USENIX Association

Proceedings of the 9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor

John Scott Robin
U.S. Air Force
scott_robin_@hotmail.com

Cynthia E. Irvine *
Naval Postgraduate School
irvine@cs.nps.navy.mil
http://cisr.nps.navy.mil

Abstract

A virtual machine monitor (VMM) allows multiple operating systems to run concurrently on virtual machines (VMs) on a single hardware platform. Each VM can be treated as an independent operating system platform. A secure VMM would enforce an overarching security policy on its VMs.

The potential benefits of a secure VMM for PCs include: a more secure environment, familiar COTS operating systems and applications, and enormous savings resulting from the elimination of the need for separate platforms when both high assurance policy enforcement, and COTS software are required.

This paper addresses the problem of implementing secure VMMs on the Intel Pentium architecture. The requirements for various types of VMMs are reviewed. We report an analysis of the virtualizability of all of the approximately 250 instructions of the Intel Pentium platform and address its ability to support a VMM. Current "virtualization" techniques for the Intel Pentium architecture are examined and several security problems are identified. An approach to providing a virtualizable hardware base for a highly secure VMM is discussed.

1 Introduction

A virtual machine monitor (VMM) is software for a computer system that creates efficient, isolated programming environments that are "duplicates" which provide users with the appearance of direct access to the real machine environment. These duplicates are referred to as virtual machines. Goldberg [12] defines a virtual machine (VM) as: "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode". A VMM manages the real resources of the computer system, exporting them to virtual machines.

A VMM offers a number of benefits not found in conventional multiprogramming systems.

1.1 VMM Benefits

First, virtual machine monitors normally allow a system manager to **configure the environment** in which a VM will run. VM configurations can be different from those of the real machine. For example, a real machine might have 32MB of memory, but a virtual machine might have only 8 MB. This would allow a developer to test an application on a machine with only 8 MB of memory without having to construct a hardware version of that real machine.

Second, virtual machine monitors allow **concurrent execution of different operating systems** on the same hardware. Users can run any operating system and applications designed to run on the real processor architecture. Thus application development for different operating systems is easier. A developer can easily test applications on many operating systems simultaneously while running on the same base platform.

Third, virtual machine monitors allow users to **isolate untrusted applications of unknown quality**. For example, a program downloaded from the Internet could be tested in a VM. If the program contained a virus, the virus would be isolated to that VM. A secure VMM will ensure that other high integrity VMs and their applications and data are protected from corruption.

Fourth, virtual machine monitors can be used to **upgrade** operating system software to a different version without losing the ability to run the older "legacy" operating system and its applications. The legacy software can run in a virtual machine exactly as it did previously on the real machine, while the new version of the operating system runs in a separate virtual machine.

Finally, VMMs can be used to construct **system software for scalable computers** that have anywhere from 10 to 100 processors. VMMs can facilitate the develop-

*The opinions in this paper are those of the authors and should not be construed to reflect those of their employers.

ment of functional and reliable system software for these computers.

Using a VMM, an additional software layer can be inserted between the hardware and multiple operating systems. This VMM layer would allow multiple copies of an operating system to run on the same scalable computer. The VMM also allows these operating systems to share resources with each other. This solution has most of the features of an operating system custom-built for a scalable machine, but with lower development costs and reduced complexity. Disco, developed for the Stanford FLASH shared-memory multi-processor [8] is an example of this solution. It uses different commercial operating systems to provide high-performance system software.

1.2 VMM Characteristics and Layers

A VMM has three characteristics [28]. First, a VMM provides **an execution environment almost identical to the original machine**; any program executed on a VM should run the same as it would on an unvirtualized machine. Exceptions to this rule result from differences in system resource availability, timing dependencies, and attached I/O devices. If resource availability, e.g. physical memory, is different, the program will perform differently. Timing dependencies may lose their validity because a VMM may intervene and execute a different set of instructions when certain instructions are executed by a VM. Finally, if the VM is not configured with all of the peripheral devices required by the real machine, application behavior will differ.

Second, a VMM must be **in control of real system resources**. No program running on a VMM can access any resource not explicitly allocated to it by the VMM. Also the VMM can regain control of previously allocated resources.

Efficiency is the third VMM characteristic. A large percentage of the virtual processor's instructions must be executed by the machine's real processor, without VMM intervention. Instructions which cannot be executed directly by the real processor are interpreted by the VMM. Some virtual machines exhibit the recursion property: it is possible to run a VMM inside of a VM, producing a new level of virtual machines. The real machine is normally called Level 0. A VMM running on Level 0 is said to be Level 1, etc.

1.3 VMM Logical Modules

A VMM normally has three generic modules: dispatcher, allocator, and interpreter. A jump to the dispatcher is placed in every location to which the machine traps. The dispatcher then decides which of its modules to call when a trap occurs. The second type of module is the allocator. If a VM tries to execute a privileged instruction that would change the resources of the VM's environment, the VM will trap to the VMM dispatcher. The dispatcher will handle the trap by invoking the allocator that performs the requested resource allocation according to VMM policy. A VMM has only one allocator module, however, it accounts for most of the complexity of the VMM. It decides which system resources to provide to each VM, ensuring that two different VM's do not get the same resource. The final module type is the interpreter. For each privileged instruction, the dispatcher will call an interpreter module to simulate the effect of that instruction. This prevents VMs from seeing the actual state of the real hardware. Instead they see only their virtual machine state.

1.4 Attractions of a Secure VMM

An isolated VM constrained by an overarching security policy enforced by the underlying secure VMM is attractive. Also, VMM technology provides stronger isolation of virtual machines than found in conventional multiprogramming environments [21]. Within a constrained VM, legacy operating systems and applications are executed unmodified and are easily upgraded and replaced even within the context of rapidly evolving software product lifecycles.

In the past, some virtual machine monitors, such as the SDC KVM/370 [11, 9, 33, 10] and the DEC VAX SVS [17], have been used to separate mandatory security classes. A secure VMM for the Intel Pentium[1] processor architecture would be very desirable because a single machine could be used to implement critical security policies while also running popular Win32 operating systems and applications.

Although the x86 processor family has been used as the base for many highly secure systems [23, 27, 26, 25, 24], it has not been considered as a VMM base. Recent increased interest in VMM technology suggests that a popular hardware base for a new generation of VMMs would be highly attractive. Before embarking on such a venture, its feasibility must be carefully examined. This paper presents an analysis to determine whether the Intel Pentium architecture can support a highly secure VMM without sacrificing user convenience.

[1] Throughout this paper, the term "Intel Pentium architecture" will refer to the architecture of the following processors, which are all trademarks of the Intel Corporation: Intel Pentium, Intel Pentium Pro, Intel Pentium with MMX Technology, Intel Pentium II and Intel Pentium III.

1.5 Paper Organization

The rest of this paper is organized as follows: Section 2 discusses the three different types of VMMs and their hardware requirements. Section 3 is an analysis of the Intel Pentium architecture with respect to the VMM hardware requirements described in Section 2. Section 4 asks if a VMM designed for the Intel Pentium architecture can be secure. Finally, Section 5 presents our conclusions and future research.

2 VMM Requirements

This section discusses each type of VMM including the Type I VMM, Type II VMM, and Hybrid VMM. It will also cover the architectural features required for the successful implementation for each VMM type.

2.1 Virtual Machine Monitors Types

An operating system consists of instructions to be executed on a hardware processor. When an operating system is virtualized, some portion, ranging from none to all, of the instructions may be executed by underlying software. The amount of software and hardware execution of processor instructions determines if one has a complete software interpreter machine (CSIM), hybrid VM (HVM), VMM, or a real machine. Each of these different types of machines provides a normal machine environment, meaning that processor instructions can be executed on them, viz. a VMM can host an operating system. However, they differ in the way that the machine environment actually executes the processor instructions. A real machine uses only direct execution: the processor executes every instruction of the program directly. A CSIM uses only software interpretation: a software program emulates every processor instruction. There has been a recent resurgence of interest in CSIM architectures [3, 18]. A VMM requires that a "statistically dominant subset" of the virtual processor's instructions be executed on the real processor [12]. Performance will be effected by the size of the subset.

VMMs primarily use direct execution, with occasional traps to software. As a result, the performance of VMMs is better than CSIMs and HVMs. An HVM is a VMM that uses software interpretation on all privileged instructions. HVMs are possible on a larger class of systems than VMMs. The definition of a VMM does not specify how the VMM gains control of the machine to interpret instructions that cannot be directly executed on the processor. As a result, there are two different types of VMMs that can create a virtual machine environment. These types are referred as Type I and Type II [12]. A Type I VMM runs on a bare machine. It is an operating system with virtualization mechanisms. It performs the scheduling and allocation of the system's resources. A Type II VMM runs as an application. The operating system that controls the real hardware of the machine is called the "host OS." The host OS does not need or use any part of the virtualization environment. Every OS that is run in the Type II virtual environment is called a "guest OS." In a Type II VMM, the host operating system provides resource allocation and a standard execution environment to each guest OS.

2.2 Execution of Privileged Instructions

When executing in a virtual machine, some processor instructions can not be executed directly on the processor. These instructions would interfere with the state of the underlying VMM or host OS and are called sensitive instructions. The key to implementing a VMM is to prevent the direct execution of sensitive instructions. Some sensitive instructions in the Intel Pentium architecture are privileged, meaning that if they are not executed at most privileged hardware domain, they will cause a general protection exception. Normally, a VMM is executed in privileged mode and a VM is run in user mode; when privileged instructions are executed in a VM, they cause a trap to the VMM. If all sensitive instructions of a processor are privileged, the processor is considered to be "virtualizable:" then, when executed in user mode, all sensitive instructions will trap to the VMM. After trapping, the VMM will execute code to emulate the proper behavior of the privileged instruction for the virtual machine. However, if sensitive, non-privileged instructions exist, it may be necessary for the VMM to examine all instructions before execution to force a trap to the VMM when a sensitive, non-privileged instruction is encountered.

The most severe performance penalty occurs when running a complete software interpreter machine (CSIM) on the same hardware. A CSIM emulates every instruction of the real processor. It does not meet Goldberg's definition [12] of a virtual machine because it does not execute any of the instructions directly on the real processor.

The following sections summarize Goldberg's analysis of processor requirements for the types of VMMs he identified.

2.3 Type I VMM

A Type I VMM runs directly on the machine hardware. It is an operating system or kernel that has mechanisms to support virtual machines. It must perform scheduling and resource allocation for all virtual machines in the system and requires drivers for hardware peripherals.

To support a Type I VMM, a processor must meet three virtualization requirements:

Requirement 1 The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. For example, a processor cannot use an additional bit in an instruction word or in the address portion of an instruction when in privileged mode.

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

Requirement 3 There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.

Sensitive instructions include:

Requirement 3A Instructions that attempt to change or reference the mode of the VM or the state of the machine.

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

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

Requirement 3D All I/O instructions.

2.4 Type II VMM

A Type II VMM runs as an application on a host operating system and relies on the host OS for memory management, processor scheduling, resource allocation, and hardware drivers. It provides only virtualization support services. To support a Type II virtual machine a processor must meet all of the hardware requirements for the Type I VMM listed above. In addition, the following software requirements must be met by the host operating system of the Type II VMM:

Weaker Requirement 3A The *host OS* cannot do anything to invalidate the requirement that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.

Requirement 2 Primitives must be available in the *host OS* to protect the VMM and other VMs from the active virtual machine. Examples include a protection primitive, address translation primitive, or a sub-process primitive.

When the virtual machine traps because it has attempted to execute a sensitive instruction, the host OS must direct the signal to the VMM. Therefore, the host OS needs a primitive to perform this action. The host OS also needs a mechanism to allow a VMM to run the virtual machine as a sub-process. The VMM must be able to simulate sensitive instructions.

A highly secure Type II VMM will require a highly secure host OS because it will depend upon the host OS. Flaws in the host OS would undermine the security of the Type II VMM.

2.5 Hybrid VMM

Often, if a processor does not meet the Type I or Type II VMM requirements, it can still implement a hybrid virtual machine monitor (HVM). A hybrid VMM has all of the advantages of normal VMMs and avoids the performance penalties of a CSIM. It is functionally equivalent to the real machine. However, an HVM and a VMM differ in that an HVM interprets every privileged instruction in software, whereas a VMM may directly execute some privileged instructions. An HVM treats the privileged mode of hardware as a pure software construct. In both a VMM and an HVM, all non-privileged instructions execute directly on the processor.

An HVM has less strict hardware requirements than a VMM in tow ways. First, the HVM does not have to directly execute non-sensitive privileged instructions because they are all emulated in software. Second, because of the emulation, the HVM need not provide additional mapping of the the most privileged processor mode into another processor privilege level. However, increased interpretative execution usually lowers the performance of an HVM relative to a VMM.

The hardware requirements for an HVM result in the following changes to the original Type I VMM requirements. First, Requirement 1, which states that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode, is eliminated. Second, Requirement 3A, which states that if an instruction attempts to change or reference the mode of the VM or the state of the machine, there must be a way to simulate the instruction, is weakened.

3 Pentium Architecture and VMMs

Goldberg [12] identified the key architectural features of third generation hardware pertinent to virtual machines:

- two processor modes of operation,
- a method for non-privileged programs to call privileged system routines,
- a memory relocation or protection mechanism such as segmentation or paging, and

- asynchronous interrupts to allow the I/O system to communicate with the CPU.

All of these still apply to the Intel Pentium architecture. It has four modes of operation, known as rings, or current privilege level (CPL), 0 through 3. Ring 0, the most privileged, is occupied by operating systems. Application programs execute in Ring 3, the least privileged. The Pentium also has a method to control transfer of program execution between privilege levels so that non-privileged tasks can call privileged system routines: the *call gate*. The Pentium also uses both paging and segmentation to implement its protection mechanisms. Finally, the Pentium uses both interrupts and exceptions to allow the I/O system to communicate with the CPU. The architecture has 16 predefined interrupts and exceptions and 224 user-defined, or maskable, interrupts.

Despite these features, the ability of the Pentium architecture to support virtualization is likely to be serendipitous as the processor was not explicitly designed to support virtualization. This section reports an analysis of the virtualizability of the Pentium against the hardware requirements described in Section 2. Every documented instruction for the Intel Pentium [2] was analyzed for its ability to support virtualization [30].

Any instruction in the processor's instruction set that violates rule 1, 2, 3 (3A, 3B, 3C, or 3D) will preclude the processor from running a Type I or Type II VMM. Additionally, any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D prevents the processor from running an HVM. By combining these two statements, one can see that any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D makes the processor non-virtualizable.

With respect to the VMM hardware requirements listed above, Intel meets all three of the main requirements for virtualization.

Requirement 1: *The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.* Intel meets this requirement because the method for executing privileged and non-privileged instructions is the same. The only difference between the two types of instructions in the Intel architecture is that privileged instructions cause a general protection exception if the CPL is not equal to 0.

Requirement 2:*There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.* Intel uses both segmentation and paging to implement its protection mechanism. Segmentation provides a mechanism to divide the linear address space into individually protected address spaces (segments). Segments have a *descriptor privilege level* (DPL) ranging from 0 to 3 that specifies the privilege level of the segment. The DPL is used to control access to the segment. Using DPLs, the processor can enforce boundaries between segments to control whether one program can read from or write into another program's segments.

Requirement 3:*There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.* The Intel architecture uses interrupts and traps to redirect program execution and allow interrupt and exception handlers to execute when a privileged instruction is executed by an unprivileged task. However, the Pentium instruction set contains **sensitive, unprivileged instructions**. The processor will execute unprivileged, sensitive instructions without generating an interrupt or exception. Thus, a VMM will never have the opportunity to simulate the effect of the instruction.

After examining each member of the Pentium instruction set, it was found that seventeen instructions violate Requirement 3. All seventeen instructions violate either part B or part C of Requirement 3 and make the Intel processor non-virtualizable. To construct a truly virtualizable Pentium chip one must focus on these instructions. They are discussed in more detail below.

3.1 Sensitive Register Instructions

Several Intel instructions break hardware virtualization Requirement 3B. The rule states that instructions are sensitive if they read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.

3.1.1 SGDT, SIDT, and SLDT Instructions

The SGDT, SIDT, and SLDT instructions violate this rule in a similar way. In protected mode, all memory accesses pass through either the *global descriptor table* (GDT) or *local descriptor table* (LDT). The GDT and LDT contain segment descriptors that provide the base address, access rights, type, length, and usage information for each segment. The interrupt descriptor table (IDT) is similar to the GDT and LDT, but it holds gate descriptors that provide access to interrupt and exception handlers. The GDTR, LDTR, and IDTR all contain the linear addresses and sizes of their respective tables.

All three of these instructions (SGDT, SIDT, SLDT) store a special register value into some location. The SGDT instruction stores the contents of the GDTR in a 6-byte memory location. The SLDT instruction stores the segment selector from the LDTR in a 16 or 32-bit general-purpose register or memory location. The SIDT

[2]The analysis was based on available documentation as of 22 June 1999 and involved approximately 250 instructions.

Table 1: **Important CR0 Machine Status Word Bits**

Bit	Flag Name	Description
0	PE - Protection Enable	Enable Protected Mode when set and real mode when clear
1	MP - Monitor Coprocessor	Controls the interaction of the WAIT or FWAIT instruction with the TS flag.
2	EM - Emulation	If clear, processor has an internal or external floating point unit
3	TS - Task Switched	Allows delayed saving of the floating point unit context on a task switch until the unit is accessed by the new task.
4	ET - Extension Type	For 386 and 468 processors, indicates whether an Intel 387 DX math co-processor is present (hard-coded to 1 on >Pentium processors).
5	NE - Numeric Error	Enables internal or PC-style mechanism for FPU error reporting.

instruction stores the contents of the IDTR in a 6-byte memory location. These instructions are normally only used by operating systems but are not privileged in the Intel architecture. Since the Intel processor only has one LDTR, IDTR, and GDTR, a problem arises when multiple operating systems try to use the same registers. Although these instructions do not protect the sensitive registers from reading by unprivileged software, the processor allows partial protection for these registers by only allowing tasks at CPL 0 to load the registers. This means that if a VM tries to write to one of these registers, a trap will be generated. The trap allows a VMM to produce the expected result for the VM. However, if an OS in a VM uses SGDT, SLDT, or SIDT to reference the contents of the IDTR, LDTR, or GDTR, the register contents that are applicable to the host OS or Type I VMM will be given. This could cause a problem if an operating system of a virtual machine (VMOS) tries to use these values for its own operations: it might see the state of a different VMOS executing within a VM running on the same VMM. Therefore, a Type I VMM or Type II VMM must provide each VM with its own virtual set of IDTR, LDTR, and GDTR registers.

3.1.2 SMSW Instruction

The SMSW instruction stores the machine status word (bits 0 through 15 of control register 0) into a general-purpose register or memory location. Bits 6 through 15 of CR0 are reserved bits that are not supposed to be modified. Bits 0 through 5, however, contain system flags that control the operating mode and state of the processor and are described in Table 1.

Although this instruction only stores the machine status word, it is sensitive and unprivileged. Consider the following scenario: A VMOS is running in real mode within the virtual environment created by a VMM running in protected mode. If the VMOS checked the MSW to see if it was in real mode, it would incorrectly see that the PE bit is set. This means that the machine is in protected mode. If the VMOS halts or shuts down if in protected mode, it will not be able to run successfully.

This instruction is only provided for backwards compatibility with the Intel 286 processor [16]. Programs written for the Intel 386 processor and later are supposed to use the MOV instruction to load and store control registers, which are privileged instructions. Therefore, SMSW could be removed and only systems requiring backward compatibility with the Intel 286 processor would be affected. Application software written for the Intel 286 and 8086 processors should be unaffected because the SMSW instruction is a system instruction that should not be used by application software.

3.1.3 PUSHF and POPF Instructions

The PUSHF and POPF instructions reverse each other's operation. The PUSHF instruction pushes the lower 16 bits of the EFLAGS register onto the stack and decrements the stack pointer by 2. The POPF instruction pops a word from the top of the stack, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. The PUSHFD and POPFD instructions are the 32-bit counter-parts of the POPF and PUSHF instructions. Pushing the EFLAGS register onto the stack allows the contents of the EFLAGS register to be examined. Much like the lower 16 bits of the CR0 register, the EFLAGS register contains flags that control the operating mode and state of the processor. Therefore, the PUSHF/PUSHFD instructions prevent the Intel processor from being virtualizable in the same way that the SMSW instruction prevents virtualization. In virtual-8086 mode, the IOPL must equal 3 to use the PUSHF instructions. Of the 32 flags in the EFLAGS register, fourteen are reserved and six are arithmetic flags. Table 2 describes the bits of concern.

The POPF instruction allows values in the EFLAGS register to be changed. Its 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

Table 2: **Important EFLAGS Register Bits**

Bit	Flag Name	Description
8	TF - Trap	Set to enable single-step mode for debugging.
9	IF - Interrupt Enable	Controls processor response to maskable interrupt requests.
10	DF - Direction	If set, string instructions process addresses from high to low.
12-13	IOPL - I/O Privilege Level	I/O privilege level of the currently running task.
14	NT - Nested Task	Set when the current task is linked to the previous task.
16	RF - Resume	Controls processor response to debug exceptions.
17	VM - Virtual-8086 Mode	Enables Virtual-8086 mode when set.
18	AC - Alignment Check	Enables alignment checking of memory references.
19	VIF - Virtual Interrupt	Virtual image of the IF flag.
20	VIP - Virtual Interrupt Pending	Indicates whether or not an interrupt is pending.
21	ID - Identification	If a program can set or clear this instruction, the CPUID instruction is supported.

register can be modified except 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, the VM, RF, IOPL, VIP, and VIF flags are unaffected by the POPF instruction. In protected mode, there are several conditions based on privilege levels. First, if the CPL is greater than 0 and less than or equal to the IOPL, all flags can be modified except IOPL, VIP, VIF, and VM. The interrupt flag is altered when the CPL is at least as privileged as the IOPL. Finally, if a POPF/POPFD instruction is executed without enough privilege, an exception is not generated. However, the bits of the EFLAGS register are not changed.

The POPF/POPFD instructions also prevent processor virtualization because they allow modification of certain bits in the EFLAGS register that control the operating mode and state of the processor.

3.2 Protection System References

Many Intel instructions violate Requirement 3C: Instructions are sensitive if they reference the storage protection system, memory or address relocation system.

3.2.1 LAR, LSL, VERR, VERW

Four instructions violate the rule in a similar manner: LAR, LSL, VERR, and VERW. The LAR instruction loads access rights from a segment descriptor into a general purpose register. The LSL instruction loads the unscrambled segment limit from the segment descriptor into a general-purpose register. The VERR and VERW instructions verify whether a code or data segment is readable or writable from the current privilege level. The problem with all four of these instructions is that they all perform the following check during their execution: (CPL $\rightarrow$ DPL) OR (RPL $\rightarrow$ DPL). This conditional checks to ensure that the current privilege level (located in bits 0 and 1 of the CS register and the SS register) and the requested privilege level (bits 0 and 1 of any segment selector) are both greater than the descriptor privilege level (the privilege level of a segment). This is a problem because a VM normally does not execute at the highest privilege (i.e., CPL $= 0$). It is normally executed at the user or application level (CPL $= 3$) so that all privileged instructions will cause traps that can be handled by the VMM. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VMOS running at CPL $= 3$ uses any of the four instructions listed above to examine a segment descriptor with a DPL < 3, it is likely that the instruction will not execute properly.

3.2.2 POP Instruction

The reason that the POP instruction prevents virtualization is very similar to that mentioned in the previous paragraph. The POP instruction loads a value from the top of the stack to a general-purpose register, memory location, or segment register. However, the POP instruction 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 prevents virtualization is because it depends on the value of the 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. Additionally, 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 greater than the DPL, a general protection exception is raised. As in the previous case, if a VM's CPL

is 3, these privilege level checks could cause unexpected results for a VMOS that assumes it is in CPL 0.

3.2.3 PUSH Instruction

The PUSH instruction also prevents virtualization because it references the protection system. The PUSH instruction allows a general-purpose register, memory location, an immediate value, or a segment register to be pushed onto the stack. This cannot be allowed because bits 0 and 1 of the CS and SS register contain the CPL of the current executing task. The following scenario demonstrates why these instructions could cause problems for virtualization. A process that thinks it is running in CPL 0 pushes the CS register to the stack. It then examines the contents of the CS register on the stack to check its CPL. Upon finding that its CPL is not 0, the process may halt.

3.2.4 CALL, JMP, INT n, and RET

The CALL instruction saves procedure linking information to the stack and branches to the procedure given in its destination operand. There are four types of procedure calls: near calls, far calls to the same privilege level, far calls to a different privilege level, and task switches. Near calls and far calls to the same privilege level are not a problem for virtualization. Task switches and far calls to different privilege levels are problems because they involve the 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 a 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 manner similar to a call gate. The main difference is that the target operand of the call instruction specifies the segment selector of a task gate instead of a call gate. Both call gates and task gates have many privilege level checks that compare the CPL and RPL to DPLs. Since the VM normally operates at user level (CPL 3), these checks will not work correctly when a VMOS tries to access call gates or task gates at CPL 0.

The discussion above on LAR, LSL, VERR, and VERW provides a specific example of how running a CPL 0 operating system as a CPL 3 task could cause a problem. The JMP instruction is similar to the CALL instruction in both the way that it executes and the reasons it prevents virtualization. The main difference between the CALL and the JMP instruction is that the JMP instruction transfers program control to another location in the instruction stream and does not record return information.

The INT instruction is also similar to the CALL instruction. The INT n instruction performs a call to the interrupt or exception handler specified by n. INT n does the same thing as a far call made using the CALL instruction except that it pushes the EFLAGS register onto the stack before pushing the return address. The INT instruction references the protection system many times during its execution.

The RET instruction has the opposite effect of the CALL instruction. It transfers program control to a return address that is placed on the stack (normally by a CALL instruction). 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 stack 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 prevents virtualization because having a CPL of 3 (the VM's privilege level) could cause the DS, ES, FS, and GS registers to not be cleared when they should be. The IRET/IRETD instruction is similar to the RET instruction. The main difference is it returns control from an exception, interrupt handler, or nested task. It prevents virtualization in the same way that the RET instruction does.

3.2.5 STR Instruction

Another instruction that references the protection system is the STR instruction. The STR instruction 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 currently executing task. This instruction prevents virtualization because it allows a task to examine its requested privilege level (RPL). Every segment selector contains an index into the GDT or LDT, a table indicator, and an RPL. The RPL is represented by bits 0 and 1 of the segment selector. The RPL is an override privilege level that is checked (along with the CPL) to determine if a task can access a segment. The RPL is used to ensure that privileged code cannot access a segment on behalf of an application unless the application also has the privilege to access the segment. This is a problem because a VM does not execute at the highest CPL or RPL (RPL = 0), but at RPL = 3. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VM running at a CPL and RPL of 3 uses STR to store the contents of the task register and then examines the infor-

mation, it will find that it is not running at the privilege level at which it expects to run.

3.2.6 MOVE Instruction

Two variants of the MOVE instruction prevent Intel processor virtualization. These are the two MOV instructions that load and store control registers. The MOV opcode that stores segment registers allows all six of the segment registers to be stored to either a general-purpose register or to a 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 and examine the contents of that register to find that it is not operating at the expected privilege level. The MOV opcode that loads segment registers does offer some protection because it does not allow the CS register to be loaded at all. However, if the task tries to load the SS register, several privilege checks occur that become a problem when the VM is not operating at the privilege level at which a VMOS is expecting–typically 0.

The analysis of Section 3 shows that the Intel processor is not virtualizable according to Goldberg's hardware rules.

4 Pentium-Based "VMM" Security

This section will examine several security issues for a VMM designed for the Intel Pentium architecture. We begin with a brief review of previous secure VMMs. Second, use of Intel processors for highly secure systems is discussed. Third, ways to provide virtual machine monitors on unmodified Intel platforms are examined to gain insight into the challenges faced in a virtual machine monitor effort. Next we discuss the security impact of using unmodified Intel platforms for VMMs. Finally, a better approach to creating a highly secure VMM on the Intel architecture is covered.

4.1 Are Secure VMMs Possible?

An early discussion of VMMs and security argued that the isolation provided by a combined VMM/OS provided better software security than a conventional multiprogramming operating system. It was also suggested that the redundant security mechanisms found in the VMM and the OS executing in one of its virtual machines enhanced security [21]. Penetration experiments indicated that redundant weak implementations are insufficient to secure a system [5, 11].

KVM/370 was an early secure Type I VMM [11, 33, 9]. Called a "security retrofit," two approaches to the work were examined: (1) "hardening" of the existing VM/370 control program (CP) to repair identified penetration vulnerabilities and (2) a redesign of the VM/370 CP to place all security-relevant functionality within a formally verified security kernel based upon the reference monitor concept [4]. (Note that the first approach was abandoned because flaw remediation did not provide a guarantee of the absence of yet undetected, exploitable security flaws.) The redesigned system consisted of four domains:

1. A minimized security kernel and verified trusted processes executing in supervisor state.
2. Semi-trusted processes executing in real problem state. These processes managed some global data, were audited, had access only to virtual addresses.
3. Non-kernel control programs (NKCPs) that executed the non-security relevant bulk of the VM/370 control program in real problem state. Each NKCP executed at a single security level and had access only to virtual addresses.
4. User VMs executing in real problem state under the control of a NKCP, with the same security level as the NKCP.

A security kernel is defined as hardware and software that implements the reference monitor concept [4]. A reference monitor enforces authorized access relationships between the subjects and objects within a system. It imposes three design requirements on its implementations:

1. The mechanism must be tamperproof.
2. The mechanism must always be invoked.
3. The mechanism must be small enough to be to subject to analysis and tests to ensure completeness.

The VAX Security Kernel[17] was a highly secure Type I VMM. The system's hardware, microcode, and software were designed to meet TCSEC Class A1 assurance and security requirements [22]. The project also maintained standard VMS and Ultrix-32 interfaces to run COTS operating systems and applications in virtual machines.

The VAX VMM security kernel allowed multiple virtual machines to run concurrently on a single VAX system. It could support a large number of simultaneous users and provided isolation and controlled sharing of sensitive data.

The VAX processor, much like the Intel Pentium processor, contained several sensitive, unprivileged instructions. It also had four rings. The security kernel designers modified the VAX processor microcode to make it

virtualizable. The four instructions that prevented virtualization on the VAX processor were: CHM, REI, MOVPSL, and PROBE [13]. The CHM instruction switches to a mode of equal or increased privilege. The REI instruction switches to a mode of equal or decreased privilege. The MOVPSL instruction is used to read the Processor Status Longword (similar to the machine status word in the Intel architecture). The PROBE instruction is used to determine the accessibility of a page of memory. These four instructions read or write one of the following pieces of sensitive data: the current execution mode, the previous execution mode, the modify bit of a page table entry, and the protection bit of a page table entry.

To support compatibility with existing operating systems and applications, some of the microcode changes included: defining a new VM mode bit, defining a new register called VMPSL, defining a VM-emulation exception, as well as the four instructions described above.

Ring compression, implemented entirely in software, was used to avoid certain processor modifications. The protection between compressed layers is weakened; however, this choice had little security impact since, although the VMS operating system for the VAX used all four rings, all three inner rings were in fact used for fully trusted operating system software.

The VAX I/O hardware was difficult to virtualize because its I/O mechanisms read and write various control and status registers in the I/O space of physical memory. To overcome this difficulty, the VAX security kernel I/O interface used a special, performance-optimized kernel call mechanism. To use this mechanism, a virtual machine executed a Move To Privileged Register (MTPR) instruction to a special kernel call register. The MTPR instruction trapped the security kernel software that performed the I/O. Untrusted device drivers were written for each guest OS in order to run on the VMM.

The VAX security kernel applied mandatory and discretionary access controls to virtual machines. The kernel assigned every virtual machine an access class consisting of a secrecy class (based on the Bell and LaPadula model [6]) and an integrity class (based on the Biba model [7]). The kernel supported access control lists on all objects including real devices, disk and tape volumes, and security kernel volumes. The VMM security kernel differed from a typical secure operating system because the subjects and objects are virtual machines and virtual disks, not files and processes, which are implemented by each guest OS.

It is worth noting that timing channels in VMMs [33] were addressed in the context of the VAX VMM work [14]. Despite the challenge of timing channel mitigation, VMMs provide a solution to the problem of sharing while running legacy or commercial code securely with firewalling between the VMs managed by a highly secure VMM kernel.

The VAX security effort lead to several conclusions: (1) Every ring of a processor can be emulated, but this is often not necessary. (2) Emulating a start I/O instruction is simpler and cheaper than emulating memory-mapped I/O. (3) Defining the VM as a particular processor or family of processors makes the VM more portable than if it were a reflection of the actual hardware. For example, if a VM is defined to be a Pentium processor, the VM will work on a Pentium II or Pentium III processor. (4) VM performance suffers when sensitive instructions are forced to trap to emulation software. (5) There are alternatives to modifying the microcode support for every privileged instruction to meet the needs of the VMM. (6) If a VMM is a security kernel, dependencies between the VMM and VMs must be scrutinized.

The Alpha architecture is designed to support virtualization [2]. It is designed to contain no errors that would allow protection mechanisms to be bypassed. Even "UNPREDICTABLE" results or occurrences are constrained so that security and virtualization are supported. The processor may hold or loose information as a result of "UNDEFINED" operations; however, these operations can only be triggered by privileged software. Privileged Architecture Library code (PALcode) provides a non-microcoded interface for privileged instructions. All privileged instructions must be implemented in PALcode and may be processor-model specific. The Alpha architecture supports PALcode replacement, thus allowing per-OS code to yield high performance. A VMM on the Alpha would have PALcode for all supported operating systems.

Unlike the Alpha which explicitly forbids state data from registers to be spread, most processors permit leakage of information from unpredictable results. The multiple address spaces of the Intel x86 architecture family allows such leakage [35].

The observations in this section should be considered in any attempt to design a secure Type I VMM for the Intel Pentium architecture.

4.2 Pentium Security Support

Intel 80x86 processors provide support for well understood requirements of secure systems [32]. These include call gates, segmentation, several hardware privilege levels and privileged instructions [15]. The combination of segmentation and rings are particularly supportive of secure system design and implementation [34]. The processor family was the choice for past and present trusted systems: the Boeing MLS Lan (A1)[3] [23], Gemini Trusted Network Processor

[3]TCSEC evaluation classes are given in parentheses.

(A1)[27], Verdix VSLAN (B2) [26], TIS Trusted Xenix (B2)[25], and the XTS-300 (B3) [24].

4.3 Pentium Virtualization Techniques

Since the Intel Pentium architecture is not truly virtualizable, current VMMs for the hardware base [36] must use a bit of "trickery" to realize a VMM. Each method must detect sensitive but unprivileged instructions before they are executed by a VM.

4.3.1 Pure Emulation

Pure emulation allows one system architecture to be mapped into another system architecture. By modeling a large part of the x86 instruction set in software, emulation allows x86 operating systems and applications to run on non-x86 platforms [19]. The disadvantage of emulation is significant performance degradation; no instructions are ever executed directly on the hardware. The performance degradation in Java compilers bears witness to this observation. Advances in compiler technology can help, but without specialized machine support, performance will never achieve that of a Type I VMM on a comparable hardware base. Advanced techniques, such as dynamic translation, can improve performance. Dynamic translation allows sequences of small, x86 architecture code to be translated into native-CPU code "on-the-fly." Since the native code is cached or even optimized, it can run significantly faster. This is the approach used by Transmeta [18], which provides pure emulation and is a complete software interpreter machine (CSIM) [12]. The use of register shadowing and soft memory may permit support of VMM technology. It is worth pointing out that in such systems, the morphed code must be protected from tampering or leakage of secrets. It is not clear whether such security concerns are addressed in the current generation of binary translation systems.

4.3.2 OS/API Emulation

Applications normally communicate with an operating system with a set of APIs. OS/API emulation [20] involves intercepting and emulating the behavior of the APIs using mechanisms in the underlying operating system. The out-of-kernel OS emulation used for certain Mach architectures [29] might be considered a variant of this approach. This allows applications designed for other x86 operating systems to be run. This strategy is used in Wine which provides "an implementation of the Windows 3.x and Win32 API on top of X and Unix" [37]. Wine has a program loader that allows unmodified Windows 3.1/95/NT binary files[4] to run on Intel x86-based Unix machines, such as Linux, FreeBSD, and Solaris. It allows application binaries files to run natively and achieves better performance than the pure emulation technique described above. However, OS/API emulation only works on members of the x86 OS family for which the APIs have been emulated. Furthermore, OS/API emulation is very complex. A VMM is less complicated and requires fewer updates with each new release of the OS.

4.3.3 Virtualization

A third technique is virtualization. Most hardware is only designed to be driven by one device driver. The Intel Pentium CPU is not an exception to this rule. It is designed to be configured and used by only one operating system. Features and instructions of the processor designed for applications are generally not a problem for virtualization and can be executed directly by the processor. A majority of a processor's load comes from these types of instructions. However, as discussed above, certain sensitive instructions are not privileged in the Intel architecture, making it difficult for a VMM to detect when they are executed. A strategy for virtualizing the Intel architecture would be as follows:

- Non-sensitive, unprivileged application instructions can be executed directly on the processor with no VMM intervention.
- Sensitive, privileged instructions will be detected when they trap after being executed in user mode. The trap should be delivered to the VMM that will emulate the expected behavior of the instruction in software.
- Sensitive, unprivileged instructions must be detected so that control can be transferred to the VMM.

The hardest part of the virtualization strategy is handling the seventeen problem instructions described in Section 3. Lawton describes how this is accomplished for FreeMWare[20].[5] It analyzes instructions until one of the following conditions is encountered:

1. A problem instruction.
2. A branch instruction.

[4] MS-DOS, Windows 3.1, Windowds 95, Windows 98, and Windows NT 4.0 are all trademarks of the Microsoft Corporation. All other tradmarks, including Red Hat Linux, Caldera OpenLinux, SuSE Linux, FreeBSD, and Solaris are trademarks of their respective owners.

[5] As of March 23, 2000, FreeMWare is called Plex86.

3. The address of an instruction sequence that has already been parsed.

If 1 or 2 is encountered, a breakpoint must be set at the beginning of the problem or branch instruction. If 3 is encountered, execution continues normally since this code has been analyzed already and necessary breakpoints have been installed. The complexity of this approach may render a highly secure VMM unachievable.

Code is allowed to run natively on the processor until it reaches a breakpoint. If the breakpoint occurred because of a problem instruction, its behavior is emulated by the VMM. If the breakpoint occurred because of a branch instruction, it is necessary to single step through its execution and begin analyzing instructions again at the branch target address. If the target address is not computed and has already been analyzed and marked as safe, then the branch instruction can also be marked as safe and it can run natively on the processor on subsequent accesses. Computed branch addresses require special attention. These instructions must be dynamically monitored to ensure that execution does not branch to unanalyzed code. A table might be used to keep track of the breakpoints.

Some instructions may write into memory, possibly into the address of instructions that have already been analyzed and marked as safe. The paging system is used to prevent this by write protecting any page of memory in the page tables that has already been analyzed and marked as safe. All page entries that point to the physical page with analyzed code would have to be write protected since multiple linear addresses can be mapped to the same physical page. When a write-protect page fault occurs, the VMM can unprotect the page and step through the instructions. A breakpoint can be installed before any problematic instructions. Finally, the page should be write-protected again. Instructions that cross page boundaries involve tow write-protected pages. Tables are used to track previously analyzed instructions.

Also pass-through I/O devices, timing issues, and virtualizing descriptor loading must be addressed.

Pass-Through I/O Devices: It may be useful to allow a device driver in the guest OS to drive hardware for a device that is not supported by the host OS. For example, a Linux host OS will not support a Winmodem. Pass-through devices allow a guest OS to communicate with devices using a pass-through mechanism that handles I/O reads and writes. Because control of the real hardware is turned over to the VMOS, pass-through I/O devices render security problematic.

Timing: A VMM must accurately emulate system timers. Every time slice of native code execution is bounded by an exception generated by the system timer when the execution time slice is over. The exception vectors to a routine defined in the VMM's IDT for a guest OS. A mechanism is needed that measures the time between these exceptions to emulate an accurate timer. On Intel Pentium processors, performance monitoring could be used. The RDTSC, Read Time Stamp Counter, instruction gives an accurate time stamp reading. The instruction is also executable in CPL 3, allowing efficient use in user-level VMM code.

Virtualization of Descriptor Loading: For two reasons a Pentium-based VMM must have its own set of LDT, GDT, and IDT tables. First, it allows the segment register mechanisms to work naturally. Second, it allows the VMM to have its own set of exception handlers.

Since all privilege levels (0-3) in a VM are mapped into CPL 3, the CPL is not sufficient when trying to load code that is more privileged (i.e. numerically less) than CPL 3. CPL 3 code can load descriptors as expected as long as the GDTR and LDTR registers point to the guest OS's descriptor tables. When running system code in CPL 3, exceptions are generated when loading a descriptor with that has CPL < 3. This does not occur when system code is executed at CPL 0. To solve this problem, one must trap and emulate instructions that load the segment registers when running at CPL < 3. All instructions that examine segment registers with PL < 3 must be virtualized because they may look at the RPL field.

A private GDT and LDT for the virtualization of code at CPL < 3 can also help solve this problem. Since, the instructions that reference the GDTR and LDTR are emulated, they can be loaded with values that point to the private GDT and LDT. The private descriptor tables would start out empty and generate exceptions when a segment register loads. When this happens, a private descriptor is generated that allows the next segment register load to execute natively. Every time the GDTR and LDTR are reloaded, the private descriptor tables are cleared.

4.3.4 Other Virtualization Considerations

Disco is an implementation of a Type I VMM for the Flash multi-processor [8]. It runs several different commercial operating systems on virtual machines to provide high-performance system software. Some of the key insights of the Disco implementation applicable to virtualizing the Intel Pentium architecture are described below.

Virtual CPUs: Multiple VMMs are multiplexed onto a common physical processor by using virtual processors. A data structure is kept for each virtual CPU that contains register contents, TLB contents, and other state information of the virtual CPU when it is not running on the real CPU. The VMM is responsible for managing the virtual CPUs and ensures that the effects of traps are handled properly by the executing virtual processor.

Virtual Physical Memory: To virtualize physical memory, an extra level of address translation that maintains VM physical-to-machine address mappings is used. Virtual machines are given physical addresses that start at address zero and continue to the size of the VM's memory. These physical addresses could be mapped to machine addresses used by the Intel processor using the hardware-reloaded TLB of the Intel processor. The VMM protects and manages the page table. When the VMOS tries to insert a virtual-to-physical mapping in the TLB, the VMM emulates this by translating the physical address into the corresponding VM address and inserting this into the TLB.

Virtual I/O Devices: The VMM must intercept device accesses from virtual machines and forward them to physical devices. Instead of trying to use every device's real device driver, one special device driver for each type of device is used. Each device has a monitor call that is used to pass all command arguments to the VMM in a single trap. Many devices such as disks and network interfaces require direct memory access (DMA) to physical memory. Normally these device drivers use parameters that include a DMA map. The VMM must intercept these DMA requests and translate physical addresses into machine addresses.

We note that since the VMM must control devices, a VMM for the Intel Pentium architecture must be provide device drivers for each VMOS. Loadable drivers would be particularly convenient.

Virtual Network Interface: So that VMs can communicate with each other, they use standard distributed protocols such as NFS. Disco manages a virtual subnet that allows this communication. A copy-on-write strategy for transferring data between VMs reduces the amount of copying. Virtual devices use Ethernet addresses and do not limit the maximum transfer unit of packets.

4.4 Unmodified Pentiums: VMM Security Concerns

To be a high-assurance secure computing system, security policies are correctly enforced, even under hostile attack. Examples of such systems are at least TCSEC Class B2 or an equivalent level in the Common Criteria [1]. The systems' protection mechanisms must be structured and well-defined. When dealing with highly sensitive information, labels are needed to order information into equivalence classes. Also, for environments where users are also categorized into equivalence classes based on clearances or other ordering techniques, a very effective protection mechanism is needed.

Current VMMs for the Intel architecture do not meet these requirements although some vendors claim security as a feature [31]. One claims that their product can "isolate and protect each operating environment, and the applications and data that are running in it" [36]. Another claim is that the system does "not make any assumptions concerning the software that runs within the virtual machine. Even a rogue application or operating system is confined..." Given such claims, it is worthwhile to ask how well current VMMs can enforce the VM isolation needed to support a mandatory security policy. Note that this analysis is based on assumptions regarding how virtualization is being accomplished. The following sections describe some potential problems if such systems were to be used to separate mandatory security levels.

4.4.1 Resource Sharing

A problem results from resource sharing between virtual machines. If two virtual machines have access to a floppy drive, information can flow from one VM to the other. Files could be copied from one VM to the floppy, thus giving the other VM access to the files.

4.4.2 Networking and File Sharing

A similar problem results from support of networking and file sharing. Here two virtual machines at different security levels could communicate information. Exploitable mechanisms include Microsoft Networking, Samba, Novell Netware, Network File System, and TCP/IP. For example, using TCP/IP, a VM could FTP to either a host OS or guest Linux OS and transfer files.

4.4.3 Virtual Disks

The ability to use virtual disks is also a problem. A virtual disk is a single file that is created in the host OS and used to encapsulate an entire guest disk, including an operating system and its applications. Anyone with access to this file in the host operating system could copy all information in the virtual disk to external media. The attacker could then install the virtual machine monitor on his own system and open the copied virtual disk.

Another problem is that any host OS application with read access to the file containing the virtual disk can examine the contents of virtual disk. For example, host OS file utilities such as grep can be used to search for specific strings in the virtual file system. Our tests using a Linux host OS and a Windows NT guest OS showed that a sensitive string could be located by grep in seconds on an approximately 300 MB virtual disk.

Both problems could be remedied by restricting access to the virtual file. Yet, to achieve this with any measure of assurance, a secure host OS is required.

4.4.4 Program Utilities

Tools for virtual machine interoperation may cause problems. For example, after installing VMware-Tools [36] in a guest OS, the cursor can move freely between the host OS desk-top and those of the VMs. Another feature is the ability to cut and paste between virtual machines using a feature similar to the Windows clipboard. The potential security danger if virtual machines were running at different mandatory security levels is obvious.

4.4.5 Host Operating System

For a Type II VMM, many security vulnerabilities emerge due to the lack of assurance available in the underlying host operating system. Flaws in host OS design and implementation will render the virtual machine monitor and all virtual machines vulnerable.

4.4.6 Serial and Printer Ports

Implementation of serial and printer ports presents another security problem. Before starting a virtual machine, a configuration of the guest OS must be loaded or created. A configuration option for parallel and serial ports is to have output of all parallel/serial ports go to a file in the file system of host OS. Thus on the guest OS, user attempts to print will result in output to a host OS file. Users could easily transfer information so that others could read the printer file in the host OS if its permissions were not managed carefully.

4.5 Intel-Based VMM for High Security

We conclude that current VMMs for the Intel architecture should not be used to enforce critical security policies. Furthermore, it would be unwise to try to implement a high assurance virtual machine monitor as a Type II VMM hosted on a generic commercial operating system. Layering a highly secure VMM on top of an operating system that does not meet reference monitor criteria would not provide a high level of security.

Yet the Intel Pentium processor architecture has many features that can be used to implement highly secure systems. How can these be applied?

A better approach would be to build a Type I VMM as a microkernel. The secure microkernel could be very small, making it easier for the VMM to meet the reference monitor verifiability requirement. The use of minimization, rigorous engineering, and code correspondence contribute to ensuring that the implementation is free of intentional as well as accidental flaws.

The Type I VMM would provide virtual environments on the machine. It would intercept all attempts to handle low-level hardware functions from the VMs and would control all of the devices and system features of the CPU. The microkernel could allow each VM to choose among a specific set of virtual devices, which may or may not map directly to the real devices installed on the system.

There are two advantages to using a Type I VMM to separate mandatory security levels. First, a Type I VMM can provide a high degree of isolation between VMs. Second, existing popular commercial operating systems for the processor and their applications can be run in this highly secure environment without modification. A VMM eliminates the need to port software to a special secure platform and supports the functionality of current application suites.

The biggest disadvantage to a Type I approach is that device drivers must be written for every device. This is a problem because of the wide variety of peripheral types and models available. (Note that a less secure Type II VMM avoids this problem by using existing drivers written for the host OS.) This disadvantage can be overcome when developing a secure solution by only supporting certain types and manufacturers of devices. It is not out of the ordinary for highly secure solutions to require specific types of hardware.

Before trying to implement a secure Type I VMM for the Pentium, it might be advantageous to modify the chip. Two alternative modifications could make virtualization easier. First, all seventeen unprivileged, sensitive instructions of the Intel architecture could be changed to privileged instructions. All instructions would trap naturally and the VMM could emulate the behavior of the instruction. However, this solution may cause problems in current operating systems because these seventeen instructions would now trap.

An alternative is to implement a trap on op-code instruction [12]. A new instruction is added that allows an operating system to declare instructions that should be treated as if they were privileged. This makes virtualization easier without affecting current operating systems.

Other virtualization approaches require additional code to force sensitive, unprivileged instructions to be handled by VMM software. As a result, two security concerns arise. First, the security kernel may not be considered minimal because of the extra virtualization code. Second, virtualization of the unmodified processor requires checking every instruction before it executes. Such checking is likely to doom to failure creation of a high assurance VMM.

5 Conclusions and Future Work

The feasibility of implementing a secure virtual machine monitor on the Intel Pentium has been explored.

VMM types and their hardware requirements were reviewed. Then, a detailed study of the virtualizability of all 250 Pentium instructions was conducted to determine if the processor could meet the hardware requirements of any type of VMM. The analysis showed that seventeen instructions did not meet virtualization requirements because they were sensitive and unprivileged.

After defining a strategy to "virtualize" the Pentium architecture, an analysis was conducted to determine whether a Pentium-based secure virtual machine monitor is able to securely isolate classified from unclassified virtual machines could be built. We conclude that current VMM products for the Intel architecture should not be used as a secure virtual machine monitor.

The Intel Pentium processor family already has many features that support the implementation of highly secure systems. Slight modifications to the processor would significantly facilitate development of a highly secure Type I VMM.

An effort is currently underway to examine the Intel IA64 architecture to determine how its new relate to the construction of secure systems and virtualization. The possible use of virtualization techniques for processors supporting fast binary translation is also being explored.

Acknowledgements

The authors wish to acknowledge the insight, guidance and suggestions made by Steve Lipner as this research progressed. We are grateful to Dr. Paul Karger for careful review of our manuscript, suggestions and encouragement. We wish to thank James P. Anderson for unflagging encouragement of our work and Timothy Levin for insightful discussions and review of the paper. We are grateful to the Department of the Navy for its support of the Naval Postgraduate School Center for Information Studies and Research, which made this research possible.

References

[1] ISO/IEC 15408 - Common Criteria for Information Technology Security Evaluation. Technical Report CCIB-98-026, May 1998.

[2] Alpha Architecture Handkbook. Technical Report Order Number: ECQD2KC-TE, October 1998.

[3] E. R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the Opportunities of Binary Translation. *IEEE Computer*, 33(3):40–45, March 2000.

[4] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I,DITCAD-758206. Vol. II, DITCAD-772806).

[5] C. Attanasio, P. Markenstein, and R. J. Phillips. Penetrating an Operating System: a Study of VM/370 Integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[6] D. E. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corp., Bedford, MA, 1973.

[7] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, MITRE Corp., 1977.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scaleable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[9] B. Gold, R. Linde, R. J. Peller, M. Schaefer, J. Scheid, and P. D. Ward. A security retrofit fo vm/370. In R. E. Merwin, editor, *National Computer Conference*, volume 48, pages 335–344, New York, NY, June 1979. AFIPS.

[10] B. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in Retrospect. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 13–23, Oakland, CA, April 1984. IEEE Computer Society Press.

[11] B. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. Vm/370 security retrofit program. In *Proceedings 1977 Annual Conference*, pages 411–418, Seattle, WA, October 1977. A.C.M.

[12] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. Ph.D. thesis, Harvard University, Cambridge, MA, 1972.

[13] J. Hall and P. T. Robinson. Virtualizing the VAX Architecture. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 380–389, Toronto, Canada, May 1991.

[14] W.-M. Hu. Reducing Timing Channels with Fuzzy Time. In *Proceedings 1991 IEEE Symposium on Research in Security and Privacy*, pages 8–20. IEEE Computer Society Press, 1991.

[15] Intel. *Intel Architecture Software Devloper's Manual, Volume 1: Basic Architecture*. Intel Corporation, Santa Clara, CA, 1999.

[16] Intel. *Intel Architecture Software Devloper's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, Santa Clara, CA, 1999.

[17] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A Retrospective on the VAX VMM Security Kernel. *Transactions on Software Engineering*, 17(11):1147–1165, November 1991.

[18] A. Klaiber. The Technology Behind Crusoe^{TM} Processors. Transmeta Corporation, Santa Clara, CA, January 2000. also http://www.transmeta.com.

[19] K. Lawton. http://www.bochs.com, July 1999.

[20] K. Lawton. Running Multiple Operating Systems Concurrently on the IA32 PC Using Virtualization Techniques. http://www.freemware.org/research/paper.txt, June 1999.

[21] S. E. Madnick and J. J. Donavan. Application and Analysis of the Virtual Machine Approach to Information System Security. In *ACM SIGARCH-SYSOPS Workshop on Virtual Computer Systems*, pages 210–224, Boston, MA, March 1973. A.C.M.

[22] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

[23] National Computer Security Center. *Final Evaluation Report: Boeing Space and Defense Group, MLS LAN Secure Network Server System*, 28 August 1991.

[24] National Computer Security Center. *Final Evaluation Report of HFSI XTS-200*, CSC-EPL-92/003 C-Evaluation No. 21-92, 27 May 1992.

[25] National Computer Security Center. *Final Evaluation Report: Trusted Information Systems, Inc. Trusted XENIX Version 4.0*, January 1994.

[26] National Computer Security Center. *Final Evaluation Report: Verdix Corporation VSLAN 5.1/VSLANE 5.1*, 11 January 1994.

[27] National Computer Security Center. *Final Evaluation Report of Gemini Computers, Incorporated Gemini Trusted Network Processor, Version 1.01*, 28 June 1995.

[28] G. Popek and R. Goldberg. Formal Requirements for Virtualizable 3rd Generation Architectures. *Communications of the A.C.M.*, 17(7):412–421, 1974.

[29] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Society International Conference COMPCON 89*, San Francisco, CA, February 1989. IEEE Computer Society Press.

[30] J. S. Robin. Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor. Master's thesis, Naval Postgraduate School, Monterey, CA, September 1999.

[31] M. Rosenblum. Lecture at Stanford University. 17 August 1999.

[32] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[33] M. Schaefer and B. Gold. Program Confinement in KVM/370. In *Proceedings 1977 Annual Conference*, pages 404–410, Seattle, WA, October 1977. A.C.M.

[34] M. D. Schroeder and J. H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Comm. A.C.M.*, 15(3):157–170, 1972.

[35] O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 211–222, Oakland, CA, May 1995. IEEE Computer Society Press.

[36] VMware Inc. *Welcome to VMware, Inc – Virtual Platform Technology*, March 1999. http://www.vmware.com/standards/index.html.

[37] Wine. http://www.winehq.com, June 2000.