UEFI Hypervisors – Winning the Race to Bare Metal

Don Bailey
Hypervista Technologies
Herndon, VA 20170 USA
+001 703 582 1277
don@hypervista-tech.com
Black Hat Conference Proceedings 2008

Abstract. Improvements in Operating Systems security have created a race to "bare metal" between malware and protection software authors. Bare metal capabilities are defined as software or firmware applications that run outside the context of the OS and are therefore very powerful and difficult to detect. Hypervisors and SMM based rootkits are examples of bare metal capabilities. Because the digital battle ground is contracting, advantage goes to the one who can establish a presence there first.

The Unified Extensible Firmware Interface (UEFI), the emerging BIOS framework, implements very powerful pre-OS capabilities and therefore is the perfect framework to give bare metal capabilities the quickest possible foothold even **before** the OS bootloader is called. This paper discusses combining hypervisor and UEFI technologies to effect hypervisor control of a platform at the earliest opportunity, during the pre-boot phase.

Keywords: Hypervisor, Intel® VMX, Unified Extensible Firmware Interface (UEFI), Pre-OS environment, Hardware-based attack vector, UEFI Runtime device driver

1 Introduction

The efforts to lock down and harden OS kernels over the past five years have begun to pay dividends. Initiatives such as Microsoft's *Trustworthy Computing Initiative* [1] and the software security evangelizing of Michael Howard [2] and others has resulted in Operating Systems and applications that are prohibitively more difficult to exploit. Malware and therefore software protection authors have shifted the focus of their efforts to bare metal. Buffer overflow types of attacks are fading to the ash heap of exploitation history and the threat from hypervisor and SMM based rootkits is rising.

Concrete examples of the looming bare metal threats are Joanna Rutkowska's *Blue Pill* hypervisor project, which she introduced at Black Hat 2006 [3], and the SMM rootkit introduced at Black Hat 2008 by Shawn Embleton and Sherri Sparks of Clear Hat Consulting [4]. The implication for system security that *Blue Pill* and *SMM Rootkits* represent is clear. Capabilities that run outside the context of the OS enjoy near omnipotence on the platform and are very difficult to detect using common methods.

Virtualization is enjoying resurgence in popularity. Interest in this not so new technology is being driven in part by recent innovations by Intel® and AMD® who have provided support for virtualization in silicon. Intel® refers to their CPU based support for virtualization alternatively as *Vanderpool, VT-x, or VMX*. AMD® refers to theirs as *Pacifica or SVM.* Prior to the introduction of VMX and SVM, semi-efficient implementations of virtualization such as hypervisors used a technique known as para-virtualization. Para-virtualization requires that the OS source code be modified, which in the case of Linux is not a problem. In the absence of access to the OS source code, as in the case of Microsoft Windows®, dynamic binary translation of roughly seventeen Intel® Instruction Set Architecture (ISA) instructions is required. When you consider that these seventeen ISA instructions require trapping and modification dynamically, you can understand the use of the term "semi-efficient". The performance hit associated with dynamic binary translation is significant and can be seen in the "lagginess" of the mouse movement when running Windows in a VMWare® session, but the true impact of the performance degradation is generalized system wide. By providing support for virtualization in silicon, Intel® and AMD® have obviated the need for dynamic binary translation and para-virtualization, minimizing the performance hit associated with virtualization, and thereby stimulating renewed interest and "buzz" around hypervisors. Hypervisors are the quintessential bare metal construct.

UEFI is poised to replace the venerable BIOS. Legacy BIOS is an amazing accomplishment in software engineering. BIOS has lasted over 25 years and through that time the PC architecture has changed drastically. Nevertheless, BIOS engineers at many different companies and in many different countries have kept BIOS viable with patches and upgrades through the years. However, when the 512 byte boot block was demonstrated to be too small for the 64-bit Itanium instructions, the death nil of BIOS was rung. UEFI is the future of BIOS. Now that Microsoft ® is providing support for UEFI in their Windows Vista x64 SP1, the adoption of UEFI will likely accelerate.

The major impediment now to widespread adoption of UEFI is the paucity of motherboards that support it. This technology is just now emerging on the PC scene, but will radically change it once the motherboard OEMs begin supporting it in numbers. During our recent search for a UEFI compliant

PC motherboard, even Intel®, AMI®, and Phoenix® had difficulty identifying PC motherboards that support UEFI 2.x in June 2008. AMI® identified the Micro-Star International® P45 series of motherboards as being UEFI 2.0 compliant, and while this motherboard is available for purchase, MSI® won't have the UEFI 2.0 BIOS ready for release until August 2008. Phoenix® partner CalSoftLabs® identified an Intel® reference board by the name of *MONTEVINA* that supports UEFI 2.0. *MONTEVINA*, which is the code name of the Intel® fifth generation *Centrino* platform based on the *Penryn* 45mn processor, will be branded the *"Centrino 2 vPro"*. The *Centrino 2 vPro* won't be ready for release until mid-July 2008.

The most intriguing aspect of UEFI, at least in the context of this thesis, is its rich pre-OS capabilities. In many respects the UEFI Framework can be viewed as an OS in its own right. For example, the UEFI Framework provides mechanisms for prioritizing execution on the CPU, mechanisms for managing devices (device driver loading, device I/O protocols), and mechanisms for managing memory. In fact, the UEFI Framework provides the capability to establish network connections, provide rich graphics support, load drivers, and run applications ***before*** the OS bootloader is called. The pre-OS capabilities of UEFI can be used to launch a hypervisor, establishing bare metal control of the platform at the earliest possible moment.

2 Bare Metal Hypervisors

Hypervisors come in two different types; Type-1 (bare metal) hypervisors and Type-2 (hosted) hypervisors.

2.1 Types of Hypervisors Defined

Type-2 (*hosted)* hypervisors reside on top of or along side the OS; it's an application you launch from the OS. Examples of type-2 hosted hypervisors are VMWare® Workstation, QEMU, Microsoft® Virtual PC, and others. For the purposes of this paper, we are not interested in type-2 hypervisors but it's important to understand the difference between a bare metal hypervisor and a hosted hypervisor.

Type-1 (*bare metal)* hypervisors run directly on the platform hardware and acts as an OS control program, i.e., the OS loads on top of the hypervisor and runs as a **guest**. Type-1 bare metal hypervisors can exercise purview over every aspect of the platform. The bare metal hypervisor can control the OS, memory, and all applications running on the platform. It is the omnipresent nature of bare metal hypervisors that make them an excellent payload for UEFI pre-OS capabilities.

2.2 x86 Architectural Challenges to Virtualization

The x86 architecture is particularly difficult to virtualize. In a classic paper (Popek and Goldberg 1974), which was inspired by Goldberg's doctorate dissertation (1972), Popek and Goldberg formally derived the conditions under which an ISA can efficiently support a hypervisor [5]. According to the Popek and Goldberg paper there are seventeen x86 ISA instructions that can not be efficiently virtualized and therefore must be dealt with in architecting a x86 hypervisor.

Per Popek and Goldberg, in order to support a Type-1 bare metal hypervisor, a processor must meet three virtualization requirements:

1. The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. A processor must not use an additional bit in an instruction word or in the address portion of an instruction when in privileged mode.

2. There must be a method such as a protection system or an address translation system to protect the real system and any other guest OS from the active guest OS.

3. There must be a way to automatically signal the hypervisor when a guest OS attempts to execute one of the seventeen sensitive instructions. It must also be possible for the hypervisor to simulate the effect of the instruction. Sensitive instructions include:

3A. Instructions that attempt to change or reference the mode of the guest OS or the state of the machine.

3B. Instructions that read or change sensitive registers and/or memory locations such as the clock register and interrupt registers.

3C. Instructions that reference the storage protection system, memory system, or address relocation system. This class of instruction includes instructions that would allow the guest OS to access any location not in its virtual memory.

3D. All I/O instructions.

The seventeen sensitive instructions mentioned all violate one of the listed requirement 3 (3A - 3D) above.

Several of the seventeen sensitive instructions violate requirement 3B (sensitive register instructions), namely: SGDT (Store Global Descriptor

Table), SIDT (Store Interrupt Descriptor Table), and SLDT (Store Local Descriptor Table). These instructions are normally only used by the OS but are NOT privileged in the Intel® Architecture. Since Intel® processors only have one LDTR, IDTR and GDTR, a problem arises when multiple operating systems try to use the same registers.

The next sensitive instruction is the SMSW (Store Machine Status Word) instruction. SMSW stores the machine status word (bits 0 - 15 of CR0) into a general purpose register or memory location. Bits 6 - 15 of CR0 are reserved and not to be modified. However, bits 0 - 5 contain system flags that control the operating mode and state of the processor. Although SMSW only stores the machine status word, it is sensitive and unprivileged. You can see the problem if a guest OS is running in real mode within a hypervisor running in protected mode. If the VM checked the MSW to see if it was in real mode, it would incorrectly see that it was in protected mode (PE bit set) and could halt or shutdown and not be able to run successfully.

The next two sensitive instructions are PUSHF and POPF (and their 32-bit versions PUSHFD and POPFD). The issue with these instructions is similar to SMSW because pushing the EFLAGS register onto the stack allows examination of operating mode and state. POPF allows some of the EFLAGS bits to be changed. It varies based on the processor's current operating mode. In real-mode, or when operating at CPL 0, all non-reserved flags in the EFLAGS register can be modified except for the VM, VIP, and VIF flags. In virtual-8086 mode, the IOPL must equal 3 to use the POPF instructions. The IOPL allows an OS to set the privilege level needed to perform I/O. In virtual-8086 mode, these key flags are not affected by POPF. However, in protected mode, there are several conditions based on privilege levels. For example, if CPL is greater than 0 and <= to the IOPL, all flags can be modified (except IOPL). If POPF/POPFD is executed without enough privilege, an exception is NOT generated.

The next set of the seventeen sensitive instructions violate requirement 3C above (Protection System References). Namely, LAR (Load Access Rights byte), LSL (Load Segment Limit), VERR/VERW (Verify a segment for reading or writing). The problem with these instructions is they all perform the following check during their execution (CPL -> DPL) OR (RPL -> DPL). This condition checks to ensure that the current privilege level (located in bits 0 and 1 of the CS register and SS register) and the requested privilege level (bits 0 and 1 of any segment selector) are both greater than the descriptor privilege level (privilege level of a segment). This is a problem because prior to VMX and SVM, guests didn't normally execute at the highest privilege level (CPL 0). For example, in Xen, guests run at CPL 2 (ring-2) - they only "think" they are running at CPL 0. Therefore, if a guest running at CPL 2 executes any of LAR, LSL, VERR or VERW to examine a segment descriptor with a DPL < 3, it is likely that the instruction

will not execute properly.

POP and PUSH are also included in this category of problematic instructions for similar reasons. POP cannot be used to load the CS register since it contains the CPL. A value that is loaded into a segment register must be a valid segment selector. The reason that POP is one of the problematic seventeen instructions is it depends on the value of CPL. If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL, a general protection exception is raised. Furthermore, if the DS, ES, FS, or GS register is being loaded, the segment being pointed to is a nonconforming code segment or data, and the RPL and CPL are > the DPL, a general protection exception is raised. Therefore, as in the case with LAR, LSL, VERR and VERW, if the guest is at CPL 3 (ring 3) and did a privilege level check it would likely fail because it thinks it's running at CPL 0. If a process that thinks it's running at CPL 0 pushes CS onto the stack and checks it's CPL it will see that it's running at CPL 3 and may crash.

The next set of problematic instructions is CALL, JMP, INT n, and RETS CALL saves procedure linking information on the stack and branches to the procedure given in its destination argument. Naturally there are four types of calls (near, far calls to the same privilege level, far calls to a different privilege level, and task switches). Task switches and far calls to different privilege levels are a problem for virtualization because they involve CPL, DPL, and RPL. If a far call is executed to a different privilege level, the code segment for the procedure being accessed has to be accessed through the call gate. A task uses a different stack for every privilege level. Therefore, when a far call is made to another privilege level, the processor switches to a stack corresponding to the new privilege level of the called procedure. A task switch operates in a similar manner as a call gate. (The main difference being the target operand of the call instruction specifies the segment selector of a task gate instead of a call gate). Both call gate and task gate have many privilege level checks to compare the CPL and RPL to DPLs. Since the guest is running at CPL 2 or 3, these checks won't work properly with the guest OS tries to access call gates or task gates at CPL 0. The JMP and INT n instructions have similar problems for virtualization. (The INT n instruction references the protection system many times during it's execution). Naturally, the RET instruction has the opposite effect as CALL in that it transfers control to a return address placed on the stack (normally by CALL). The RET instruction can be used for three different types of returns: near, far, and inter-privilege-level returns. Much like the CALL instruction, the inter-privilege-level far return examines the privilege levels and access rights of the code and stacks segments that are being returned to determine if the operation should be allowed. The DS, ES, FS and GS segment registers are cleared by the RET instruction if they refer to segments that cannot be accessed by the new privilege level. Therefore,

RET is problematic for virtualization because a guest running at CPL 3 could cause the DS, ES, FS and GS segment registers to not be cleared when they should be.

The next problematic instruction of the seventeen instructions is STR (Store Task Register) because it references the protection system. The STR stores the segment selector from the task register into a general purpose register or memory location. The segment selector that is stored with this instruction points to the task state segment of the current executing task. This instruction is problematic for virtualization because it allows a task to examine its requested privilege level (RPL).

The last problematic instruction for virtualization is (believe it or not) MOV. The MOV opcode that stores segment registers allows all six of the segment registers to be stored to either a general purpose register or memory location. This is a problem because the CS and SS registers both contain the CPL in bits 0 and 1. Thus, a task could store the CS or SS in a general purpose register to find that it's not running at the expected CPL. The MOV opcode that loads segment registers does offer some protection because it won't allow the CS register to be loaded at all. However, if a task tries to load the SS register, several privilege level checks occur that become problematic for the reasons already explained.

2.3 Vanderpool (VMX) and Pacifica (SVM) to the Rescue

One approach to getting around the issues caused by the seventeen problematic x86 instructions is to avoid them all together by re-writing critical portions of the OS kernel to know when it’s running in a virtual machine and call out to the hypervisor when necessary. This is a process known as *para-virtualization.* Para-virtualization is the approach taken by early versions of Xen. The major drawback with respect to para-virtualization is it requires access to the OS kernel source code, which while appropriate for Linux, it takes Microsoft® Windows off the table.

Without access to the OS kernel source, another approach to getting around the issues caused by the seventeen problematic instructions is to implement *dynamic binary translation.* Dynamic binary translation involves trapping the seventeen problematic instructions and converting the source binary to a target binary program. Once trapped, the seventeen problematic instructions are funneled through the target binary program code instead of the source binary.

While para-virtualization and dynamic binary translation are effective solutions to the seventeen problematic x86 instructions, they are hardly

efficient solutions. The performance hit associated with para-virtualization and dynamic binary translation is not trivial.

Recognizing the growing importance of virtualization, Intel® and AMD® decided to address the seventeen problematic instructions by providing support for virtualization in hardware (silicon) and expanded the x86 ISA by ten (10) instructions. In doing so, they've eliminated the need for para-virtualization and dynamic binary translation. Hallelujah!

2.4 Intel® VT-x (VMX)

We'll focus on Intel's® implementation of virtualization because we've built our hypervisor around the Intel® technology and therefore we're most familiar with it as opposed to AMD's® SVM. While there are some difference between VMX and SVM, the differences are minor.

VT-x makes creating a hypervisor rather straight forward. The following are the basic steps required to set-up a hypervisor. First, you must make sure the processor is in 64-bit long mode with paging enabled. Then:

1) Check for VMX support in processor

   IF CPUID.1:ECX.VMX[5] = 1 THEN VMX is supported

2) Check to see if BIOS has disabled VMX

   IF IA32_FEATURE_CONTROL MSR (MSR Address 3AH) [2] = 0

   THEN BIOS has disabled VMX

   IF the Lock Bit (bit [0] of the same MSR is set:

   IF IA32_FEATURE_CONTROL MSR (MSR Address 3AH) [0] = 1

THEN the VMX bit (bit [2]) can not be set and VMX is absolutely disabled with no chance to enable it except via BIOS upgrade. IF the lock bit is cleared, then the VMX bit can be set via software.

3) Enable VMX

   SET CR4.VMXE[13] = 1

4) Create and populate the Virtual Machine Control Structure (VMCS)

5) Call **VMPTRLD** to make the VMCS active

6) Call **VMXON** to launch the hypervisor

The ten (10) VMX instructions added by Intel® to the IA32 ISA are:

VMXON / VMXOFF	Enable / Disable VMX Operation
VMCLEAR	Initialize VMCS Region
VMPTRLD / VMPTRST	Load / Store Current VMCS Pointer
VMREAD / VMWRITE	Read / Write VMCS Values
VMLAUNCH / VMRESUME	Launch or Resume Guest OS
VMCALL	Call from Guest OS into hypervisor

When VMX is enabled, the CPU runs in one of two modes:

VMX Root (fully privileged ring-0)
VMX non-Root (partially privileged ring-0)

The hypervisor runs in VMX root mode, while the guest OS runs in VMX non-root mode. A VMEXIT, which is a transition from VMX non-root guest OS mode to the hypervisor, can be caused by a number of events and conditions. There are specific and special I/O requests and other events that must be handled by the hypervisor, after which a transfer back to VMX non-root (guest OS) can be made.

A goal we held during the construction of our hypervisor was to keep it as small and lightweight as possible. We firmly believe that a smaller code base is easier to maintain and secure. We are looking to eventually store our hypervisor in Flash on the motherboard and size does matter in that deployment scenario.

VMX makes the construction and management of hypervisors accessible. Hypervisors are a very powerful because they provide an omnipresent purview over the platform. Because of the power of hypervisors and because VT-x has made creating them so accessible, there is an imperative in making sure your hypervisor gets loaded first in order to control or block other bare metal constructs that try to install after yours. There is tremendous value in getting there first. This is where UEFI and its pre-OS capabilities come in to the picture.

3 Unified Extensible Firmware Interface (UEFI)

UEFI is a bootloader and runtime interface between the platform firmware and the OS. UEFI is poised to replace the venerable BIOS we come to love and revile all these years. As 64-bit OSes begin to dominate the landscape, UEFI will emerge as the dominant supporting BIOS. Commodity PC motherboards are just now showing up in the marketplace and Microsoft® is

supporting UEFI 2.0 in Windows Vista x64 SP1. We are truly on the cusp of the advent of UEFI and all the power and utility it brings.

3.1 UEFI - A Programmer's Overview

UEFI describes a programmatic interface between the platform firmware and the OS. Two of the most outstanding aspects of UEFI are:

1) Its well defined set of interfaces that is highly flexible and scalable, which encourages innovation and creativity
2) Its rich and extensive pre-boot environment, which enables among other things pre-boot networking and pre-boot graphics support. We will use this pre-boot environment

A welcome aspect of UEFI is its high level language programming environment. Developing UEFI capabilities is very much a C-language type development activity. We use the TianoCore® SDK and our old linker and compiler friend, the Intel® Server2003 DDK to link and compile our UEFI components. As an example of the high level language nature of UEFI code, the following code example is from the TianoCore® SDK:

```
/*++

Copyright (c) 1998 Intel Corporation

Module Name:

    rtdriver.c

Abstract:

    Test runtime driver

Revision History

--*/

#include "efi.h"
#include "efilib.h"

EFI_STATUS
TestRtUnload (
    IN EFI_HANDLE       ImageHandle
    );

CHAR16  *RtTestString1 = L"This is string #1";
CHAR16  *RtTestString2 = L"This is string #2";
CHAR16  *RtTestString3 = L"This is string #3";
EFI_GUID RtTestDriverId = { 0xcc2ac9d1, 0x14a9,
0x11d3, 0x8e, 0x77, 0x0, 0xa0, 0xc9, 0x69, 0x72,
0x3b };
```

```
        EFI_STATUS
        InitializeTestRtDriver (
            IN EFI_HANDLE           ImageHandle,
            IN EFI_SYSTEM_TABLE     *SystemTable
            )
        {
            EFI_LOADED_IMAGE        *Image;
            EFI_STATUS              Status;

            // Initialize the Library.

            InitializeLib (ImageHandle, SystemTable);
            Print(L"Test RtDriver loaded\n");

            // Add an unload handler

            Status = BS->HandleProtocol (ImageHandle,
        &LoadedImageProtocol, (VOID*)&Image);
            ASSERT (!EFI_ERROR(Status));
            Image->Unload = TestRtUnload;

            // Add a protocol so someone can locate us

            Status = LibInstallProtocolInterfaces
        (&ImageHandle, &RtTestDriverId, NULL, NULL);
            ASSERT (!EFI_ERROR(Status));

            // Modify one pointer to verify fixups don't
            // reset it

            Print(L"Address of RtTestString3 is %x\n",
        RtTestString3);
            Print(L"Address of RtTestString3 pointer is
        %x\n", &RtTestString3);
            RtTestString3 = RtTestString2;
            return EFI_SUCCESS;
        }

        EFI_STATUS
        TestRtUnload (
            IN EFI_HANDLE       ImageHandle
            )
        {
            DEBUG ((D_INIT, "Test RtDriver unload being
        requested\n"));
            LibUninstallProtocolInterfaces (ImageHandle,
        &RtTestDriverId, NULL, NULL);
            return EFI_SUCCESS;
}
```

Fig. 1 – UEFI BIOS Code Example [6]

As you can see from Fig. 1, UEFI BIOS code looks very much like C-language device driver code. Gone are the days of 16-bit assembly language BIOS patching and coding. UEFI makes BIOS code writing accessible to a broader range of programmers, encouraging innovation and creativity.

The code example in Fig. 1 highlights a few important aspects of UEFI; the first we want to point out is the ***UEFI Systems Table.***

The UEFI Systems Table is the most important data structure in UEFI [7]. A pointer to the UEFI Systems Table is passed into each driver and application as part of its entry point. From the UEFI Systems Table, an UEFI executable image can access important UEFI services, which include:

1) Protocol Services
2) UEFI Runtime Services (we're particularly interested in Runtime Services as part of this thesis)
3) UEFI Boot Services

A UEFI Protocol is a block of function pointers and data structures. The UEFI specification defines a number of protocols, but developers can extend functionality by defining their own protocols. There are a number of protocols defined in the UEFI SDK that extend capabilities and are not defined by the UEFI specification [8]. One example of a TianoCore® protocol that extends the specification is the pppd protocol. The UEFI pppd protocol is UEFI Point-to-Point Protocol Daemon that provides a standard way to establish network connectivity over serial. The point is that via UEFI protocols, individual developers can easily add their own UEFI capabilities by writing their own UEFI protocols.

The Boot Services and Runtime Services are accessed via a UEFI Boot Services Table and UEFI Runtime Services Table, which are both data fields in the UEFI Systems Table.

UEFI images contain the old familiar PE/COFF header that defines the format of the executable code. The header defines the processor type (IA-32, Itanium®, or the processor neutral type UEFI Byte Code). The header also defines the image type:

1) UEFI Application
2) UEFI Boot Services Drivers
3) UEFI Runtime Drivers

UEFI Applications and UEFI Boot Services Drivers run in the pre-OS environment of UEFI, that is, they run before the OS bootloader is called. It is quite possible to establish network connectivity, run all manner of

applications, and invoke rich graphics support, etc. all before the OS is called.

However, the memory allocated by UEFI Applications is reclaimed when the UEFI application exits. In the case of a UEFI Boot Services Driver the memory allocated is reclaimed when the OS bootloader calls ***ExitBootServices()***. UEFI applications and Boot Services do not persist beyond the boot phase; they are not available once the OS loads.

Of these three image types, only the **UEFI Runtime Driver** image ensures that its memory and state persist beyond the bootloader phase, i.e. the driver functionality remains available during the lifetime of the OS instance. UEFI Runtime drivers are loaded in memory marked as *EfiRuntimeServicesCode,* and their data structures are reserved as *EfiRuntimeServicesData.* These types of memory are not reclaimed and preserved after the *ExitBootServices() is called.* UEFI Runtime Drivers coexist with and can be invoked by a UEFI-aware OS. It is this characteristic of the UEFI Runtime Driver that makes it well suited to the construction of a UEFI Hypervisor.

4 Summary

The race to bare metal is on! Bare metal capabilities are stealthy, powerful and growing in numbers. We've focused on the hypervisor as our bare metal capability of choice, but there are others such as SMM rootkits; others will surely come. There is tremendous advantage in getting your bare metal capability on the platform first because you can control everything that comes afterward. UEFI provides the opportunity to get your bare metal capability on platform first, even before the OS bootloader is called.

Bibliography / References

[1] http://www.microsoft.com/mscorp/twc/default.mspx
[2] Howard, Michael, AND LeBlanc, David, *Writing Secure Code*, ISBN-7-7356-1588-8.
[3] Rutkowska, Joanna, *Subverting Vista Kernel for Fun and Profit,* Proceedings of Black Hat 2006.
[4] Embleton, Shawn, AND Sparks, Sherri, *A New Breed of Rootkit, The System Management Mode (SMM) Rootkit*, Proceedings of Black Hat 2008.
[5] Popek, G. J., AND Goldberg, R. P. *Formal requirements for virtualizable third generation architectures*. *Commun.* ACM *17, 7* (1974) 412-421.
[6] EFI_Toolkit_2.0.0.1/apps/rtdriver/rtdriver.c
[7] Zimmerman, Vincent, AND Rothman, Michael, AND Hale, Robert, *Beyond BIOS*, ISBN-0-9743649-0-8
[8] http://www.uefi.org/specs/