A Framework to Secure Peripherals at Runtime

Fengwei Zhang[1], Haining Wang[2], Kevin Leach[3], and Angelos Stavrou[1]

[1] George Mason University, Fairfax, VA, USA
[2] College of William and Mary, Williamsburg, VA, USA
[3] University of Virginia, Charlottesville, VA, USA

Abstract. Secure hardware forms the foundation of a secure system. However, securing hardware devices remains an open research problem. In this paper, we present IOCheck, a framework to enhance the security of I/O devices at runtime. It leverages System Management Mode (SMM) to quickly check the integrity of I/O configurations and firmware. IOCheck is agnostic to the operating system. We use random-polling and event-driven approaches to switch into SMM. We implement a prototype of IOCheck and conduct extensive experiments on physical machines. Our experimental results show that IOCheck takes 10 milliseconds to check the integrity of a network card and a video card. Also, IOCheck introduces a low overhead on Windows and Linux platforms. We show that IOCheck achieves a faster switching time than the Dynamic Root of Trust Measurement approach.

Keywords: Integrity, Firmware, I/O Configurations, SMM

1 Introduction

As hardware devices have become more complex, firmware functionality has expanded, exposing new vulnerabilities to attackers. The National Vulnerabilities Database (NVD [1]) shows that 183 firmware vulnerabilities have been found since 2011. The Common Vulnerabilities and Exposures (CVE) list from Mitre shows 537 entries that match the keyword 'firmware,' and 94 new firmware vulnerabilities were found in 2013 [2]. A recent study shows that 40,000 servers are remotely exploitable due to vulnerable management firmware [3]. Attackers can exploit these vulnerabilities in firmware [4] or tools for updating firmware [5].

After compromising the firmware of an I/O device (e.g., NIC card), attackers alter memory via DMA [4, 6, 7] or compromise surrounding I/O devices [8, 9]. Fortunately, the Input Output Memory Management Unit (IOMMU) mechanism can protect the host memory from DMA attacks. It maps each I/O device to a specific area in the host memory so that any invalid access fails. Intel Virtualization Technology for Directed I/O (VT-d) is one example of IOMMU. AMD also has its own I/O virtualization technology called AMD-Vi. However, IOMMU cannot always be trusted as a countermeasure against DMA attacks, as it relies on a flawless configuration to operate correctly [10]. In particular, researchers have demonstrated several attacks against IOMMU [11–13].

Static Root of Trust for Measurement (SRTM) [14] with help from the Trust Platform Module (TPM) [15] can check the integrity of the firmware and I/O configurations while booting. It uses a fixed or immutable piece of trusted code, called the Core Root of Trust for Measurement (CRTM), contained in the BIOS at the start of the entire booting chain, and every piece of code in the chain is measured by the predecessor code before it is executed, including firmware. However, SRTM only secures the booting process and cannot provide runtime integrity checking.

Trust Computing Group introduced Dynamic Root of Trust for Measurement (DRTM) [16]. To implement this technology, Intel developed Trusted eXecution Technology (TXT) [17], providing a trusted way to load and execute system software (e.g., OS or VMM). TXT uses a new CPU instruction, `SENTER`, to control the secure environment. Intel TXT does not make any assumptions about the system state, and it provides a dynamic root of trust for Late Launch. Thus, TXT can be used to check the runtime integrity of I/O configurations and firmware. AMD has a similar technology called Secure Virtual Machine, and it uses the `SKINIT` instruction to enter the secure environment. However, both TXT and SVM introduce a significant overhead on the late Launch Operation (e.g., the `SKINIT` instruction in [18]).

In this paper, we present IOCheck, a framework to enhance the security of I/O devices at runtime. It leverages System Management Mode (SMM), a CPU mode in the x86 architecture, to quickly check the integrity of I/O configurations and firmware. IOCheck identifies the target I/O devices on the motherboard and checks the integrity of their corresponding configurations and firmware. In contrast to existing firmware integrity checking systems [19, 20], our approach is based on SMM instead of Protected Mode (PM). While PM-based approaches assume the booting process is secure and the OS is trusted, our approach only assumes a secure BIOS boot to set up SMM, which is easily achieved via SRTM.

The superiority of SMM over PM is two-fold. First, we can reduce the Trusted Computing Base (TCB) of the analysis platform. Similar to Viper [20] and NAVIS [19], IOCheck is a runtime integrity checking system. Viper and NAVIS assume the OS is trusted and use software in PM to check the integrity, while IOCheck uses SMM without relying on the OS, resulting in a much smaller TCB. IOCheck is also immune to attacks against the OS, facilitating a stronger threat model than the checking systems running in the OS. Second, we achieve a much higher performance compared to the DRTM approaches [18] running in PM. DRTM does not rely on any system code; it can provide a dynamic root of trust for integrity checking. IOCheck can achieve the same security goal because SMM is a trusted and isolated execution environment. However, IOCheck is able to achieve a much higher performance over Intel TXT or AMD SVM approaches. Based upon experimental results, SMM switching time takes microseconds, while the switching operation of the DRTM approach [18] takes milliseconds.

We implement a prototype of our system using different methods to enter SMM. First, we develop a random polling-based integrity checking system that checks the integrity of I/O devices, which can mitigate transient attacks [21, 22].

To further defend against transient attacks, we also implement an event-driven system that checks the integrity of a network card's management firmware.

We conduct extensive experiments to evaluate IOCheck on both Microsoft Windows and Linux systems. The experimental results show that the SMM code takes about 10 milliseconds to check PCI configuration space and firmware of NIC and VGA. Through testing IOCheck with popular benchmarks, IOCheck incurs about a 2% overhead when we set the random polling instruction interval between [1,0xffffffff][4]. We also compare IOCheck with the DRTM approach; our results indicate that our system's switching time is three orders of magnitude faster than DRTM. Furthermore, the switching time of IOCheck is constant while the switching operation in DRTM depends on the size of the loaded secure code.

Contributions. This work makes the following contributions:

- We provide a framework that checks the integrity of I/O devices at runtime.
- IOCheck is OS-agnostic and is implemented in SMM.
- We implement a prototype that uses random-polling and event-driven approaches to mitigate transient attacks.
- We demonstrate the effectiveness of our system by checking the integrity of a popular network card and video card, and we show that our system introduces a low operating overhead on both Windows and Linux platforms.

2 Background

2.1 Computer Hardware Architecture

The Central Processing Unit (CPU) connects to the Northbridge via the Front-Side Bus. The Northbridge contains the Memory Management Unit (MMU) and IOMMU, collectively called the Memory Controller Hub (MCH). The Northbridge also connects to the memory, graphics card, and Southbridge. The Southbridge connects a variety of I/O devices including USB, SATA, and Super I/O, among others. The BIOS is also connected to the Southbridge. Figure 2 in Appendix shows the hardware architecture of a typical computer.

2.2 Firmware Rootkits

A firmware rootkit creates a persistent malware image in hardware devices such as network cards, disks, and the BIOS. The capabilities of firmware rootkits can be summarized thusly. First, firmware rootkits can modify the host memory via DMA if a system does not have an IOMMU or if it is incorrectly configured. Second, a compromised device can access sensitive data that passes through it [23]. For instance, a NIC rootkit can eavesdrop network packets containing passwords. Third, a hardware device with malicious firmware may be able to compromise surrounding devices via peer-to-peer communication. For example, a compromised NIC may access GPU memory [24]. Last but not least, an advanced firmware rootkit can even survive a firmware update [25].

[4] It takes about .5s to run 0xffffffff instructions. Table 2 explains this further.

2.3 System Management Mode and Coreboot

System Management Mode (SMM) is a CPU mode in the x86 architecture. It is similar to Real and Protected Modes. It provides an isolated execution environment for implementing system control functions such as power management. SMM is initialized by the BIOS. Before the system boots up, the BIOS loads the SMM code into System Management RAM (SMRAM), a special memory region that is inaccessible from other CPU modes. SMM is triggered by asserting the System Management Interrupt (SMI) pin on the motherboard. Both hardware and software are able to assert this pin, although the specific method depends on the chipset. After assertion, the system automatically saves its CPU states into SMRAM and then executes the SMI handler code. An `RSM` instruction is executed at the end of the SMI hander to switch back to Protected Mode.

Coreboot [26] aims to replace legacy BIOS in most computers. It performs some hardware initialization and then executes additional boot logic, called a payload. With the separation of hardware initialization and later boot logic, Coreboot provides flexibility to run custom bootloaders or a Unified Extensible Firmware Interface (UEFI). It switches to Protected Mode early in the booting process and is written mostly in C language. Google Chromebooks are manufactured and shipped with Coreboot.

3 Threat model and Assumptions

3.1 Threat Model

We consider two attack scenarios. First, we consider an attacker who gains control of a host through a software vulnerability and then attempts to remain resident in a stealthy manner. We assume such an attacker installs firmware rootkits (specifically, a backdoor [23]) after infecting the OS so that the malicious code remains even if the user reinstalls the OS.

In the second scenario, we assume the firmware itself can be remotely exploited due to vulnerabilities. For instance, Duflot et al. [4] demonstrate an attack that remotely compromises a Broadcom NIC with crafted UDP packets. Additionally, Bonkoski et al. [3] show a buffer overflow vulnerability in management firmware that affected thousands of servers.

3.2 Assumptions

An attacker is able to tamper with the firmware by exploiting zero-day vulnerabilities. Since IOCheck does not rely on the operating system, we assume the attacker has ring 0 privilege. Thus, attackers are granted more capabilities in our work than those OS-based systems [19, 20]. We assume the system is equipped with SRTM, in which CRTM is trusted so that it can perform a self-measurement of the BIOS. Once the SMM code is securely loaded into the SMRAM, we lock the SMRAM in the BIOS. We assume the SMM is secure after locking SMRAM, and we will discuss attacks against SMM in Section 7. Moreover, we assume the attacker does not have physical access to our system.

4 System Framework

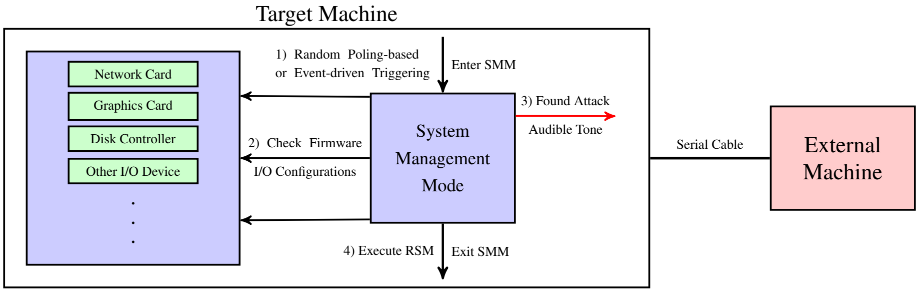

Fig. 1: Architecture of IOCheck

Figure 1 shows the architecture of IOCheck. The target machine connects to the external machine via a serial cable. In the target machine, the box on the left lists all of the I/O devices on a motherboard; the box on the right represents the System Management Mode code that checks the integrity of I/O configurations and firmware. The framework performs four steps for each check: 1) the target machine switches into SMM; 2) the SMI handler checks the integrity of target I/O devices; 3) if a potential attack has been found, the target machine plays an audible tone and SMM sends a message to the external machine via the serial cable; and 4) the target machine executes the `RSM` instruction to exit SMM. These steps are further described below.

4.1 Triggering an SMI

In general, there are software- and hardware-based methods to trigger an SMI. In software, we can write to an ACPI port to raise an SMI. For example, Intel chipsets use port `0x2b` as specified by the Southbridge datasheet. Our testbed with a VIA VT8237r Southbridge uses `0x52f` as the SMI trigger port [27]. In terms of hardware-based methods, there are many hardware devices that can be used to raise an SMI, including keyboards, network cards, and hardware timers.

The algorithm for triggering SMIs plays an important role in the system design. In general, there are polling-based and event-driven approaches used to generate SMIs. The polling-based approach polls the state of a target system at regular intervals. When we use this approach to check the integrity of a target system, it compares the newly retrieved state with a known pristine state to see if any malicious changes have occurred. However, polling at regular intervals in the system is susceptible to transient [21] or evasion attacks [22].

Transient attacks are a class of attacks that do not produce persistent changes within a victim's system. Polling-based systems suffer from transient attacks because they infer intrusions based upon the presence of an inconsistent state. Transient attacks can thus avoid detection by remove any evidence before a polling event and resuming malicious activity between polls. Mitigating these attacks requires either 1) minimizing the polling window so that there is less of a chance for the malware to clean its evidence, or 2) randomizing the polling window so that malware cannot learn a pattern for cleaning its evidence. We implement these methods in IOCheck via performance counters to trigger SMIs.

Moreover, we can use an event-driven triggering method to further mitigate transient attacks. Polling-based systems are likely to miss events between two checks that an event-driven approach would not. For instance, we can trigger SMIs when a region of memory changes, allowing us to monitor the state, including malicious changes.

4.2 Checking I/O Configurations and Firmware

Configurations of I/O Devices. Before the system boots up, the BIOS initializes all of the hardware devices on the motherboard and populates corresponding configuration spaces for each one. These devices rely on the configurations to operate correctly. Here we use the PCI configuration space and IOMMU configuration as examples.

PCI Configuration Space: Each PCI or PCIe controller has a configuration space. Device drivers read these configurations to determine what resources (e.g., memory-mapped location) have been assigned by the BIOS to the devices. Note that the PCI configurations should be static after the BIOS initialization. However, an attacker with ring 0 privilege can modify the PCI configuration space. For example, the attacker can relocate the device memory by changing the Base Address Register in the PCI configuration space. Additionally, PCI/PCIe devices that support Message Signaled Interrupts (MSI) contain registers in the PCI configuration space to configure MSI delivery. Wojtczuk and Rutkowska demonstrate that the attacker in the driver domain of a VM can generate malicious MSIs to compromise a Xen hypervisor [13]. Note that IOCheck assumes the PCI configuration remains the same after the BIOS initialization and does not consider "Plug-and-Play" PCI/PCIe devices.

IOMMU Configurations: IOMMU restricts memory access from I/O devices. For example, it can prevent a DMA attack from a compromised I/O device. IOMMU is comprised of a set of DMA Remapping Hardware Units (DRHU). They are responsible for translating addresses from I/O devices to physical addresses in the host memory. The DRHU first identifies a DMA request by BDF-ID (Bus, Device, Function number). Then, it uses BDF-ID to locate the page tables associated with the requested I/O controller. Finally, it translates the DMA Virtual Address (DVA) to a Host Physical Address (HPA), much like MMU translation. Although IOMMU gives us effective protection from DMA attacks, it relies on proper configurations to operate correctly. Several techniques have been demonstrated to bypass IOMMU [11, 13]. We can mitigate these attacks by checking the integrity of the critical configurations of IOMMU at runtime. Table 4 in Appendix shows the static configuration of IOMMU.

Firmware Integrity. We aim to check the firmware of I/O devices including the network card, graphics card, disk controller, keyboard, and mouse. We describe the process of checking a NIC, VGA, and the BIOS as examples.

Network Interface Controller: Modern network cards continue to increase in complexity. NICs usually include a separate on-chip processor and memory to support various functions. Typically, a NIC loads its firmware from Electric Erasable Programmable Read-Only Memory (EEPROM) to flash memory, and it then executes the code on the on-chip processor. IOCheck stores a hash value of the original firmware image and checks the integrity of the NIC's firmware at runtime. For some network cards [28], we can monitor the instruction pointer of the on-chip CPU through the NIC's debugging registers. This can restrict the instruction pointer to the code section of the memory region. If the instruction pointer points to a memory region that stores heap or stack data, then a code injection or control flow hijacking may have occurred.

Monitoring the integrity of the static code and instruction pointer can prevent an attacker from injecting malicious code into the firmware; however, it cannot detect advanced attacks, such as Return Oriented Programming attacks, since they technically do not inject any code. To detect these attacks, we can implement a shadow stack to protect the control flow integrity of the NIC firmware. Duflot et al. implemented a similar concept in NAVIS [19]. We will study the control flow integrity of the firmware in future work.

Video Graphics Adapter: The Video Graphics Adapter (VGA) normally requires device-specific initialization, and the motherboard BIOS does not have the knowledge of all possible vendor-specific initialization procedures. Fortunately, the PCI expansion ROM (i.e., option ROM) can be executed to initialize the VGA device. The VGA expansion ROM code is stored on the device, and this mechanism allows ROM to contain multiple images that support different processor architectures (e.g, x86, HP RISC). However, the ROM code on the device can be flashed with a customized image [29] or malicious code [30]. IOCheck uses SMM to ensure the integrity of the VGA option ROM at runtime.

Basic Input Output System: As mentioned before, SRTM can check the integrity of the BIOS at the booting time, which helps us to securely load the SMM code from the BIOS to the SMRAM. After the system boots up, attackers with ring 0 privilege might modify the BIOS using various tools (e.g., flashrom [31]). However, they are not able to access locked SMRAM. Thus, we can use the SMM code to check the runtime integrity of the BIOS. Although the modified BIOS with malicious code cannot be executed until the system resets and SRTM will detect this BIOS attack before booting, we can detect this attack earlier than SRTM, which provides runtime detection and serves as a complementary defense. Earlier detection of such attacks can also limit the damage they wreak against the system. Note that we assume CRTM in the BIOS is immutable and trusted, but attackers can modify any other BIOS code (e.g., ACPI tables). Otherwise, we cannot perform SRTM correctly.

4.3 Reporting an Alert and Exiting SMM

The last stage of IOCheck is to report any alerts to a human operator. We accomplish this task by playing an audible tone to notify a user that a potential attack may happen. To distinguish the type of attack, we use different tone frequency for a variety of I/O attacks. In addition, we use a serial cable to connect the target machine to the external machine. IOCheck assumes attackers with ring 0 privilege, which means they are able to modify hardware registers to block SMI assertions and launch a Denial-of-Service (DoS) attack against our system. We use the external machine to detect the DoS attack. For example, the random polling-based triggering in IOCheck must generate SMIs at least every maximum time interval, whereupon the external machine expects a message from SMM via the serial cable. If the external machine does not receive a log message in the interval, we conclude that a DoS attack has occurred. We also use a secret key to authenticate the log messages to avoid fake messages. Specifically, the target machine establishes a shared secret key with the external machine in the BIOS while booting. Since we trust the BIOS at startup, we can store the secret in the trusted SMRAM. Later, only the SMI handler can access it, which prevents attackers from spoofing messages.

Note that the reporting stage executes within SMM. Even if an attack disables the PC speaker or serial console in PM, we can enable it in SMM and guarantee that an audible tone and a serial message is delivered. After the reporting stage, the SMI handler simply executes the `RSM` instruction to exit from SMM.

5 System Implementation

We implement a prototype of IOCheck system using two physical machines. The target machine uses an ASUS M2V-MX_SE motherboard with an AMD K8 Northbridge and a VIA VT8237r Southbridge. It has a 2.2 GHz AMD LE-1250 CPU and 2GB Kingston DDR2 RAM. We use a PCIe-based Intel 82574L Gigabit Ethernet Controller and a PCI-based Jaton VIDEO-498PCI-DLP Nvidia GeForce 9500GT as the testing devices. To program SMM, we use open-source BIOS, Coreboot. Since IOCheck is OS-agnostic, we install Microsoft Windows 7 and CentOS 5.5 on the target machine. The external machine is a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS. It uses a 2.4GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM.

5.1 Triggering an SMI

We implement a random polling-based triggering algorithm to check integrity of I/O configurations and firmware by using performance counters to generate SMIs. The performance monitoring registers count hardware events such as instruction retirement, L1 cache miss, or branch misprediction. The x86 machines provide four of these counters from which we can select a specific hardware event to count [32]. To generate an SMI, we first configure one of the performance counters to store its maximum value. Next, we select a desired event (e.g., a retired instruction or cache miss) to count so that the next occurrence of that event will overflow the counter. Finally, we configure the Local Advanced Programmable Interrupt Controller (APIC) to deliver an SMI when an overflow occurs. Thus, we are able to trigger an SMI for the desired event. The performance counting event is configured by the `PerfEvtSel` register, and the performance counter is set by the `PerfCtr` register [32].

To randomly generate SMIs, we first generate a pseudo-random number, r, ranging from 1 to m, where m is a user-configurable maximum value. For example, a user could set m as 0xffff ($2^{16} - 1$), so the random number resides in the set [1,0xffff]. Next, we set the performance counter to its maximum value (0xffffffffffff) minus this random number ($2^{48} - 1 - r$). We also set the desired event in `PerfEvtSel` and start to count the event. Thus, an SMI will be raised after r occurrences of the desired event. We use a linear-congruential algorithm to generate the pseudo-random number, r, in SMM. We use the parameters of the linear-congruential algorithm from Numerical Recipes [33]. We use the TSC value as the initial seed and save the current random number in SMRAM as the next round's seed.

To further mitigate transient attacks, we consider event-driven-based triggering approaches. We implement an event-driven-based version of IOCheck for checking the integrity of a NIC's management firmware, and the detailed implementation is described as follows. When a management packet arrives at the PHY interface of the NIC, the manageability firmware starts to execute. We use Message Signalled Interrupts (MSI) to trigger an SMI when a manageability packet arrives at the network card. First, we configure the network card to deliver an MSI to the I/O APIC with the delivery mode specified as SMI. When the I/O APIC receives this interrupt, it automatically asserts the SMI pin, and an SMI is generated. Next, we use the SMM code to check the integrity of the management firmware. Note that the act of this triggering is generated via a hardware interrupt in the NIC, and the management firmware code is decoupled from this. Thus, we trigger an SMI for every manageability packet before the firmware has an opportunity to process it.

5.2 Checking I/O Configurations and Firmware

Network Interface Controller. We use a popular commercial network card, an Intel 82574L Gigabit PCIe Ethernet Controller, as our target I/O device. First, we check the PCIe configuration space of the network card. The NIC on our testbed is at bus 3, device 0, and function 0. To read the configuration space, we use standard PCI reads to dump the contents. We use a standard hash function MD5 [34] to hash these 256 bytes of the configuration and compare the hash value with the original one generated during booting.

Network management is an increasingly important requirement in today's networked computer environments, especially on servers. It routes manageability network traffic to a Management Controller (MC). One example of MC is the Baseboard Management Controller (BMC) in Intelligent Platform Management Interface (IPMI). The management firmware inevitably contains vulnerabilities that could be easily exploited by attackers. Bonkoski et al. [3] identified more than 400 thousand IPMI-enabled servers running on publicly accessible IP addresses that are remotely exploitable due to textbook vulnerabilities in the management firmware. The 82574L NIC [35] provides two different and mutually exclusive bus interfaces for manageability traffic. One is the Intel proprietary System Management Bus (SMBus) interface, and the other is the Network Controller - Sideband Interface (NC-SI). For each manageability interface, it has its own firmware code that implements the functions. Figure 3 in Appendix shows a high-level architectural block diagram of the 82574L NIC.

The management firmware of these two interfaces is stored in a Non-Volatile Memory (NVM). The NVM is I/O mapped memory in the NIC, and we use the EEPROM Read Register (EERD `0x14`) to read it. EERD is a 32-bit register used to cause the NIC to read individual words in the EEPROM. To read a word, we write a 1b to the Start Read field. The NIC reads the word from the EEPROM and places it in the Read Data field and then sets the Read Done field to 1b. We poll the Read Done bit to make sure that the data has been stored in the Read Data field. All of the configuration and status registers of 82574L NIC, including EERD, are memory-mapped when the system boots up. To access EERD, we use normal memory read-and-write operations. The memory address of EERD is INTEL_82574L_BASE plus EERD offset.

Video Graphics Adapter. Jaton VIDEO-498PCI-DLP GeForce 9500GT is a PCI-based video card. It is at bus 7, device 0, and function 0 on our testbed. Similar to the checking approach of NIC, we first check the PCI configuration space of the VGA device. Then, we check the integrity of the VGA expansion ROM. The VGA expansion ROM is memory-mapped, and the four-byte register at offset `0x30` in the PCI configuration space specifies the base address of the expansion ROM. Note that bit 0 in the register enables the accesses to the expansion ROM. PCI expansion ROMs may contain multiple images for different architectures. Each image must contain a ROM header and PCI data structure, which specify image information such as code type and size. Table 5 in Appendix shows the formats of ROM header and PCI data structure. Note that we only check the image for x86 architecture since our testbed is on Intel x86.

We first use the base address of expansion ROM to locate the header of the first image. Next, we read the pointer to PCI data structure at offset `0x18` to `0x19`. Then, we identify the code type at offset `0x14` in the PCI data structure. If this image is for Intel x86 architecture, we check the integrity of this image by comparing the hash values. Otherwise, we repeat the steps above for the next image.

5.3 Reporting an Alert and Exiting SMM

To play a tone, we program the Intel 8253 Programmable Interval Timer (PIT) in the SMI handler to generate tones. The 8253 PIT performs timing and counting functions, and it exists in all x86 machines. In modern machines, it is included as part of the motherboard's Southbridge. This timer has three counters (Counters 0, 1, and 2), and we use the third counter (Counter 2) to generate tones via the PC speaker. In addition, we can generate different kinds of tones by adjusting the output frequency. In the prototype of IOCheck, a continuous tone would be played by the PC speaker if a attack against NIC has been found. If an attack against VGA has been found, an intermittent tone would be played.

We use a serial cable to print status messages and debug corresponding I/O devices in SMM. The `printk` function in Coreboot prints the status messages to the serial port on the target machine. When the target machine executes the BIOS code during booting, the external machine sends a 16-byte random number to the target machine through the serial cable. Then, the BIOS will store the random number as a secret in the SMRAM. Later, the status messages are sent with the secret for authentication. We run a `minicom` instance on the external machine and verify if the secret is correct. If a status message is not received in an expected time window or the secret is wrong, we conclude that an attack has occurred.

6 Evaluation and Experimental Results

6.1 Code Size

In total, there are 310 lines of new C code in the SMI handler. The MD5 hash function has 140 lines of C code [34], and the rest of the code implements the firmware and PCI configuration space checking. After compiling the Coreboot, the binary size of the SMI handler is only 1,409 bytes, which introduces a minimal TCB to our system. The 1,409-byte code encompasses all functions and instructions required to check the integrity of the NIC and VGA firmware and their PCI configuration spaces. The code size will increase if we check more I/O devices. Additionally, other static code exists in Coreboot related to enabling SMM to run on a particular chipset. For example, a `printk` function is built into the SMM code to enable raw communication over a serial port.

6.2 Attack Detection

We conduct four attacks against our system on both Windows and Linux platforms. Two of them are I/O configuration attacks, which relocate the device memory by manipulating the PCI configuration space of NIC and VGA. The other two attacks modify the management firmware of the NIC and VGA option ROM. The Base Address Registers (BARs) in the PCI configuration space are used to map the device's register space. They reside from offset `0x10` to `0x27` in the PCI configuration space. For example, the memory location BAR0 specifies the base address of the internal NIC registers. An attacker can relocate these memory-mapped registers for malicious purposes by manipulating the BAR0 register. To conduct the experiments, we first enable IOCheck to check the PCI configuration space. Next, we modify the memory location specified by the BAR0 register on Windows and Linux platforms. We write a kernel module to modify the BAR0 register in Linux and use the RWEverything [36] tool to configure it in Windows. We also modify the management firmware of NIC and the VGA option ROM. The management firmware is stored as a Non-Volatile memory, and it is I/O mapped memory; the VGA option ROM is memory-mapped. These attacks are also conducted on both Windows and Linux platforms.

After we modify NIC's PCIe configuration or the firmware, IOCheck automatically plays a continuous tone to alert users and, the `minicom` instance on the external machine shows an attack against NIC has been found. After the modification of VGA's PCI configuration or option ROM, an intermittent tone is played by the PC speaker.

6.3 Breakdown of SMI Handler Runtime

Table 1: Breakdown of SMI Handler Runtime (Time: μs)

Operations	**Mean**	**STD**	**95% CI**
SMM switching	3.92	0.08	[3.27,3.32]
Check NIC's PCIe configuration	1169.39	2.01	[1168.81,1169.98]
Check NIC's firmware	1268.12	5.12	[1266.63,1269.60]
Check VGA's PCI configuration	1243.60	2.61	[1242.51,1244.66]
Check VGA's expansion ROM	4609.30	1.30	[4608.92,4609.68]
Send a message	2082.95	3.00	[2082.08,2083.82]
Configure the next SMI	1.22	0.06	[1.20,1.24]
SMM resume	4.58	0.10	[4.55,4,61]
Total	10,383.07		

To quantify how much time each individual step is required to run, we break down the SMI handler into eight operations. They are 1) switch into the SMM; 2) check the PCIe configuration of NIC; 3) check the firmware of NIC; 4) check the PCI configuration of VGA; 5) check the option ROM of VGA; 6) send a status message; 7) configure the next SMI; and 8) resume Protected Mode. For each operation, we measure the average time taken in SMM. We use the Time Stamp Counter (TSC) register to calculate the time. The TSC register stores the number of CPU cycles elapsed since powering on. First, we record the TSC values at the beginning and end of each operation, respectively. Next, we use the CPU frequency to divide the difference in the TSC register to calculate how much time this operation.

We repeat this experiment 40 times. Table 1 shows the average times taken for each operation. We can see that the SMM switching and resuming take only 4 and 5 microseconds, respectively. Checking 256 bytes of the PCIe/PCI configuration space register takes about 1 millisecond. The 82574L NIC has 70 bytes of SMBus Advanced Pass Through (APT) management firmware and 138 bytes of NC-SI management firmware. The size of x86 expansion ROM image is 1 KB in the testing VGA. Checking NIC's firmware takes about 1 millisecond, while checking VGA's option ROM takes about 5 milliseconds. Naturally, the size of the firmware affects the time of the checking operation. We send a status message (e.g., I/O devices are OK) in each run of the SMI handler, which is about 2 milliseconds. The time is takes to generate a random number and configure performance counters for the next SMI is only 1.22 microseconds. Thus, the total time spent in SMM is about 10 milliseconds. Additionally, we calculate the standard deviation and 95% confidence interval for the runtime of each operation.

6.4 System Overhead

Table 2: Random Polling Overhead Introduced on Microsoft Windows and Linux

	Random Polling Intervals		**Benchmark Runtime(s)**		**System Slowdown**	
	Instructions	Time (μs)	Windows	Linux	Windows	Linux
1	[1,0xffffffff]	(0,∼650,752]	0.285	0.393	0.014	0.011
2	[1,0xfffffff]	(0,∼40,672]	0.297	0.398	0.057	0.023
3	[1,0xffffff]	(0,∼2,542]	0.609	0.463	1.167	0.190
4	[1,0xfffff]	(0,∼158]	4.359	1.480	14.512	2.805
5	[1,0xffff]	(0,∼10]	91.984	18.382	∼326	∼46

To measure system overhead introduced by this approach, we use the SuperPI [37] program to benchmark our system on Windows and Linux. We first run the benchmark without IOCheck enabled. Then, we run it with different random-polling intervals. Table 2 shows the experimental results. The first column shows the random polling intervals used in the experiment. For example, (0,0xfffff] means a random number, r, is generated in that interval. We use retired instructions as the counting event in the performance counter. Thus, after running r sequential instructions, an SMI will be asserted. The second column also indicates the time elapsed. Since the CPU (AMD K8) on our testbed is 3-way superscalar [38], we assume an average number of instructions-per-cycle (IPC) is 3, and the equation for this transformation is $T = \frac{I}{(C*IPC)}$, where T is the real time, I is the number of instructions, and C is the clock speed on the CPU.

We can see from Table 2 that the overhead will increase if we reduce the random-polling interval, while small intervals have a higher probability of quickly detecting attacks. Intervals in rows 1 and 2 introduce less than 6% overhead, so intervals similar to or between them are suitable for normal users in practice. Other intervals in the table have large overhead making them unsuitable in practice. These results demonstrate the feasibility and scalability of our approach.

6.5 Comparison with the DRTM Approach

IOCheck provides a new framework for checking firmware and I/O devices at runtime. Compared to the well-known DRTM approach (e.g., Flicker [18]), SMM in IOCheck serves a similar role as the trusted execution environment in DRTM. However, IOCheck achieves a better performance in comparison. AMD uses the `SKINIT` instruction to perform DRTM, and Intel implements DRTM using a CPU instruction called `SENTER`. The SMM switching operation in IOCheck plays the same role as `SKINIT` or `SENTER` instructions in the DRTM approach. As stated in the Table II of Flicker [18], the time required to execute the `SKINIT` instruction depends on the size of the Secure Loader Block (SLB). It shows a linear growth in runtime as the size of the SLB increases. From Table 3, we can see that the SKINIT instruction takes about 12 milliseconds for 4KB of SLB. However, SMM switching only takes about 4 microseconds, which is about three orders of magnitude faster than the SKINIT instruction. Furthermore, SMM switching time is independent from the size of the SMI handler. This is because IOCheck does not need to measure the secure code every time before executing it, and we lock the secure code in SMRAM.

Table 3: Comparison between SMM-based and DRTM-based Approaches

	IOCheck	**Flicker [18]**
Operation	SMM switching	SKINIT instruction
Size of secure code	Any	4 KB
Time	3.92 μs	12 ms
Trust BIOS boot	Yes	No

Note that IOCheck trusts the BIOS boot while Flicker does not. IOCheck requires a secure BIOS boot to ensure the SMM code is securely loaded into SMRAM. However, the DRTM approach (e.g., Intel TXT) also requires that the SMM code is trusted. Wojtczuk and Rutkowska demonstrate several attacks [39, 12, 40] against Intel TXT by using SMM if the SMM-Transfer Monitor is not present. From this point of view, both systems must trust the SMM code.

7 Limitations and Discussions

IOCheck is a runtime firmware and configuration integrity checking framework. We also demonstrate the feasibility of this approach using a commercial network card. However, the current prototype of IOCheck is specific to the target system, which uses an Intel 82574L network card and JATON VIDEO-498PCI-DLP Nvidia video card. Human effort is required to expand the functionality (e.g., checking BMC or Disk Controller).

SMM uses isolated memory (SMRAM) for execution. The initial size of SMRAM is 64 KB, ranging from SMM_BASE to SMM_BASE + 0xFFFF. The default value of SMM_BASE is 0x30000, and Coreboot relocates it to 0xA0000. As the size of our SMI handler code is only 1,409 bytes, the small capacity of SMRAM may limit the scalability of IOCheck. However, the chipset in our testbed allows for an additional 4MB memory in a region called TSeg within SMRAM. Furthermore, SICE [41] demonstrates that SMM can support up to 4GB of isolated memory that can be used for memory-intensive operations such as virtualization.

Wojtczuk and Rutkowska [42] use cache poisoning to bypass the SMM lock by configuring the Memory Type Range Registers (MTRR) to force the CPU to execute code from the cache (which they injected) instead of SMRAM. Duflot also independently found the same vulnerability [43]. This vulnerability was fixed with Intel's addition of the System Management Range Register (SMRR). More recently, Butterworth et al. [25] used a buffer overflow vulnerability during the BIOS update process in SMM, although this was a bug in the particular BIOS version. Our SMM code in Coreboot does not have the same vulnerable code that facilitates this attack. To the best of our knowledge, there is no general attack that can bypass the SMM lock and compromise SMM.

The implementation of IOCheck contains 310 lines of C code. This part of the code may contain vulnerabilities that could be exploited by attackers. To reduce the possibility of vulnerable code, we sanitize the input of the SMI handler to reduce the attack surface. For instance, we do not accept any data input to the SMI handler except for the target firmware and configurations. We also carefully check the size of the input data to avoid overflow attacks [25].

SMM was not originally designed for security purposes. Researchers may argue that this makes it unsuitable for security operations. Additionally, some researchers feel that SMM is not essential to x86. However, there is no indication that Intel will remove SMM. Moreover, Intel introduced the SMM-Transfer Monitor [44] that virtualizes SMM code in order to defeat attacks [40] against TXT. In our case, SMM can be thought of as a mechanism to provide an isolated computing environment and hardware support to meet the system's requirements.

8 Related Work

To identify malware running in I/O devices, Li et al. propose VIPER [20], a software-based attestation method to verify the integrity of peripherals' firmware. VIPER runs a verifier program on the host machine, and it trusts the operating system. NAVIS [19] is an anomaly-detection system checking the memory accesses performed by the NIC's on-chip processor. It builds a memory layout profile of the NIC and raises an alert if any unexpected memory access is detected. The NAVIS program runs inside of the operating system and assumes the OS is trusted. Compared to VIPER and NAVIS, IOCheck is not running in the normal Protected Mode. It uses SMM to check the integrity of the firmware, which significantly reduces the TCB. In addition, IOCheck checks the configurations of I/O devices, which further protects them.

Compromised firmware normally performs DMA attacks against the main memory, and IOMMU (e.g., Intel VT-d or AMD-Vi) is an efficient defense. However, Sang et al. [11] identify an array of vulnerabilities on Intel VT-d. Wojtczuk et al. [12] use a bug in the `SINIT` module of the `SENTER` instruction to misconfigure VT-d, and then attackers are able to compromise the securely loaded hypervisor using a classic DMA attack so it can bypass Intel TXT. Although the main goal of this attack is to circumvent Intel TXT, we can learn that VT-d is easy to misconfigure and then an attacker can launch a DMA attack. Moreover, Stewin [10] explains several reasons that we cannot trust IOMMU as a countermeasure against DMA attacks. However, IOCheck is a generic framework that can check IOMMU configurations and provide further protection for I/O devices.

BARM [10] aims to detect and prevent DMA-based attacks. It is based on modeling the expected memory bus activity and comparing it to the actual activity. BARM relies on the OS and software applications to record all I/O bus activity in the form of I/O statistics, while IOCheck uses SMM without trusting any code in PM. IronHide [45] is a tool to analyze potential I/O attacks against

PCs. It can be used either as an offensive or defensive tool. On the offensive side, it can be used to sniff out the I/O buses, spoof the bus address used by other I/O controller, and log/inject keystrokes. On the defensive side, it injects faults over the I/O buses to simulate various I/O attacks and to identify various possible vulnerabilities. However, IronHide requires a specialized PCI-Express device, while IOCheck uses existing technology in chipsets.

Recently, SMM-based systems have been brewing in the security area [46–50]. HyperCheck [46] checks the integrity of hypervisors and uses a network card to transmit the registers and memory contents to a remote server for verification. Therefore, a compromised network card would be problematic in HyperCheck. HyperSentry [47] also uses SMM for hypervisor integrity checking, and it uses Intelligent Platform Management Interface (IPMI) to stealthily trigger an SMI. IPMI relies on BMC and its firmware to operate, while IOCheck can mitigate those attacks against firmware. Spectre [49] is a periodically polling-based system that introspects the host memory for malware detection. It uses SMM to periodically check the host memory for heap overflow, heap spray, and rootkit attacks. However, IOCheck aims to enhance the security of I/O devices, and we use random-polling and event-driven approaches to mitigate transient attacks against the periodic polling-based systems. In addition, researchers use SMM to implement stealthy rootkits [51], which requires an unlocked SMRAM to load the rootkit. As explained in [51], all post-2006 machines have locked SMRAM in the BIOS. IOCheck locks the SMM in Coreboot so that SMRAM is inaccessible after booting.

9 Conclusions

In this paper, we present IOCheck, a framework to enhance the security of I/O devices at runtime. It checks the firmware and configurations of I/O devices and does not require the trust on the OS. We implement a prototype of IOCheck using random-polling-based and event-driven approaches, and it is robust against transient attacks. We demonstrate the effectiveness of IOCheck by checking the integrity of Intel 82574L NIC and Jaton VIDEO-498PCI-DLP VGA. The experimental results show that IOCheck is able to successfully detect firmware and I/O configuration attacks. IOCheck only takes about 10 milliseconds to check the firmware and configurations, and it introduces a low overhead on both Microsoft Windows and Linux platforms. Furthermore, we compare IOCheck with the DRTM approach and show that the switching time of IOCheck is three orders of magnitude faster than that of the DRTM approach.

10 Acknowledgement

The authors would like to thank all of the reviewers for their valuable comments and suggestions. This work is supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024, National Science Foundation CRI Equipment Grant No. CNS-1205453, and ONR Grant

N00014-13-1-0088. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government, Air Force, or Navy.

References

1. National Institute of Standards, NIST: National Vulnerability Database. http://nvd.nist.gov Access time: 2014.03.04.
2. Mitre: Vulnerability list. http://cve.mitre.org/cve/cve.html
3. Bonkoski, A.J., Bielawski, R., Halderman, J.A.: Illuminating the Security Issues Surrounding Lights-out Server Management. In: Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT'13). (2013)
4. Duflot, L., Perez, Y.A.: Can You Still Trust Your Network Card? In: Proceedings of the 13th CanSecWest Conference (CanSecWest'10). (2010)
5. Chen, K.: Reversing and Exploiting an Apple Firmware Update. Black Hat (2009)
6. Stewin, P., Bystrov, I.: Understanding DMA Malware. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12). (2012)
7. Aumaitre, D., Devine, C.: Subverting Windows 7 x64 Kernel With DMA Attacks. In: HITBSecConf Amsterdam. (2010)
8. Triulzi, A.: Project Maux Mk.II. In: CanSecWest. (2008)
9. Sang, F., Nicomette, V., Deswarte, Y.: I/O Attacks in Intel PC-based Architectures and Countermeasures. In: SysSec Workshop (SysSec'11). (2011)
10. Stewin, P.: A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platforms Main Memory. In: Research in Attacks, Intrusions, and Defenses (RAID'13). (2013)
11. Sang, F., Lacombe, E., Nicomette, V., Deswarte, Y.: Exploiting an I/OMMU vulnerability. In: 5th International Conference on Malicious and Unwanted Software (MALWARE'10). (2010) 7–14
12. Wojtczuk, R., Rutkowska, J.: Another Way to Circumvent Intel Trusted Execution Technology. http://invisiblethingslab.com/resources/misc09/Another(2009)
13. Wojtczuk, R., Rutkowska, J.: Following the White Rabbit: Software Attacks against Intel VT-d (2011)
14. Trusted Computing Group: TCG PC Client Specific Implementation Specification for Conventional BIOS. `http://www.trustedcomputinggroup.org/files/resource_files/CB0B2BFA-1A4B-B294-D0C3B9075B5AFF17/TCG_PCClientImplementation_1-21_1_00.pdf` (February 2012)
15. Trusted Computing Group: TPM Main Specification Level 2 Version 1.2, Revision 116. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification` (2011)
16. Trusted Computing Group: TCG D-RTM Architecture Document Version 1.0.0. `http://www.trustedcomputinggroup.org/resources/drtm_architecture_specification` (June 2013)
17. Intel: Trusted Execution Technology. http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html
18. McCune, J., Parno, B., Perrig, A., Reiter, M., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems. (2008)
19. Duflot, L., Perez, Y.A., Morin, B.: What If You Can't Trust Your Network Card? In: Recent Advances in Intrusion Detection (RIAD'11). (2011)

20. Li, Y., McCune, J., Perrig, A.: VIPER: Verifying the Integrity of PERipherals' Firmware. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11). (2011)
21. Moon, H., Lee, H., Lee, J., Kim, L., Y., P., B., K.: Vigilare: Toward Snoop-based Kernel Integrity Monitor. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12). (2012)
22. Wang, J., Sun, K., Stavrou, A.: A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems. In: Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12). (2012)
23. Zaddach, J., Kurmus, A., Balzarotti, D., Blass, E.O., Francillon, A., Goodspeed, T., Gupta, M., Koltsidas, I.: Implementation and Implications of a Stealth Hard-Drive Backdoor. In: Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13). (2013)
24. Triulzi, A.: The Jedi Packet Trick Takes Over the Deathstar: Taking NIC Backdoors to the Next Level. In: The 12th Annual CanSecWest Conference. (2010)
25. Butterworth, J., Kallenberg, C., Kovah, X.: BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In: Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13). (2013)
26. Coreboot: Open-Source BIOS. http://www.coreboot.org/
27. VIA: VT8237R Southbridge. http://www.via.com.tw/
28. Broadcom Corporation: Broadcom NetXtreme Gigabit Ethernet Controller. http://www.broadcom.com/products/BCM5751
29. Salihun, D.: BIOS Disassembly Ninjutsu Uncovered. http://bioshacking.blogspot.com/2012/02/bios-disassembly-ninjutsu-uncovered-1st.html
30. Salihun, D.: Malicious Code Execution in PCI Expansion ROM. http://resources.infosecinstitute.com/pci-expansion-rom/ (June 2012)
31. Flashrom: Firmware flash utility. http://www.flashrom.org/
32. Advanced Micro Devices, Inc.: BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors
33. William, H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, New York (2007)
34. : MD5 Hash Functions. http://en.wikipedia.org/wiki/MD5
35. Intel: 82574 Gigabit Ethernet Controller Family: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82574l-gbe-controller-datasheet.html
36. Jeff: RWEverything Tool. http://rweverything.com/
37. SuperPI. http://www.superpi.net/
38. Advanced Micro Devices, Inc.: AMD K8 Architecture. http://commons.wikimedia.org/wiki/File:AMD_K8.PNG
39. Wojtczuk, R., Rutkowska, J.: Attacking Intel Trust Execution Technologies. http://invisiblethingslab.com/resources/bh09dc/Attacking(2009)
40. Wojtczuk, R., Rutkowska, J.: Attacking Intel TXT via SINIT Code Execution Hijacking. http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf (November 2011)
41. Azab, A.M., Ning, P., Zhang, X.: SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11). (2011)
42. Wojtczuk, R., Rutkowska, J.: Attacking SMM Memory via Intel CPU Cache Poisoning (2009)

43. Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM Reloaded. In: Proceedings of the 12th CanSecWest Conference (CanSecWest'09). (2009)
44. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual
45. Sang, F.L., Nicomette, V., Deswarte, Y.: A Tool to Analyze Potential I/O Attacks Against PCs. IEEE Security & Privacy (2013)
46. Zhang, F., Wang, J., Sun, K., Stavrou, A.: HyperCheck: A Hardware-assisted Integrity Monitor. In: IEEE Transactions on Dependable and Secure Computing. (2013)
47. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10). (2010)
48. Reina, A., Fattori, A., Pagani, A., Cavallaro, L., Bruschi, D.: When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC'12). (2012)
49. Zhang, F., Leach, K., Sun, K., Stavrou, A.: SPECTRE: A Dependable Introspection Framework via System Management Mode. In: Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13). (2013)
50. Zhang, Y., Pan, W., Wang, Q., Bai, K., Yu, M.: HypeBIOS: Enforcing VM Isolation with Minimized and Decomposed Cloud TCB. Technical report, Virginia Commonwealth University (2012)
51. Embleton, S., Sparks, S., 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 (SecureComm'08). (2008)
52. PCI-SIG: PCI Local Bus Specification Revision 3.0. http://www.pcisig.com/specifications/

Appendix

Table 4: IOMMU Configurations

Register/Table Name	**Description**
Root-entry table address	Defines the base address of the root-entry table (first-level table identified by bus number)
Domain mapping tables	Includes root-entry table and context-entry tables (second-level tables identified by device and function numbers)
Page tables	Defines memory regions and access permissions of I/O controllers (third-level tables)
DMA remapping ACPI table	Defines the number of DRHUs present in the system and I/O controllers associated with each of them

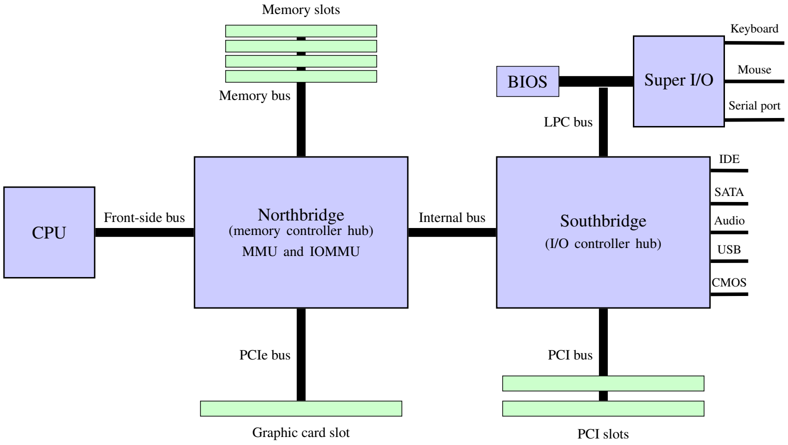

Fig. 2: Typical Hardware Layout of a Computer

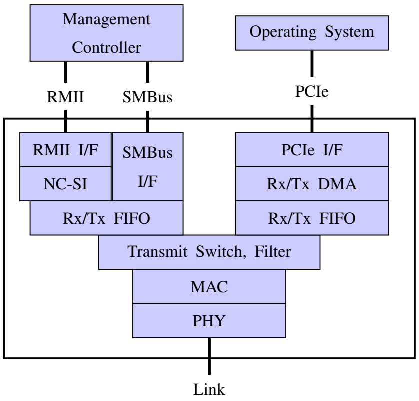

Fig. 3: Architecture Block Diagram of Intel 82574L [35]

Table 5: PCI Expansion ROM Format [52]

(a) PCI Expansion ROM Header Format for x86

Offset	**Length**	**Value**	**Description**
0h	1	55h	ROM signature, byte 1
1h	1	AAH	ROM signature, byte 2
2h	1	xx	Initialization size
3h	3	xx	Entry point for INIT function
6h-17h	12h	xx	Reserved
18h-19h	2	xx	Pointer to PCI data structure

(b) PCI Data Structure Format

Offset	**Length**	**Description**
0h	4	Signature, the string ”PCIR”
4h	2	Vendor identification
6h	2	Device identification
8h	2	Reserved
Ah	2	PCI data structure length
Ch	1	PCI data structure revision
Dh	3	Class code
10h	2	Image length
12h	2	Revision level of code/data
14h	1	Code type
15h	1	Indicator
16	2	Reserved