualization Technology for Directed I/O. *Intel Technology Journal 10*, 3 (Aug. 2006), 179 –192.
2. Aumaitre, D., and Devine, C. Subverting Windows 7 x64 Kernel with DMA attacks. Sogeti ESEC Lab: `http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbamsterdam-dmaattacks.pdf`, July 2010.
3. Boileau, A. Hit by a Bus: Physical Access Attacks with Firewire. Security-Assessment.com: `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`, Oct. 2006. Ruxcon 2006.
4. Budruk, R., Shanley, T., and Anderson, D. *PCI Express System Architecture*. The PC System Architecture Series. Addison Wesley, Pearson Education, July 2010. MindShare, Inc.
5. Bulygin, Y. Chipset based Approach to detect Virtualization Malware. TuCancUnix: `http://www.tucancunix.net/ceh/bhusa/BHUSA08/speakers/Bulygin_Detection_of_Rootkits/bh-us-08-bulygin_Chip_Based_Approach_to_Detect_Rootkits.pdf`, 2008.
6. Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.
7. Delugré, G. Closer to metal: Reverse engineering the Broadcom NetExtreme's firmware. Sogeti ESEC Lab: `http://esec-lab.sogeti.com/dotclear/public/publications/10-hack.lu-nicreverse_slides.pdf`, Oct. 2010.
8. Dornseif, M. 0wned by an iPod - hacking by Firewire. Laboratory for Dependable Distributed Systems University of Mannheim: `http://pi1.informatik.uni-mannheim.de/filepool/presentations/0wned-by-an-ipod-hacking-by-firewire.pdf`, Nov. 2004. PacSec 2004.
9. Dornseif, M., Becher, M., and Klein, C. N. FireWire – all your memory are belong to us. CanSecWest: `http://cansecwest.com/core05/2005-firewire-cansecwest.pdf`, May 2005.
10. Duflot, L., Perez, Y.-A., and Morin, B. What if you can't trust your network card? In *RAID* (2011), pp. 378–397.
11. Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card? French Network and Information Security Agency (FNISA): `http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf`, Mar. 2010.
12. Embleton, S., Sparks, S., and Zou, C. Smm rootkits: a new breed of os independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication networks* (New York, NY, USA, 2008), ACM, pp. 1–12.
13. Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium* (Feb. 2003).
14. Grawrock, D. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.
15. Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, May 2005. 3rd edition.
16. Hoglund, G., and Butler, J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
17. Intel Corporation. Intel I/O Controller Hub (ICH9) Family. Intel Corporation: `http://www.intel.com/content/dam/doc/datasheet/io-controller-hub-9-datasheet.pdf`, Aug. 2008.
18. Intel Corporation. 2nd Generation Intel Core vPro Processor Family. Intel Corporation: `http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf`, June 2011.

19. Intel Corporation. Access Accounts More Securely with Intel Identity Protection Technology. Intel Corporation: http://ipt.intel.com/Libraries/Documents/Intel_IdentityProtect_techbrief_v7.sflb.ashx, Feb. 2011.
20. King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J., and Lorch, J. R. SubVirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 314–327.
21. Kumar, A., Goel, P., and Saint-Hilaire, Y. *Active Platform Management Demystified*. Richard Bowles, 2009. Intel Press.
22. Li, Y., McCune, J. M., and Perrig, A. VIPER: Verifying the integrity of peripherals' firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Oct. 2011).
23. Maynor, D. DMA: Skeleton key of computing && selected soap box rants. CanSecWest: http://cansecwest.com/core05/DMA.ppt, May 2005.
24. Petroni, Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association.
25. Russinovich, M., and Solomon, D. A. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*, 5th ed. Microsoft Press, 2009.
26. Rutkowska, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. Internet Archive: http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html, Nov. 2004.
27. Sang, F., Lacombe, E., Nicomette, V., and Deswarte, Y. Exploiting an I/OMMU vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on* (oct. 2010), pp. 7–14.
28. Tereshkin, A., and Wojtczuk, R. Introducing Ring -3 Rootkits. black hat: http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf, July 2009.
29. Thompson, R. B., and Thompson, B. F. *PC Hardware in a Nutshell, 3rd Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.
30. Triulzi, A. Project Maux Mk.II. The Alchemist Owl: http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf, 2008.
31. Triulzi, A. The Jedi Packet Trick takes over the Deathstar. The Alchemist Owl: http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf, Mar. 2010.
32. Trusted Computing Group. TCG PC Client Specific Impementation Specification For Conventional BIOS. TCG: http://www.trustedcomputinggroup.org/files/temp/64505409-1D09-3519-AD5C611FAD3F799B/PCClientImplementationforBIOS.pdf, July 2005.
33. Wojtczuk, R., and Rutkowska, J. Attacking Intel TXT via SINIT code execution hijacking. ITL: http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, Nov. 2011.
34. Wojtczuk, R., and Rutkowska, J. Following the White Rabbit: Software attacks against Intel VT-d technology. ITL: http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf, Apr. 2011.
35. Wojtczuk, R., Rutkowska, J., and Tereshkin, A. Another Way to Circumvent Intel(R) Trusted Execution Technology. ITL: http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf, Dec. 2009.