Bypassing IOMMU Protection against I/O Attacks

Benoît Morgan, Eric Alata, Vincent Nicomette, Mohamed Kaâniche

Bypassing IOMMU protection against I/O Attacks

Benoît Morgan, Éric Alata, Vincent Nicomette, Mohamed Kaâniche
LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
7 Avenue du Colonel Roche
31000 Toulouse
Email: bmorgan@laas.fr, ealata@laas.fr, nicomett@laas.fr, kaaniche@laas.fr

***Abstract*—Attacks targeting computer systems become more and more complex and various. Some of them, so-called I/O attacks, are performed by malicious peripherals that make read or write accesses to DRAM memory or to memory embedded in other peripherals, through DMA (Direct Memory Access) requests. Some protection mechanisms to face these attacks exist and have been implemented for several years now in modern architectures. A typical example is the IOMMU proposed by Intel. However, such mechanisms are not necessarily properly configured and used by the firmware and the operating system. This experimental paper describes a design weakness that we discovered in the configuration of an IOMMU by the Intel IOMMU Linux driver and a possible exploitation scenario that would allow a malicious peripheral to bypass the underlying protection mechanism. The exploitation scenario is implemented with a PCI Express peripheral FPGA, based on Intel specifications and Linux source code analysis.**

I. Introduction

The evolution and sophistication of recent hardware architectures result from the continuously increasing requirements for software applications in terms of functionalites, processing power and speed as well as in terms of memory, storage capacities and communication possibilities. Today, software applications executed by microprocessors, like video games or electronic systems simulators need high performance levels, need to communicate over different networks and require services which cannot be provided anymore by the processor itself, like persistant external data storage or audio digital to analog encoding, etc. All these services are today provided by dedicated and independent hardware units which are external to the processor. These units, called peripherals, need to communicate with the processor for system configuration and data sharing.

Before the complexification of microarchitectures, early personal computers and their peripherals were mostly designed and built by the same compagny. The peripherals themselves used to be much less complex than today (microcode, firmwares, etc.). Processor manufacturers used to trust the peripherals. However, with the increasing demand for higher performance levels and hardware services, manufacturers improved input/output possibilities. They even specified normalized communication buses to allow tier manufacturers to complement bare architectures with complex peripherals. Then, to relieve the host processor from performing certain data copies and operation, DMA cycles have been added to these external buses, allowing as a consequence peripherals to perform read/write accesses to a set of other peripherals and RAM memory segments. These communication channels raise serious concerns about the security of such architectures, since they open some opportunities to attackers to compromise the system and hosted applications using some malicious peripherals. These attacks are so-called I/O attacks. To cope with such attacks, some hardware protection components, such as the IOMMU, from Intel, are included in modern computers. This experience report paper is aimed at analyzing the security of such protection component. In particular, it is shown, that even if this component has been introduced 10 years ago, some serious security concerns may be raised about its actual efficiency to prevent malicious I/O attacks due to some design weaknesses in its configuration. Briefly, the weakness is related to the fact that the configuration tables of the IOMMU are initialized in a DRAM region which is **not protected from DMA accesses**. A malicious peripheral may benefit from this weakness to modify these tables in memory just before the hardware setup of the IOMMU.

This paper is organized as follows. The following two sections describe fundamental components of the architecture, involved in the identified design weakness: Section II presents some basic technical background about PCI Express bus and communications that are used to perform DMA accesses while Section III presents the IOMMU component itself as well as some of its internals, that are necessary for the reader to understand the vulnerability as well as its potential exploitation. Section IV briefly presents some examples of I/O attacks discovered so far. Section V describes the vulnerability that we discovered in the configuration of the IOMMU and a scenario illustrating the possible exploitation of such vulnerability. Section VI proposes some countermeasures to cope with this vulnerability. Finally, Section VII concludes and discusses future work.

II. Technical background

This section presents basic technical background concepts related to PCI Express bus and communications, which are useful to understand the rest of the paper.

A. PCI Express bus

Several different bus specifications like Industry Standard Architecture (ISA) or Peripheral Component Interconnect (PCI) have been implemented to support the communications between the CPU and the peripherals. Today, the PCI Express bus is used in most personal computers or servers.

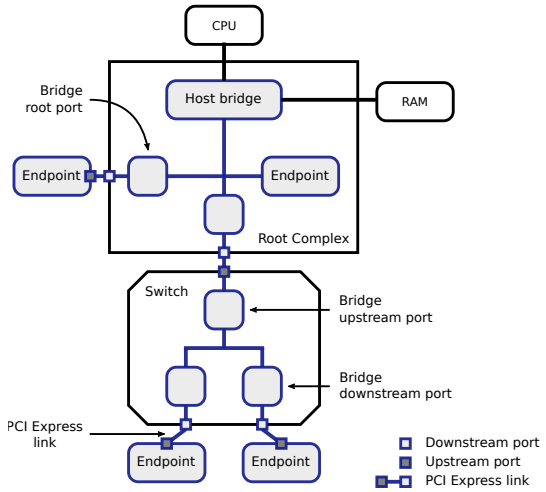

Fig. 1. Logical view of a typical PCI Express compliant architecture

PCI Express devices are connected with ports and links. A port connected to a child device is called a downstream port, while a port connected to a parent device is called an upstream port. There are three main types of PCI Express devices. The root complex, root of the bus hierarchy, is connected to the CPU thanks to the host bridge and to first level PCI Express children devices. These devices can be endpoints (so-called peripherals in the paper) and bridges (Figure 1). A bridge connects two different logical bus domains with an upstream and a downstream port. Finally, a switch is a collection of bridges. We note that the root complex can host peripherals and bridges. These bridges possess downstream ports only, which are called root ports.

B. Communications

Each PCI Express device is identified with a PCI logical bus, device and function identifier (id), noted `bus:dev.fun`. This identification is used to route PCI Express messages between devices.

A PCI Express message type is defined for every kind of transactions, like configuration or memory read and write for example. A transaction can be posted, which means that it does not expect a reply from the requested device, also it can be non-posted where a reply, called the completion, is needed. For example a memory write is posted because the sender doesn't expect any response. At the opposite, a memory read is non-posted and leads to the transmission of a memory read completion by the receiver.

The receiver of a message is either identified by its identifier or by an address. This address corresponds to a location in the main memory or to a register of another device if the registers of this device are memory mapped. Regarding a memory read message, it contains a destination address and a device requester id. The destination of the corresponding memory read completion is the associated requester id. PCI Express messages are therefore routed by address or id. To route the messages correctly, bridges are configured by the host to know which ids and memory ranges are responsible for downstream (Figure 2).

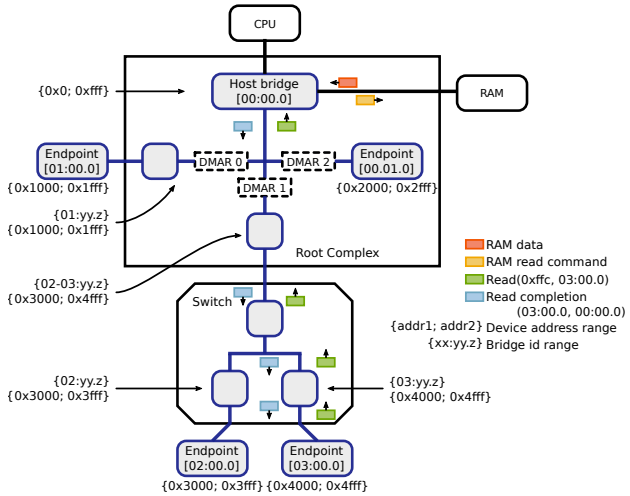

Fig. 2. Memory read message routing

At boot time, the devices are not configured (they do not know their own identifier). Indeed, the manufacturer can not know the bus on which the device will be plugged. However, when the BIOS "Basic Input Output Systems" looks for all available devices (using the simple assembler instruction `mov` from the PCI Express space mapped in memory), each memory access is processed by the host bridge. The host bridge then translates this access into a PCI Express configuration request which is routed to the corresponding bus. In particular, this request contains the identifier of the contacted device. So, if this device is available in the system, it receives this request and then knows its identifier. This step is important to allow a device to communicate.

PCI Express peripherals are able, through DMA requests, to have access to other peripheral memories and to the main RAM, even in the kernel memory regions, without any control of the processor. This raises a major concern from the security point of view if the peripheral behavior can be controlled by an attacker. To mitigate this threat, Intel has developed a hardware component to filter PCI Express messages. This component, called IOMMU, is presented in the next section.

III. IOMMU Internals – DMA Remapping

IOMMUs are designed to virtualize the memory space of the peripherals. An architecture can contain several IOMMUs, each one being dedicated to a subset of installed devices (Figure 2). IOMMUs translate and filter requests according to the protection domain assigned to the emitter device. A protection domain is simply a set of translation policies. The process is divided in two phases. The first one identifies the protection domain assigned to the emitter device. This phase, called device to domain mapping, is conceptually similar to an address translation but instead, it associates PCI identifiers to address translation domains. In the second phase, called address translation, the addresses used by peripherals

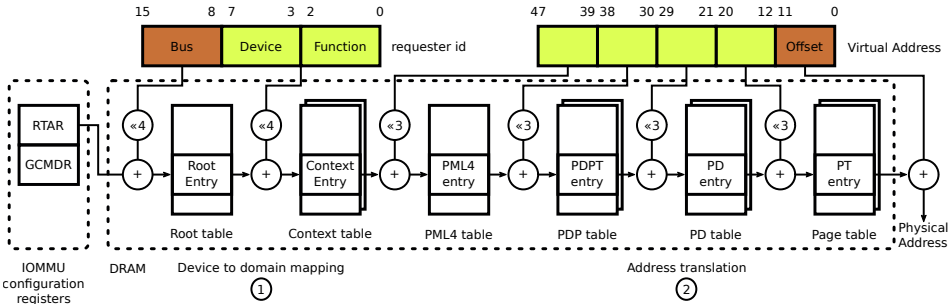

Fig. 3. DMA Remapping with 48 bits virtual addresses in 4kB pages

memory accesses are translated by the DMA Remapping units (DMAR), before crossing the host bridge (Figure 2). This translation is similar to the one made by the cores Memory Management Unit (MMU). Access controls are applicable in the two translation phases.

Each DMAR unit can be configured separately. The behavior of a DMAR is defined by a configuration page and a tree structure (Figure 3) configured at bootime in the DRAM. This configuration page contains the main control register called Global Command Register (GCMDR) which is responsible to activate the translation mechanism thanks to the Translation Enable bit (TE) [1, 10]. It also contains a pointer to the root of the structures (i.e. the root entry of the root table in the figure 3), named the Root Table Address Register (RTAR). The location of the configuration page of a DMAR is identified thanks to a dedicated register in the memory controller. The identifier of a PCI Express message sender is used to index the first two tables of the tree structure (root table and context table) in the first phase. The resolved address is then used to pinpoint the structures for the second phase. These last structures are indexed with the destination address of the PCI Express message. The result is the physical address of the translation. During the handling of these tables, a dedicated bit can indicate if the access is granted or forbidden. Finally, we note that the second translation phase can be deactivated (pass through mode) thanks to the translation type field of a context entry.

In this section we have introduced the theoretical functioning of DMAR. Unfortunately, like other hardware or software systems, these mechanisms may be inefficient if the implementation and the configuration are not correct. In the following, we briefly present some examples of such vulnerabilities before describing in detail the novel vulnerability we discovered as well as the exploitation example we implemented.

IV. I/O VULNERABILITIES

Many I/O attacks have been presented in the literrature [2], [3], [4], [5], [6]. In this paper, we focus on DMA attacks. These attacks were described in several studies. In particular, [7] demonstrates that an outsider can compromise a remote network card to control the system, remotely. Fortunately, most of these vulnerabilities have been fixed with the integration of IOMMU in the systems.

As presented in the previous section, a well configured DMAR unit is theoretically able to enforce an expected access control policy on a DMA accessible memory. The control access policy is given by the DMAR domain and a domain is associated to a device through device to domain mapping. Consequently, in order to be efficient, the DMAR unit needs to identify precisely each DMA capable device.

However, in [8], the authors take benefit of the colocation of PCI Express and PCI bus to exploit a weakness in the filtering performed by the IOMMU. Indeed, the PCI to PCI Express bridge uses its own PCI id as the requester id when translating PCI read / write cycles to PCI Express messages, acting like a proxy. Therefore, all the devices behind the PCI Express to PCI bridge are sharing the identity of the bridge from the IOMMU point of view, and so the same domain.

To illustrate the problem, let an Ethernet card and an Ipod be two devices installed behind a PCI Express to PCI bridge. If the Ethernet device driver configures the DMAR unit to map a memory region for DMA read and write, this association will also apply for the Ipod. The Ipod is therefore also able to have access to the region mapped by the Ethernet device driver. Authors called this vulnerability source id sharing. They exploited it by injecting malicious Ethernet frames into an IP kernel stack with a malicious firewire controller plugged behind the same bridge of a legitimate network card. Finally, they succeeded in corrupting an operating system ARP cache injecting well chosen ARP reply packets.

Even if such attacks are today inefficient in recent architectures, we discovered that the configuration of the IOMMU by the IOMMU Linux driver in recent kernels presents weaknesses that could be exploited by an attacker. In the following, we present this vulnerability and the attack scenario we implemented.

V. BYPASSING IOMMU

This section describes the discovered weakness in the Linux kernel. First, the architecture used for the experiment and the attack assumptions are presented. Then, the weakness is presented along with the attack scenario. Finally, the experiment is detailed.

A. Intel implementation

In this section, we introduce the Intel hardware architecture considered in our study. Two main physical components can

be distinguished in modern Intel architectures: the Processor and the Platform Controller Hub (PCH). In our case, we used a standard desktop machine hosting an Intel Haswell i7-4770 processor associated to the Intel PCH C220 (Figure 4). Our processor contains the cores, but unlike the older pre Nehalem microarchitectures, it also hosts an integrated graphic processor and the deprecated northbridge (providing the memory controller, southbridge bus interface, etc.), called now the system agent.

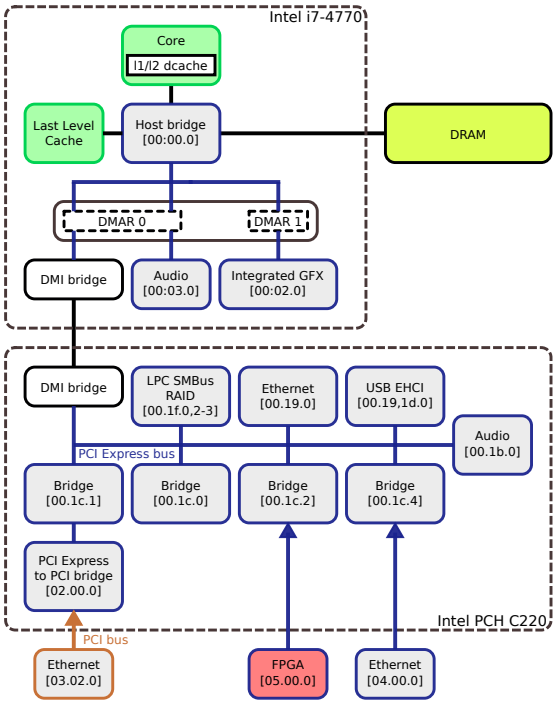

Fig. 4. Architecture of the system

The processor is connected through the proprietary DMI bus and bridges to the second main component of the architecture, ex southbridge, the PCH. C220 PCH hosts some I/O controllers like USB or Ethernet and audio. It also defines four root port bridges. Three of them are directly connected to expansion slots. The last one is connected to a PCI Express to PCI bridge itself connected to a PCI expansion slot. In PCI Express terms, we can see that the root complex is both part of the processor and PCH. Also, the host bridge is in the processor.

Two DMAR units are available. The first one is dedicated to the integrated video card embedded within the processor. The second one ensures the control and filtering of PCI Express messages from all other peripherals. In order to take benefit of these DMAR units, the operating system must be updated and correctly configured.

B. Attack assumptions

The attack assumptions are more easily understood using a scenario. Let Alice and Bob two employees of the same company. They both share the same machine, which is quite common for average to big sized enterprises. Alice wants to spy on Bob's activity. To be stealthy enough, she does not want to use classical software techniques which can be quite easily detectable in the case of a forensic analysis. She opts for a DMA attack which is stealthier. Alice and Bob run a GNU/Linux distribution with Intel DMAR activated but no Intel TXT late launch [9] (which is outside the scope of this paper). Alice has successfully identified a vulnerable PCI Express peripheral with the associated exploit allowing remote controlled arbitrary PCI Express reads and writes generation (like [7] as an example). Also, she selected a COTS DMA rootkit to get control on low level software stacks to do her spying.

C. Firmware and Linux IOMMU driver vulnerabilities

The vulnerability we discovered relies on a weakness of the BIOS used in the architecture and a weakness in the Intel IOMMU Linux driver `drivers/iommu/intel-iommu.c`. The latter makes the IOMMU configuration vulnerable during the boot up process. In order to understand the details, it is necessary to review the boot sequence used on a Linux machine.

1) Device configuration: At startup, IOMMUs are deactivated. During the boot time configuration, the firmware scans the PCI configuration space to discover and initialize vital devices, like the main VGA controller, loading and executing its embedded firmware (VGA BIOS). Read and write accesses to the configuration space send PCI Express configuration messages to the targeted device. Devices know their PCI ids after the first scan, as explained in section II-B. Therefore, the peripherals are able to generate messages and access the DRAM at the early firmware execution stage, long before the execution of the bootloader and then of the operating system kernel. At the end of the execution of the BIOS, the control is given to the bootloader and the kernel.

2) IOMMU configuration: The kernel is uncompressed and loaded by the bootloader, GNU GRUB in our experiment. Linux kernel modules and drivers are then loaded and initialized. DMAR is configured by one of these drivers. The Intel IOMMU driver creates the translation structures and saves them in the main DRAM. It builds the address translation domains and device to domain mapping before copying the root table pointer into the associated register. Finally, the driver activates DMAR by setting the TE bit of the GCMD Register. Let us note that these structures are stored in memory in **areas not protected by any security mechanism**.

3) Cache policy: The vulnerability we discovered is exploitable also because Linux flushes cache lines (L1 / L2 and Last Level Cache) after every table entry modification, to ensure the integrity of the structure in memory.

4) Physical memory space: We realized that as long as the machine hardware configuration is not modified, the physical addresses of the DMAR root table are not changed. This property can simplify the exploitation presented in this paper, preventing the attacker from searching structures to modify in the memory space.

D. Linux IOMMU driver exploitation

The goal of this attack is to bypass the DMAR address translation mechanism for a malicious peripheral and so enable read and write transactions to all DMA accessible memory ranges from this malicious peripheral. Our exploitation aims at bypassing the IOMMU memory protection without altering the integrity of the kernel itself. Fulfilling this constraint makes the attack more difficult to identify.

1) Exploration phase: As explained in Section V-B, the exploitation must begin with an exploration phase, carried out on a architecture similar to the targeted system architecture.

Figure 4 illustrates the possible position of a malicious peripheral in the architecture. The exploited vulnerable peripheral (converted into a malicious peripheral by a remote attack for instance) is simulated by a FPGA, embedded on a PCI Express peripheral development board. The FPGA is connected to the `00:1c.2` PCH root port. Our accesses are controlled and translated by DMAR unit 0, associated to DMI bridge upstream. After the startup process, the FPGA is associated to the identifier `05:00.0`. The different access requests of the FPGA are processed through the sixth entry of the root table, then the first entry of the context table and, depending on the accessed address, the corresponding address translation tables (Figure 3).

Configuration structures of the DMAR unit 0 are setup by the Linux driver. This driver stores these structures in the DRAM (as we explained previously, these structures do not stay in L1 or L2 cache, they are regularly flushed in memory). An analysis of the driver indicates that the location of these structures doesn't change from one boot to another. They are always located at the same address, provided that the hardware configuration and the operating system are kept intact. In addition, the analyses of the system lead to the identification of free pages accessible from DMA requests.

This information is the pre-requisite to perform the attack presented in the next section.

2) Implementation: The easiest way for a malicious peripheral to access DRAM is to enable pass through mode in the context entry of the DMAR (the address translation part of Figure 3 must grant every access of the malicious peripheral). The following steps describe a way to grant these accesses at the startup of the system.

First, the configuration of the system, at startup, allows all peripherals to write to DRAM. Thus, the malicious peripheral can easily produce a malicious context table in a free memory page (step 0, Figure 5), with all entries to pass through. This step must be performed at the beginning of startup.

When Linux begins its execution, it first writes its own root table (step 1 and 2). The second step of the attack must overwrite the sixth entry of this root table (it correponds to the bus 5 of our FPGA) before the IOMMU activation through bit TE.

Since these addresses do not change as long as the hardware configuration of the machine does not change, we do not have to scan the memory space for similar patterns. This is important because there is very short delay between the time Linux root table is written and the time the IOMMU is activated and it would probably be difficult to scan the memory to look for specific patterns in this short delay.

To maximize the chances to actually overwrite the root table entry during this short time interval, the malicious peripheral floods the bus with PCI Express write requests to the sixth entry of the root table (step ∞). Finally, Linux activates the IOMMU (steps 3) **with compromised DMAR structures**. From now, the malicious peripheral can perfom all DMA accesses it wants.

E. FPGA device implementation

We based the development of our malicious peripheral on Milkymist System On Chip [10], which is originally a video DJing open hardware project. Thanks to the hardware flexibility of FPGAs and the thoughtful modularity of hardware and software developed for Milkymist, we removed unneeded functionalities and added a PCI Express end point stack with minimum effort. Milkymist is made for a custom board hosting a Xilinx virtex 4 FPGA[11]. Since we needed a PCI Express connector and gigabit transceivers to develop the PCI Express peripheral, we have chosen a Xilinx ML605.

Our SOC contains the original Milkymist Lattice Mico32 microprocessor (LM32), Onchip ROM, Ethernet MAC, bus bridges, caches and controllers (Figure 6). In addition, we developed the malicious PCI Express peripheral, PCIE-EP. This core brings host memory access to the SOC through PCI Express memory messages. With PCIE-EP, the LM32 is able to program memory reads, memory writes (with a high rate mode). Note that this SOC is flexible and can be adapted for other purposes, e.g., for the implementation of integrity tests in the context of a hardware assisted trusted architecture as presented in [12].

F. Results and demonstration

In order to demonstrate that the exploitation of this vulnerability allows a potential attacker to further take control of the host, we considered an example of a kernel rootkit that we injected in kernel memory through DMA requests performed by our malicious FPGA device, once it has successfully modified the IOMMU configuration so that it can make read/write accesses to kernel memory. Our rootkit is a binary code which is injected in kernel memory and modifies the behavior of the `setuid` kernel function. Each time this function is called, the `euid` (effective uid) of the calling task is systematically set to

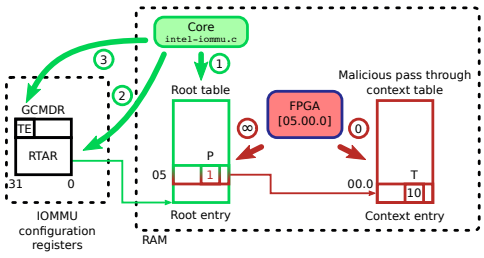

Fig. 5. Linux IOMMU driver attack implementation

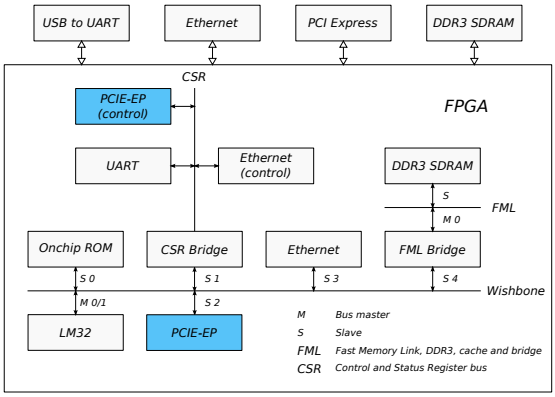

Fig. 6. Malicious device FPGA system on chip

0. We developed a small C code to call `setuid` function, that we executed by a non-root user, both in the presence and in the absence of our exploitation of the IOMMU configuration. The short video at[1] shows the rootkit installation and use.

VI. Secured boot process with IOMMU

Efficient protection against this attack can be implemented either by specifically and precisely configuring the hardware of each computer, or by redesigning the Intel architecture. However, setting such specific protections is quite difficult for the end user which is not a hardware and operating system expert. For instance, Intel TXT is one of these solutions but its installation and deployment is quite tricky. As a consequence, up to our knowledge, Intel TXT is not a solution that is largely deployed so far. It is therefore necessary to find solutions that are efficient without being too restrictive.

From our point of view, even if the operating systems have difficulties in taking into account all the details of all hardware platforms, they should use some existing features to harden the security of the system. We can mention especially the Bus Master Enable bit (BME) which can prevent the peripherals from sending messages. If this bit is set in the configuration of the different PCI Express bridges, the peripherals that are connected to these bridges cannot send any DMA requests by themselves anymore and thus cannot perform the exploitation we described in this paper. This bit is correctly set at the startup of the system. However, the firmware disables this protection before giving the control to the bootloader. This behavior is really surprising and we are currently investigating to understand its justification.

We also note that our platform brings additional security features: some DMA protected configurable memory segments. The DMA Protected Range (DPR) specified by the processor and the Protected Memory Ranges (PMR) implemented in the IOMMU [13, Vol. 2, 2.5]. Linux does not use these memory segments placing IOMMU structures outside the protected ranges. Devices are consequently able to read and write the IOMMU configuration before its activation. These DMA protected memory segments are quite usual in modern architectures and should be systematically used to set up such hardware protection components, such as the IOMMU. Once again, we noticed that the Linux kernel does not use these protected memory segments.

But, despite these protections, the system remains vulnerable to DMA attacks while the firmware is being executed, in the first moment of the boot process. This weakness is due to the fact that the firmware does not filter DMA accesses. It can be exploited by a malicious peripheral to modify the code of either Linux or the firmware itself and so prevent the activation of the IOMMU. At this time, it is necessary to check the integrity of software components by using a technology like Intel TXT.

[1]http://homepages.laas.fr/nicomett/SSTIC2016/iommu-pwn-sstic.webm

VII. Conclusion

This paper presents an experiment demonstrating the possible exploitation of a vulnerability related to the configuration of the IOMMU by the Linux IOMMU kernel driver. This attack makes it possible for malicious peripherals to make read and write accesses in main memory and to bypass the protection mechanisms embedded in the IOMMU. We are currently studying other operating systems (such as Windows, BSD systems) in order to check whether this vulnerability is only related to Linux kernel or not. In the same way, we also plan to investigate the boot process when the Intel TXT is activated to check that this technology does not suffer from similar weaknesses.

References

[1] Intel Corporation. Intel Virtualization Technology for Directed I/O. http://www.intel.com.

[2] M. Dornseif, “Owned by an ipod: Firewire/1394 issues,” in *PacSec*, 2004.

[3] M. Becher, M. Dornseif, and C. N. Klein, “FireWire: all your memory are belong to us,” in *CanSecWest*, 2005.

[4] D. Maynor, “Dma: Skeleton key of computing and selected soap box rants,” in *CanSecWest*, 2005.

[5] D. Aumaitre and C. Devine, “Subverting windows 7 x64 kernel with dma attacks. sogeti esec lab (july 2010), http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbamsterdam-dmaattacks.pdf.”

[6] P. Stewin and L. Bystrov, “Understanding dma malware,” in *Lecture Notes in Computer Science*, 2012, vol. 7591, ch. Detection of Intrusions and Malware and Vulnerability Assessment, pp. 21–41.

[7] L. Duflot, Y.-A. Perez, and B. Morin, “What if you can’t trust your network card ?” in *14 International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011, pp. 378–397.

[8] F. Lone Sang, V. Nicomette, and Y. Deswarte, “Attaques dma peer-to-peer et contremesures,” in *Actes du 9e Symposium sur la Sécurité des Technologies de lInformation et des Communications (SSTIC 11)*, 2011.

[9] Intel Corporation. Intel Trusted Execution Technology (Intel TXT) Software Development Guide, Measured Launched Environment Developers Guide. http://www.intel.com.

[10] Sébastien Bourdeauducq. Milkymist System On Chip. , https://github.com/m-labs/milkymist,.

[11] ——. Milkymist one board. https://events.ccc.de/camp/2011/Fahrplan/attachments/1874_mm_ccc2011.pdf.

[12] B. Morgan, E. Alata, V. Nicomette, M. Kaaniche, and G. Averlant, “Design and implementation of a hardware assisted security architecture for software integrity monitoring,” in *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, Nov 2015, pp. 189–198.

[13] Intel Corporation. Mobile 4th Generation Intel Core TM Processor Family, Mobile Intel Pentium Processor Family, and Mobile Intel Celeron Processor Family. http://www.intel.com.