VT und Pacifica

Vortrag zum Seminar „Virtuelle Maschinen und Emulatoren“

Alexander Würstlein
arw@arw.name

11. September 2007

Inhalt

Inhaltsverzeichnis

1 Virtualisierbarkeit des Pentium

1.1 Einleitung

Die weit verbreitete i386[1]-Architektur, auf der auch der Pentium-Prozessor und dessen Nachfolger und Nachbauten aufbauen, ist nicht sicher virtualisierbar.

Um dennoch virtualisierte Umgebungen auf der x86-Architektur bereitstellen zu können wurden bisher verschiedene Softwareverfahren eingesetzt, die in späteren Vorträgen erleutert werden. Da diese aber oft um eine annehmbare Performance zu erzielen detailierte Kenntnis der Gastbetriebssysteme voraussetzen haben die Prozessorhersteller AMD und Intel jeweils unabhängig voneinander Befehlssatzerweiterungen entwickelt, die die vorgenannten Probleme beseitigen und die x86-Architektur virtualisierbar machen sollen.

[1] synonym: x86-, Pentium-Architektur

Intel nennt diese Erweiterung „VT-x“ und AMD „Secure Virtual Machine“ (SVM), die Techniken wurden jedoch im Laufe der Entwicklung teilweise mehrmals umbenannt, es sind verwirrenderweise auch die Bezeichnungen „VMX“, „Intel VT“ und „Vanderpool Technology“ von Intel und „Pacifica“ und „AMD-V“ von AMD gebräuchlich.

Ich werde im Folgenden zunächst am Beispiel des Pentium die Probleme der i386-Architektur aufzeigen[2] und anschliessend deren Lösung durch die genannten Befehlssatzerweiterungen vorstellen. Soweit ich Unterschiede nicht ausdrücklich erwähne beziehen sich alle Aussagen gleichermassen auf die Produkte beider Hersteller.

1.2 Voraussetzungen

Allgemeine Voraussetzung für die Virtualisierbarkeit einer Prozessorarchitektur ist nach Popek und Goldberg[3], daß alle sensitiven Instruktionen eine Untermenge der privilegierten Instruktionen sein müssen. Privilegierte Instruktionen sind Instruktionen, die im Usermodus des jeweiligen Prozessors immer einen Trap auslösen. Sensitive Instruktionen sind Instruktionen, die für die Virtualisierung problematische Auswirkungen haben, im Sinne daß entweder Schutzmechanismen umgangen werden oder sich die virtuelle Maschine erkennbar anders verhält als eine reale Maschine.

Dementsprechend gibt es zwei Klassen sensitiver Instruktionen, kontrollsensitive, die den Prozessormodus oder den Speicherschutz ändern und verhaltenssensitive die verschiedenes Verhalten abhängig vom Prozessormodus oder Speicherschutz zeigen. Beides möchte man durch Traps abfangen können um die Schutzmechanismen und die Trennung der virtuellen Maschine von der realen aufrecht zu erhalten beziehungsweise das Verhalten der virtuellen Maschine für die darin ablaufenden Programme ununterscheidbar von realer Hardware zu machen.

Ausserdem ist es notwendig, daß die privilegierte und die nichtprivilegierte Form einer Instruktion dieselbe Syntax haben, denn ein Gastsystem wird immer die privilegierte Form zu benutzen versuchen. Weiterhin muss die Architektur natürlich die oben erwähnten Schutzmechanismen wie Speicherschutz zur Verfügung stellen.

1.3 Probleme

In der für VT-x massgeblichen x86-Architektur (hier am Beispiel des Pentium) ergibt sich folgende Situation: Zwar ist die Syntax privilegierter und unprivilegierter Instruktionen gleich und die Architektur bietet Speicherschutz an, doch die erstgenannte Voraussetzung ist nicht erfüllt: Es existieren mehr als 17 problematische[3] Operationen, die zwar sensitiv aber nicht privilegiert sind. Im Folgenden möchte ich näher auf diese Instruktionen eingehen.

Man kann die sensitiven Instruktionen nach ihrer Funktion in 4 Gruppen einteilen, jeweils mit den entsprechenden Opcodes:

1. ändern den Maschinenzustand
   keine

2. lesen oder ändern problematische Register oder Speicherstellen
   `SGDT, SIDT, SLDT, SMSW, PUSHF, POPF`

[2] nach [1]

[3] die Anzahl variiert, je nachdem ob man Varianten wie bedingte Sprünge mitzählt

3. betreffen den Speicherschutz oder erlauben Zugriff auf eigentlich gesperrte Speicherbereiche
   `LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT, RET, STR, MOVE`

4. I/O
   keine

Im einzelnen möchte ich nun die Funktion dieser Opcodes zusammen mit den auftretenden Problemen erörtern:

`SGDT, SIDT` und `SLDT`[4] lesen das jeweilige CPU-Register[5], das einen Zeiger auf die jeweilige Tabelle enthält und schreiben diesen Zeiger in das als Operand angegebene Register oder die angegebene Speicherstelle. Diese drei Instruktionen sind auch im Usermodus ausführbar und verhalten sich genau wie in den entsprechenden privilegierten Modi, ein Benutzerprogramm kann somit die Adressen der entsprechenden Tabellen erhalten. Jedoch wird jedes Betriebssystem über Speicherschutzmechanismen Benutzerprogrammen den Zugriff auf diese Adressen verwehren, genau dies ist auch das Problem das bei diesen Instruktionen eine Virtualisierung verhindert: ein Gastbetriebssystem wird erwarten die entsprechenden Tabellen lesen und schreiben zu dürfen. Durch das Fehlen eines Traps an dieser Stelle wird dem Hypervisor nicht ermöglicht dem Gastsystem „falsche“ Tabellen unterzuschieben.

Ringmodell beim Pentium

Zum besseren Verständnis der weiteren Ausführungen ist es notwendig kurz das Ringmodell des Pentium vorzustellen.

Die verschiedenen Ringe oder „privilege level“[6] werden von 0 bis 3 nummeriert. Hierbei stellt Ring 0 den höchstberechtigten Ring dar, Ring 3 den am niedrigsten berechtigten. Die Bits 0 und 1 im `CS`- und `SS`-Register zeigen den aktuellen Ring (`CPL`[7]) an.

Die Ringe 0 bis 2 sind privilegiert, dürfen also privilegierte Anweisungen ausführen. In den `IOPL`-Bits im `EFLAGS`-Register wird festgelegt, welche Ringe auf den I/0-Adressraum zugreifen dürfen, der Zugriff wird erlaubt wenn $\texttt{IOPL} \geq \texttt{CPL}$. Ähnliches gilt für Segmente über das descriptor privilege level `DPL` und das request privilege level `RPL`. Jedoch sind hier die Kriterien wann ein Zugriff gewährt wird etwas komplizierter[8].

Benutzersoftware wird in diesem Modell in Ring 3 laufen, der Betriebssystemkern in Ring 0. Die Ringe 1 und 2 sind für Betriebssystemkomponenten wie zum Beispiel Gerätetreiber oder Betriebssystemprozesse in Mikrokernelsystemen vorgesehen[9].

[4] store global / local / interrupt descriptor table, enthalten Segmentdeskriptoren (global und local) und Interruptdeskriptoren

[5] `GDTR, LDTR, IDTR`

[6] abgekürzt „PL“

[7] current privilege level

[8] näheres in [7] Kapitel 4.5

[9] Das einzige dem Autor bekannte Betriebssystem, dass die Ringe 1 und 2 benutzt ist OS/2, Minix benutzt zumindest Ring 1. Die Benutzung von mehr als Ring 0 und 3 stellt ein Portabilitätsproblem dar, denn die meisten anderen Architekturen besitzen nur 2 Ringe.

PUSHF, POPF und SMSW schreiben und lesen das EFLAGS-Register beziehungsweise das Maschinenstatuswort. Diese Register enthalten beispielsweise das „interrupt enable“-bit, das I/O privilege level und den Prozessormodus.

Ein Gastbetriebssystem wird versuchen Werte mittels POPF ins EFLAGS-Register zu schreiben, entweder um darin Werte zu setzen oder um nach einem Systemaufruf den Zustand wiederherzustellen. Im unprivilegierten Modus werden diese Änderungen stillschweigend ignoriert was sowohl die Funktion des Gastsystems beeinträchtigt als auch dem Hypervisor einen Eingriff unmöglich macht.

Jeder Prozess kann diese Werte lesen ohne einen Trap auszulösen. Hierdurch kann ein Gastsystem feststellen, dass es virtualisiert ist, da die vorgenommenen Änderungen wie erwähnt ignoriert wurden.

SMSW wird verwendet um auf 80386 und neueren Prozessoren Eigenschaften älterer (80286) Prozessoren zu emulieren. Auch hier kann das Gastsystem, ähnlich wie im Fall von EFLAGS am Unterschied zwischen dem erwarteten und tatsächlichen Wert feststellen, dass es in einer virtuellen Umgebung läuft.

LAR[10]**, LSL**[11]**, VERR**[12] **und VERW**[13] liefern Informationen (Zugriffsrechte, Grenzen, Les- und Schreibbarkeit) aus Segmentdeskriptoren. Zwar werden die Berechtigungen zum Zugriff auf den jeweiligen Segmentdeskriptor geprüft, bei unberechtigtem Zugriff wird aber kein Trap ausgelöst sondern nur das zero-flag gesetzt. Warum dies geschieht ist dem Autor nicht bekannt.

CALL, JMP, INT und RET sind Anweisungen, die mit Funktionsaufrufen und der Rückkehr aus Funktionen im Zusammenhang stehen.

CALL wird für normale Funktionsaufrufe innerhalb desselben oder in einen anderen Ring verwendet sowie für Prozesswechsel. Ähnliches gilt für JMP[14], was im Unterschied zu CALL keine Vorkehrungen für einen Rücksprung trifft. INT entspricht CALL, jedoch erfolgt der Aufruf indrekt über die Adresse der aufzurufenden Funktion anhand des angegebenen Index aus der Interruptdeskriptortabelle[15].

In allen diesen Fällen stellt ein Aufruf in einen anderen Ring ein Problem dar, da hier der Aufruf implizit den Stack wechselt. Ein Gastbetriebssystem wird sich auf diesen Mechanismus verlassen, was natürlich zu Problemen führt wenn es, ebenso wie seine Anwendungen, in Ring 3 läuft. Hierbei würde der Stack nicht gewechselt, neben dem offensichtlichen Problem, dass die Virtualisierung feststellbar wird wird auch die Isolation von Betriebssystem und Programmen emfindlich verletzt.

RET[16] ist die Umkehrung von CALL, es hat also auch das Problem, dass eine Rückkehr vom Gastbetriebssystem in eine Anwendung den Stack wechseln müsste aber nicht wird. Zusätzlich werden bei der Rückkehr bestimmte Segmentregister geleert, wenn die Berechtigungen des Prozesses zu dem zurückgekehrt wird nicht ausreichen. Auch dies wird nicht passieren wenn das Gastbetriebssystem ebenso wie seine Anwendungen in Ring 3 laufen, mit denselben Problemen wie beim Stack.

[10] Load Access Rights
[11] Load Segment Limit
[12] VERify Readable
[13] VERify Writable
[14] JMP steht hier auch für alle bedingten Sprünge wie zum Beispiel JNZ (Jump if Not Zero)
[15] IDT
[16] und IRET, für eine Rückkehr aus Interrupthandlern

PUSH, POP und MOV sind die vermutlich am meisten benutzten Anweisungen in dieser Aufzählung. Ihr Zweck ist das Lesen und Schreiben von Werten in Register, an Speicherstellen und auf den Stack. Problematisch ist dabei vor allem, dass beim Auslesen der Segmentregister das `CPL` als Teil des Registerinhalts dem Gastsystem bekannt wird, wodurch es die virtualisierte Umgebung entdecken kann. Ausserdem ist das Verhalten bei unterschiedlichem `CPL` leicht anders, was ebenfalls die Virtualisierung „verraten“ kann.

2 Vorgenommene Erweiterungen

2.1 Prinzip

Wie in der Einleitung erwähnt haben als Antwort auf die vorgestellten Probleme die Hersteller AMD und Intel die Technologien SVM und VT-x entwickelt.

Die Technologien beider Hersteller haben dieselbe Funktionsweise gemein, unterscheiden sich aber in der Syntax.

Beide unterscheiden zusätzlich zu den oben beschriebenen Privilegierungsstufen (Ringen) einen „root“ bzw. „non-root“-Modus. Hierbei läuft der Hypervisor im „root“, das Gastsystem auf seinem virtuellen Prozessor im „non-root“-Modus. Der virtuelle Prozessor verhält sich hierbei genau wie ein nichtvirtueller Prozessor derselben Serie, es wurden also keine Änderungen an den oben beschriebenen Operationen direkt vorgenommen.

2.2 Benutzung

Beide Technologien beschreiben virtuelle Prozessoren bzw. Umgebungen durch Kontrollstrukturen, die „VM control block“ (VMCB, AMD) oder „VM control structure“ (VMCS, Intel) genannt werden. Diese leicht abweichenden Bezeichnungsweisen finden sich auch weiterhin bei den Instruktionen.

Um den Prozessor in den neu eingeführten Prozessormodus zu schalten benutzt man bei Intel `VMXON`, zum Ausschalten `VMXOFF`, bei AMD funktioniert entsprechendes mit dem Bit `EVER.SVME`. Bevor eine VM gestartet werden kann muss zuerst die entsprechende Kontrollstruktur aufgesetzt werden, dazu gibt es die Instruktionen `VMLOAD/SAVE` (AMD) und `VMPTRLD/ST, VMCLEAR/READ/WRITE` (Intel).

Diese Kontrollstrukturen enthalten Informationen ueber die Speicherbereiche, die der VM zugeordnet sind, die Registerinhalte der VM, wenn sich diese gerade nicht in Ausführung befindet und eine Reihe von Parametern und Flags, die das Verhalten der VM steuern.

Nachdem eine VM mit `VMLAUCH` (Intel) oder `VMRUN` (AMD) gestartet wurde wird der Prozessor unprivilegierten Code wie gewohnt ausführen. Bei den oben genannten kritischen Instruktionen, bei Interrupts oder Traps erfolgt eine Rückkehr zum Hypervisor (AMD: Intercept, Intel: VMEXIT). Welche Instruktionen oder Ereignisse eine solche Rückkehr auslösen ist in der Kontrollstruktur der VM einstellbar.

Nachdem vom Hypervisor die entsprechend vorgesehene Behandlung durchgeführt wurde kann mit `VMRUN` (AMD) oder `VMRESUME` (Intel) die Ausführung der VM fortgesetzt werden, dazu wird der Prozessorzustand der zuvor in die Kontrollstruktur gesichert worden war wiederhergestellt. In der Kontrollstruktur können ebenfalls vom Hypervisor virtuelle Interrupts oder Traps durch Flags ausgelöst werden, die dann das

Gastbetriebssystem behandelt. Dadurch kann die VM mit virtueller und realer Hardware kommunizieren.

Mit `VMCALL` oder `VMMCALL` (Intel bzw. AMD) ist auch ein expliziter Rücksprung aus der VM möglich, vergleichbar mit einem Syscall. Vorstellbare Verwendungsmöglichkeiten hierfür wären paravirtualisierte[17] Hardware oder ein verbessertes Scheduling, bei dem ein aktuell unbeschäftigtes Gastsystem explizit seine aktuelle Zeitscheibe über einen solchen Aufruf aufgibt. Es sind dem Autor jedoch keine Implementierungen hierzu bekannt.

2.3 I/O

Wie bereits erwähnt kann eine VM Ein- und Ausgabe über Interrupts und natürlich auch zugeordnete Speicherbereiche durchführen. Hierbei können auch Interrupts von realer Hardware, die zuerst im Hypervisor behandelt werden weitergegeben werden (evtl. auch gefiltert). Der Zugriff auf Speicherbereiche (DMA), die Betriebssystem und Hardware zur Kommunikation nutzen ist jedoch sehr problematisch.

Zwar wird der Zugriff des Gastsystems auf fremde Speicherbereiche durch die MMU verhindert, die Hardware unterliegt jedoch keinen solchen Einschränkungen und hat unkontrollierten Zugriff auf den gesamten physikalischen Speicher. Ein Gastsystem, dem erlaubt wird mit realer Hardware zu kommunizieren, kann also den Speicherschutz umgehen, indem es ein geeignetes Gerät anweist, fremde Speicherbereiche zu lesen oder zu schreiben.

Die mögliche Schadwirkung wird zwar vom jeweiligen Gerät abhängen, ein direktes Auslesen fremder Speicherbereiche durch eine VM könnte schwierig sein, aber ein möglicher Umweg wäre beispielsweise das Verschicken fremder Speicherbereiche durch die Netzwerkkarte an sich selbst oder Dritte.

Da es unter diesen Umständen nicht vernünftig ist virtuellen Maschinen reale Hardware zuzuordnen läuft die aktuelle Entwicklung in diesem Bereich auf die Einführung einer sog. „IOMMU“ hinaus.

Diese übersetzt analog zu einer MMU Adressen auf dem PCI-Bus in physikalische Adressem im RAM. Damit ist eine Beschränkung des Zugriffs einzelner Geräte auf ausgewählte Speicherbereiche möglich, die oben beschriebenen Verletzungen des Speicherschutzes durch DMA werden verhindert.

Zusätzlich schliesst eine IOMMU bereits länger bekannte Schwachstellen wie Firewire, wo über eine Firewire-Verbindung von einem anderen Rechner per DMA der komplette physikalische Speicher eines Rechners zugänglich ist.

Entsprechende Technologien sind ausserhalb des x86-Marktes schon seit längerem erhältlich unter dem Namen DVMA bei Sun und TCE bei IBM. AMD und Intel planen eine Einführung fuer 2008.

3 Ausblick

3.1 Linux KVM und andere

Der Linux-Kernel enthält seit Version 2.6.20 Unterstützung für SVM und VT, welche mit einem modifizierten Qemu namens KVM (Kernel Virtual Machine) die CPU-

[17] minimale Gerätetreiber im Gastsystem, die ausschliesslich für Gebrauch in virtuellen Umgebungen gedacht sind, hierbei kann auf eine virtuelle Hardwareschicht und deren Ansteuerung verzichtet werden was die Performance stark verbessert

Erweiterungen zur Beschleunigung von Qemu nutzbar macht. Die virtuelle Maschine ist dabei ein Userspace-Prozess, der über `/dev/kvm` mit dem Kernel kommuniziert und die virtuelle Hardware die Qemu bereitstellt nutzt.

Xen benutzt VT oder SVM nur für Betriebssysteme, für die keine Treiber zur Paravirtualisierung verfügbar sind, also üblicherweise Windows. Für Mac OS X ist Parallels besonders beliebt, dieses läuft ausschliesslich auf Rechnern mit Intel VT und bietet einen beeindruckenden Funktionsumfang wie beispielsweise die Einblendung von Fenstern von Windows-Anwendungen in der MacOS-Oberfläche und Interaktion wie Drag&Drop mit diesen Fenstern.

VMWare und VirtualBox benutzen laut eigener Aussage die CPU-Erweiterungen nur in Ausnahmefällen, im allgemeinen sei deren bisheriges Verfahren (siehe spätere Vorträge) schneller.

3.2 Virtualisierung anderer Architekturen

Andere Hersteller als Intel und AMD arbeiten natürlich ebenfalls im Zuge der allgemeinen Aufregung um Virtualisierung an entsprechenden Produkten oder bieten diese bereits seit längerem an.

Die Firma Sun stellt Prozessoren mit der Sparc-Architektur her. In der neuesten Auflage der Architektur mit dem Ultrasparc T1 und neuerdings T2 ist der Prozessor, den das Betriebssystem „sieht“, nur noch ein virtueller Prozessor der Architektur „sun4v“. Als Hypervisor dient die Firmware, die in einem neuen, „hyperpriviledged“ genannten Modus läuft.

Hierdurch können Systemresourcen wie Prozessorkerne, Speicher und in Verbindung mit der vorgenannten IOMMU auch PCI-Geräte den virtuellen Systemen fest zugeordnet werden (Partitionierung).

Bereits seit sehr langer Zeit baut IBM Rechner mit hardwareunterstützter Virtualisierung, aktuell in der neuesten Prozessorgeneration Power 4 und 5 unter der Bezeichnung „Logical Partitions“.

3.3 Zusammenfassung

Nach den Kriterien von Popek und Goldberg ist die Pentium-Architektur nicht virtualisierbar. Die Einführung eines neuen Betriebsmodus löst dieses Problem ohne gleichzeitig inkompatible Änderungen an der Architektur vornehmen zu müssen. Die Firmen Intel und AMD haben vom Funktionsumfang äquivalente aber nicht direkt kompatible Befehlssatzerweiterungen zu diesem Zweck unter den Bezeichnungen „IVT“ und „SVM“ geschaffen. Für die nahe Zukunft geplante Techniken wie IOMMU werden auch die Zuordnung physikalischer Geräte zu VMs sicher ermöglichen.

Literatur

[1] J. Robin, C. Irvine *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. *Proceedings of the 9th USENIX Security Symposium*, 2000, The USENIX Association, Denver. `http://citeseer.ist.psu.edu/robin00analysis.html`

[2] R. Uhlig, G. Neiger et. al. *Intel Virtualization Technology*. *Computer*, May 2005, IEEE Computer Society, New York.

[3] J. Schlumberger *Virtualisierungstheorie*, Vortrag im Hauptseminar Konzepte und Techniken virtueller Maschinen und Emulatoren. *Friedrich-Alexander-Universität*, 24. Mai 2007, Erlangen. http://www3.informatik.uni-erlangen.de/Lehre/virME/SS2007/

[4] S. Gora *Bittere Pille* – Schadsoftware beutet Hardwarevirtualisierung aus. *iX*, 4/2007, Heise Zeitschriften Verlag, Hannover.

[5] N.N. *UltraSPARC Architecture 2005. Draft D0.8.8*, 15 Jun 2006, Sun Microsystems, Santa Clara. http://opensparc-t1.sunsource.net/index.html

[6] N.N. *AMD64 Technology* – AMD64 Architecture Programmer's Manual Vol. 1-5. Sep 2006, Advanced Micro Devices. http://www.amd.com/de-de/Processors/TechnicalResources/0,,30_182_739_7044,00.html

[7] N.N. *Intel 64 and IA-32 Architectures Software Developer's Manual* Vol. 1-5. May 2007, Intel. http://developer.intel.com/products/processor/manuals/index.htm

[8] N.N. *IOMMU*. Wikipedia, englisch, 2007-05-25. http://en.wikipedia.org/wiki/IOMMU

[9] N.N. *Pacifica (virtual machine)*. Wikipedia, englisch, 2007-05-25. http://en.wikipedia.org/wiki/Pacifica_%28virtual_machine%29

[10] N.N. *AMD I/O Virtualization Technology (IOMMU) Specification*. Feb 2007, Advanced Micro Devices. http://www.amd.com/de-de/Processors/TechnicalResources/0,,30_182_739_7044,00.html