Fachhochschule Münster

Fachbereich Elektrotechnik und Informatik

Bachelorarbeit

zur Erlangung

des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Informatik

HyperFuzz: Entwicklung einer Schnittstelle zum Fuzzen von Hypervisoren

Erstprüfer	Prof. Dr.-Ing. Sebastian Schinzel
Zweitprüfer	Hendrik Schwartke
vorgelegt am	19. August 2016
von	Fabian Ising

Eidestattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Steinfurt, 19. August 2016, _____

Fabian Ising

Abstract

Virtualisierung ist ein wichtiges Feld in der modernen IT-Umgebung, insbesondere im Bereich des Cloud Computing. In der Vergangenheit wurden immer wieder Schwachstellen in Virtualisierungsumgebungen gefunden und ausgenutzt. Hat ein Angreifer volle Kontrolle über eine virtuelle Maschine, so kann er solche Fehler ausnutzen und so die Virtualisierungsumgebung kompromittieren. Ein generischer, automatisierbarer Ansatz, um solche Schwachstellen zu finden wurde jedoch bisher nicht vorgestellt. Das in dieser Bachelorarbeit entwickelte und beschriebene Projekt bietet eine generische Schnittstelle um beliebige Virtualisierungsumgebungen und emulierte Hardware durch einen Fuzzing-Ansatz zu testen. Das Projekt nutzt Nested Virtualization, um eine Userspace-Schnittstelle anzubieten. Über diese kann die Kommunikation zwischen einer virtuellen Maschine und ihrer emulierten Hardware beobachtet und manipuliert werden.

Das entwickelte Projekt ist in der Lage eine Vielzahl von Crashes des inneren Gastes durch die Manipulation der Hardwarekommunikation zu verursachen. Weiterhin wurden verschiedenste Fehlermeldungen der Virtualisierungsumgebung verursacht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

CR3 Control Register 3, Prozessor Register, das die Adresse des Page Directory und der Page Tables angibt.

Gast Virtuelle Maschine, die auf dem Host ausgeführt wird.

Host Physische Hardware, auf der die Virtualisierung ausgeführt wird.
Hypervisor Software die virtuelle Maschinen erstellt und ausführt.

Innerer Gast Innerste virtuelle Maschine, die vom äußeren Gast ausgeführt wird.

Virtual Machine Monitor Siehe Hypervisor.

Äußerer Gast Äußerste virtuelle Maschine, die den inneren Gast ausführt.

Abkürzungsverzeichnis

DMA Direct Memory Access.

EPT Extended Page Tables.

GFN Guest Frame Number.
GPA Guest physical address.
GVA Guest virtual address.

HPA Host physical address.
HVA Host virtual address.

IRQ Interrupt Request.

KVM Kernel-based Virtual Machine.

MMIO Memory Mapped I/O.
MMU Memory Management Unit.
MTF Monitor Trap Flag.

NGPA Nested guest physical address.
NGVA Nested guest virtual address.

QEMU Quick Emulator.

SPTE Shadow Page Table Entry.

TLB Translation Lookaside Buffer.

vCPU Virtual CPU.
VMCS Virtual Machine Control Structure.
VMM Virtual Machine Monitor.
VMX Virtual Machine Extensions.

1 Einleitung

Virtualisierung ist in der heutigen Zeit eines der wichtigsten Mittel, um sowohl Rechenleistung auf großen Servern auf mehrere Anwendungen oder virtuelle Hosts zu verteilen als auch um Anwendungen voneinander zu trennen und sie auf anderen Plattformen ausführbar zu machen.

Dabei wird Virtualisierung häufig nicht nur für den eigenen Bedarf oder zur Verwendung durch vollkommen vertrauenswürdige Nutzer freigegeben. Anbieter von virtuellen Servern bieten ihren Kunden zum Beispiel häufig mehr oder weniger vollständigen Zugriff auf eine virtuelle Maschine. Damit trennen sie diese sowohl von der realen Hardware als auch von den anderen virtuellen Maschinen auf der gleichen Hardware.

Ein weiteres wichtiges Feld, das Virtualisierung nutzt, ist das Cloud Computing. Auch hier wird häufig Nutzern voller Zugriff auf virtuelle Maschinen gegeben. Kann hier ein Nutzer Kontrolle über den Host oder andere virtuelle Maschinen erlangen, ist die Cloud-Umgebung kompromittiert.

Ein Angreifer der vollen Zugriff auf eine virtuelle Maschine hat, kann mit der virtuellen Hardware frei kommunizieren. Deshalb muss er sich nicht an die Beschränkungen halten, die etwaige Treiber ihm auferlegen. Ports und Memory Mapped I/O (MMIO) Speicher können direkt angesprochen werden, ohne dass eine Überprüfung stattfinden muss. Deshalb ist es von äußerster Wichtigkeit, dass die von der virtuellen Hardware angebotenen Schnittstellen robust und gegen Manipulation und vor Missbrauch geschützt sind.

Virtualisierung ist zumeist jeweils zu Teilen im User-[1] und im Kernelspace[2] implementiert. Dadurch besteht durch Fehler in der Implementierung die Gefahr sowohl den Userpace als auch den Kernelspace des Hostes zu kompromittieren. Die Kompromittierung des Kernelspace würde einer Übernahme des gesamten Systems gleichkommen.

Die Gefahren durch Angriffe auf die Virtualisierungsebene sind somit zahlreich und ein aktuelles sowie relevantes Problem.

1.1 Motivation

In der Vergangenheit kam es immer wieder zu Angriffen auf den Host von virtuellen Maschinen. Diese ermöglichten unter anderem über Fehler in der emulierten Hardware die Ausführung von beliebigem Code auf der Hostmaschine.[3] Auch der Angriff von einer virtuellen Maschine auf eine andere virtuelle Maschine auf dem gleichen Host über die Virtualisierungsebene wurde bereits mehrmals realisiert.[4]

In der Vergangenheit wurde keine Lösung vorgestellt, die die Suche nach Fehlern in verschiedenen Hypervisoren automatisiert. Diese Arbeit beschreibt die Entwicklung eines Tools, das diese

[1] Im Fall QEMU/KVM ist dies QEMU.
[2] Im Fall QEMU/KVM ist dies KVM.
[3] Siehe zum Beispiel CVE-2016-3710.
[4] Siehe zum Beispiel CVE-2015-3456.

Lücke schließt. Es wird ein Fuzzing-Ansatz genutzt, um Fehler in Hypervisoren, insbesondere in KVM, und in emulierter Hardware, insbesondere von QEMU, automatisiert zu finden.

Ziel dieser Arbeit ist es, eine Lösung zu entwickeln, die verschiedene Hypervisoren testen kann. Hierbei soll die Lösung unabhängig davon sein, wie die Implementierung dieser Hypervisoren konkret aussieht.

1.2 Bedrohungsszenario

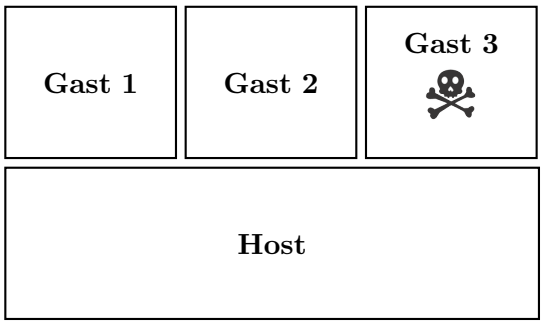


Abbildung 1: Bedrohungsszenario[5]

In Abbildung 1 ist zu sehen, wie sich das Bedrohungsszenario bei Fehlern in Virtualisierungsumgebungen darstellt. Ein Angreifer besitzt die vollständige Kontrolle über eine virtuelle Maschine auf dem Host. Auf diesem Host laufen beliebig viele weitere virtuelle Maschinen. Die virtuellen Maschinen sollten durch die Virtualisierungsumgebung komplett voneinander getrennt sein. Über Fehler in der Virtualisierungsumgebung erlangt der Angreifer jedoch Kontrolle über andere virtuelle Maschinen auf dem gleichen Host oder über den Host selbst.

1.3 Lösungsansatz

Um die hypervisorunabhängige Fuzzing-Lösung zu entwickeln, wird die sogenannte Nested Virtualization genutzt. Es wird eine virtuelle Maschine in einer anderen virtuellen Maschine ausgeführt. Dabei wird die Kommunikation zwischen der inneren und der äußeren virtuellen Maschine vom Host überwacht und manipuliert. Auf diese Art wird versucht, Fehler in der Virtualisierungsumgebung zu finden, die der äußere Gast bereitstellt.

Hierbei wird jegliche Kommunikation zwischen dem Hypervisor und seinem Gast beobachtet sowie manipuliert. Dies beinhaltet Portzugriffe, Memory Mapped I/O (MMIO) und Direct Memory Access (DMA).

[5]Totenkopf Quelle: `https://www.iconfinder.com/icons/192531/crossbones_skull_spooky_icon`

1.4 Verwandte Arbeiten

Bereits vor dieser Arbeit gab es Ansätze, Hypervisoren mittels Nested Virtualization auf Schwachstellen zu prüfen. Eine Veröffentlichung zu diesem Thema stellt das von Felix Wilhelm 2015 veröffentlichte *XenPwn* (vgl. [19]) dar. Diese beschreibt, wie im Hypervisor XEN über das Tracing von Speicherzugriffen Schwachstellen gefunden werden können. Der wesentliche Unterschied zu dieser Arbeit besteht darin, dass eine Beschränkung auf einen speziellen Hypervisor (XEN) stattfindet und nur Speicherzugriffe betrachtet werden.

Weiterhin gab es einige Ansätze, Hypervisoren direkt zu Fuzzen, wie zum Beispiel das *XenFuzz* Projekt (vgl. [1]). Diese spezialisieren sich zumeist auf einen bestimmten Hypervisor, in diesem Fall XEN. Hierbei wird zudem nicht die Kommunikation zwischen Hypervisor und virtueller Maschine gefuzzt, sondern die Nutzung von Hypercalls, also Aufrufe, die es dem Gast erlauben direkt mit dem Hypervisor zu kommunizieren.

Ein Projekt, das die Kommunikation zwischen emulierter Hardware und virtueller Maschine manipuliert, ist ein 2016 auf der *Hack in the Box Conference Amsterdam* vorgestelltes Fuzzing Framework des *360 Marvel Team* (vgl. [17]). Dieses ist in der Lage verschiedene Hypervisoren zu testen, erfordert jedoch ein gewisses Maß manueller Anpassung auf den jeweiligen Hypervisor. Außerdem werden hier System Hooks, also Funktionen, die bei der Ausführung von bestimmtem Systemcode ausgeführt werden, verwendet, um die Fuzzing Funktionalität im Gast umzusetzen.

Bis jetzt wurde jedoch keine Lösung vorgestellt, die die Suche nach Schwachstellen von Hypervisoren generisch, unabhängig vom zu testenden Hypervisor und verwendetem Gast, umsetzt.

1.5 Aufbau der Arbeit

Diese Arbeit ist unterteilt in vier Teile:

(I) Der erste Teil der Arbeit dient dazu dem Leser relevante Grundlagen zur Virtualisierung und der Verwendung von QEMU/KVM zu vermitteln. Dies ist das Kapitel 2.

(II) Der zweite Teil dieser Arbeit beschreibt zuerst die Anforderungen, die an das Projekt gestellt wurden, in Kapitel 3. Weiterhin wird in Kapitel 4 die Entwicklung der Fuzzing Lösung beschrieben und auf wichtige Elemente der Implementierung eingegangen. Außerdem wird die Umgebung und die Durchführung des Fuzzings beschrieben.

(III) Der dritte Teil dieser Arbeit präsentiert die Ergebnisse des Projektes. Dieser Teil befindet sich in Kapitel 5.

(IV) Zum Schluss wird ein Ausblick auf zukünftig geplante Funktionalität und Verbesserungen, sowie Herausforderungen gegeben. Außerdem werden die wichtigsten Aspekte und Punkte der Arbeit noch einmal in einem Fazit erläutert. Dies sind die Kapitel 6 und 7.

2 Grundlagen

2.1 Virtualisierung

In dieser Arbeit bezeichnet der Begriff Virtualisierung eine Softwaretechnologie, die die Ausführung mehrerer Betriebssysteme (Gäste) auf einem Hardwaresystem (Host) ermöglicht. Virtualisierung stellt hierbei eine Abstraktion des ausgeführten virtuellen Systems von der physischen Hardware dar. Diese Abstraktion ermöglicht es unter anderem mehrere Gäste auf einem einzigen Host auszuführen, ohne dass diese vollständigen Zugriff auf die physische Hardware erhalten. Zu diesem Zweck erfolgt häufig eine Emulation von Hardware.

Die Erklärungen der folgenden Abschnitte zeigen die Gegebenheiten bei der Verwendung von KVM und QEMU mit Intel Prozessoren auf. Diese sind größtenteils analog für die Verwendung anderer Virtualisierungsumgebungen.

2.1.1 Prozessor-Sicherheit

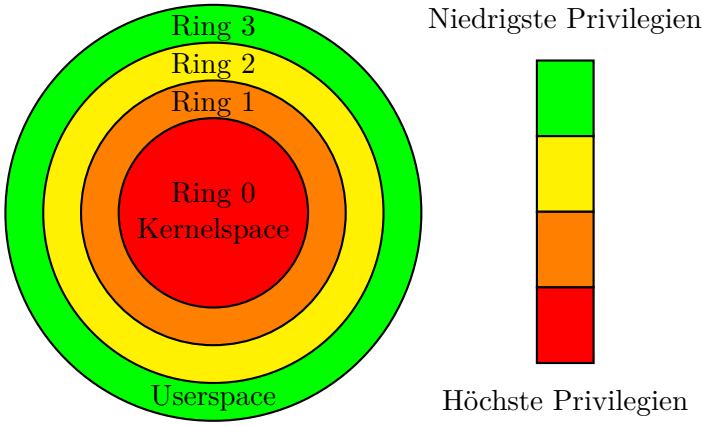


Abbildung 2: CPU Ringe [6]

Um Daten und Funktionalität von Systemen vor Fehlern und Angriffen zu schützen, bieten moderne Prozessoren sogenannte Protection Rings. Diese Ringe bestimmen, welche Berechtigungen der in ihnen ausgeführte Code besitzt. Der unterste Ring, Ring 0, hat hierbei vollen Zugriff auf die physische Hardware und kann (nahezu) beliebig mit ihr kommunizieren. Da in diesem Ring meist der Betriebssystemkern (Kernel) ausgeführt wird, nennt man diesen auch Kernelspace.

In der x86-Architektur existieren vier Ringe. Der Ring 1 und der Ring 2 werden nur sehr selten genutzt [7]. Der Ring 3 hingegen ist der Ring, in dem im Normalfall jegliche Benutzerprogramme ausgeführt werden. Er wird deswegen auch häufig als Userspace bezeichnet.

[6] Sinngemäß entnommen aus [4].
[7] Gegenbeispiel: OS/2 nutzte Ring 2 für I/O Operationen, Virtualbox nutzt Ring 1 für die Virtualisierung (vgl. [18]).

Um die volle Funktionsfähigkeit des Systems zu gewährleisten, existieren Kommunikationsschnittstellen zwischen den Ringen. So können die meisten Userspace-Programme ihre Aufgaben nur über Systemcalls erledigen. Diese führen, nach Überprüfung der Privilegien, zur Ausführung von Kernel Code in Ring 0.

2.1.1.1 Hypervisoren

Als Hypervisor, auch Virtual Machine Monitor (VMM), wird eine Software bezeichnet, die virtuelle Maschinen erstellt und ausführt. Um diese Funktionalität zur Verfügung zu stellen, muss ein Hypervisor gewisse Funktionen der Gäste emulieren. Dies betrifft insbesondere die Nutzung von Instruktionen, die erhöhte Privilegien und damit die Ausführung in Ring 0 erfordern.

Die Ausführung aller privilegierten Gast-Instruktionen direkt in Ring 0 würde dazu führen, dass der Gast volle Kontrolle über die Hardware des Hosts erhält. Deshalb ist es nötig, dass der Hypervisor über diese Instruktionen informiert wird und sie gegebenenfalls unterbinden oder emulieren kann. Das Gast-System bemerkt dies in der Regel nicht.

Um dies effizient zu ermöglichen, bieten moderne Prozessoren eine weitere Unterteilung der Privilegien in Ring 0. Diese Erweiterung wird bei Intel als Virtual Machine Extensions (VMX) bezeichnet. Hierbei heißen die beiden Modi `VMX Off` und `VMX On`. Der `VMX Off` Modus entspricht dem klassischen Verhalten des Ring 0. `VMX On` hingegen bietet dem Hypervisor die Möglichkeit, privilegierte Instruktionen des Gastes geeignet zu behandeln. Siehe hierzu auch Abschnitt 2.1.4.

2.1.2 Virtualisierung von Hardware

Dem virtualisierten Betriebssystem ist im Allgemeinen nicht bewusst, dass es nicht auf realer Hardware ausgeführt wird. Aus diesem Grund muss der Hypervisor dem virtualisierten System (Gast) zusätzlich virtualisierte Hardware zur Verfügung stellen. Software, die solche virtuelle Hardware zur Verfügung stellt, wird auch als Hardware-Emulator bezeichnet. Der Gast kann mit der virtuellen Hardware wie mit realer Hardware kommunizieren. Diese Kommunikation erfolgt im wesentlichen über die folgenden Schnittstellen:

- Port I/O
- MMIO
- Interrupts
- Zugriffe auf den RAM des Gastes, Direct Memory Access (DMA)

Da es sich bei dem Betriebssystem im Gast um ein vollständiges System handelt, kann es mit seiner Hardware auf jede beliebige Art kommunizieren. Es muss sich dabei nicht an etwaige Beschränkungen durch Treiber halten und sich auch nicht zwingend an die vorgesehenen Kommunikationsprotokolle halten.

2.1.3 Page Faults

In der Memory Management Unit (MMU) werden virtuelle Adressen von Prozessen in physische Adressen im RAM übersetzt. Dabei wird der virtuelle Adressbereich in sogenannte Pages, die zusammenhängende Abschnitte des RAMs darstellen, eingeteilt. Der physische RAM hingegen wird in sogenannte Frames unterteilt, die zumeist die gleiche Größe wie Pages besitzen. Um Daten in einem Frame lesen und schreiben zu können, muss diesem eine Page zugeordnet sein, die die Daten im physische Speicher enthält.

Der virtuelle Adressbereich ist häufig größer als der physische RAM, da jedem Prozess vorgetäuscht wird, den vollen Adressbereich nutzen zu können. Weiterhin besitzt jeder Prozess seinen eigenen virtuellen Adressbereich. Deshalb sind nicht allen virtuellen Pages Frames zugeordnet, sondern zumeist nur einem kleinen Teil. Greift ein Prozess auf eine virtuelle Adresse zu, deren Page noch keinem Frame zugeordnet wurde, so wird ein sogenannter Page Fault ausgelöst. Daraufhin wird der entsprechende Frame gegebenenfalls als Page in den physischen RAM geladen oder eine neue für diesen Prozess allokiert. (vgl. [14])

2.1.4 VM Entries und VM Exits

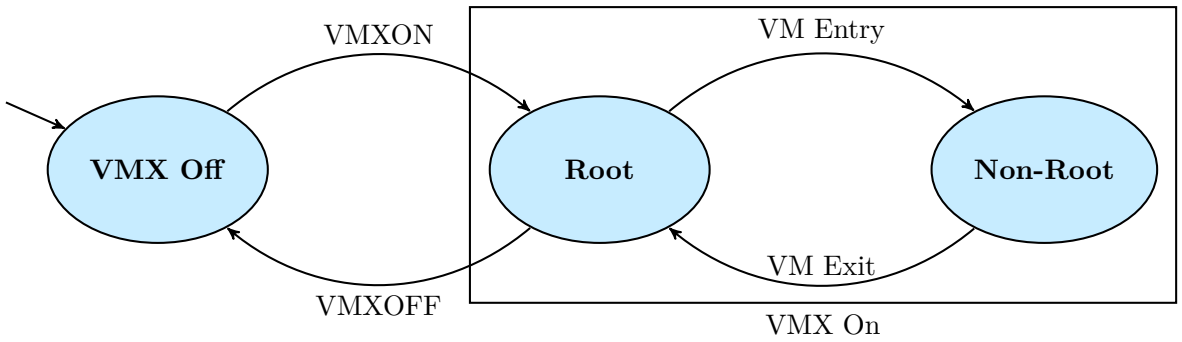


Abbildung 3: VMX Modi[8]

Das virtualisierte System soll im Allgemeinen von der tatsächlichen Hardware getrennt werden. Es soll nicht auf jegliche Instruktionen und Hardwarefunktionen vollen Zugriff besitzen. Deshalb ist es nötig, vor dem Ausführen von Instruktionen im Gast einen Wechsel vom vollprivilegierten Modus, bei Intel-Prozessoren als *VMX root* bekannt, in einen weniger privilegierten Modus (*VMX non-root operation*) durchzuführen. Diese Wechsel wird bei Intel-Prozessoren als *VMX Transition* bezeichnet. (vgl. [7, Kap. 23.3, S. 23-1])

Eine VMX Transition in die *VMX non-root operation* wird als VM Entry bezeichnet und kann durch die CPU-Instruktionen `VMLAUNCH` und `VMRESUME` ausgelöst werden. Bei diesem Wechsel wird unter anderem der Prozessorstatus der virtuellen Maschine initialisiert (`VMLAUNCH`) beziehungsweise geladen (`VMRESUME`). Weiterhin wird der Inhalt der Machine Specific Registers (MSR) wiederhergestellt, sowie diverse Überprüfungen durchgeführt. (vgl. [7, Kap. 26 VM ENTRIES,S. 26-1])

[8] Sinngemäß entnommen aus [6].

Eine VMX Transition in die *VMX root operation*, ein so genannter VM Exit, wird durch das Ausführen bestimmter Instruktionen und das Auftreten bestimmter Ereignisse ausgelöst. Gründe für VM Exits werden in [7, Appendix C, S. C-1-C-3] dargestellt. Bei einem VM Exit wird unter anderem der Prozessorstatus der virtuellen Maschine gesichert, die MSRs gesichert und der Prozessorstatus für *VMX root operation* wiederhergestellt. (vgl. [7, Kap. 27 VM EXITS, S. 27-1])

2.1.4.1 Monitor Trap Flag

Bereits seit dem Intel 8086 existiert im Flags Register das sogenannte Trap Flag, das dazu dient, die CPU in den Singlestepping-Modus zu versetzen. Das Setzen dieses Flags führt dazu, dass nach jeder Instruktion ein Interrupt ausgelöst wird. Daraufhin wird die Interrupt Service Routine ausgeführt. Mit Hilfe des Trap Flags ist es möglich, Programme Schritt für Schritt auszuführen und zu debuggen. (vgl. [2])

Die Verwendung des Trap Flags zum Debuggen virtueller Maschinen ist jedoch sehr aufwändig, da auch Instruktionen, die auf dem Host und nicht im Gast ausgeführt werden, einen Interrupt auslösen. Diese sind von den zu debuggenden Instruktionen im Gast nur schwer zu unterscheiden.

Zu diesem Zweck gibt es bei Intel-Prozessoren das sogenannte Monitor Trap Flag (MTF), das es ermöglicht, eine virtuelle CPU in den Singlestepping-Modus zu versetzen. So wird das Debuggen der virtuellen CPU vereinfacht. Wird das Monitor Trap Flag gesetzt, so führt das Ausführen der meisten Instruktionen zu einem VM Exit, der vom Hypervisor behandelt werden kann. Hierzu muss dieser einen Exit Handler für den entsprechenden Exit Reason registrieren.

2.1.5 Shadow MMU

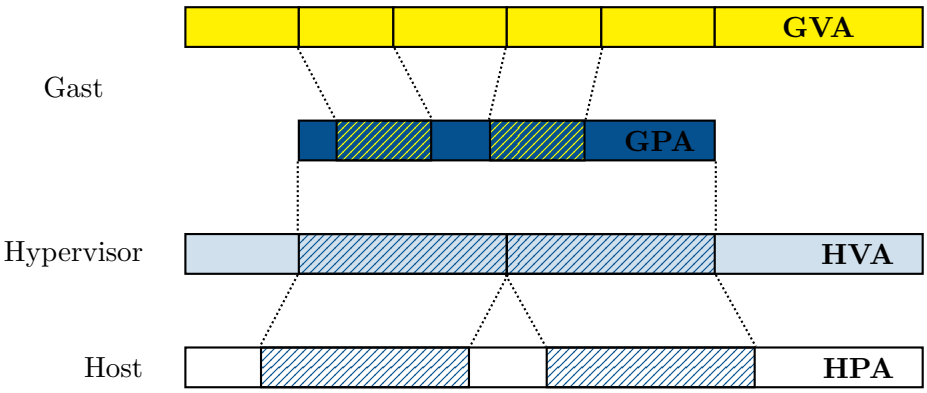


Abbildung 4: Speicherorganisation virtueller Maschinen[9]

Auf einem physischen System dient die MMU unter anderem dazu, virtuelle Adressen in physische Adressen zu übersetzen, um Prozessen Zugriff auf den RAM zu ermöglichen.

[9] Sinngemäß entnommen aus [5].

Bei der Virtualisierung wird jedoch ein weiteres Abstraktionslevel von Adressen eingeführt. So müssen nicht nur virtuelle Adressen von Prozessen auf physische Adressen umgesetzt werden, sondern virtuelle Adressen von Prozessen im Gast (Guest virtual address (GVA)) in physische Gast Adressen (Guest physical address (GPA)). Diese wiederum müssen in physische Adressen auf dem Host (Host physical address (HPA)) übersetzt werden.

Zu diesem Zweck wurde in KVM die Shadow MMU eingeführt. Sie dient dazu, diese Übersetzungen möglichst effizient auf die Hardware MMU abzubilden. Das Kernelement der Shadow MMU sind die so genannten Shadow pages, die die Shadow Page Table Entries (SPTEs) enthalten, die die tatsächliche Abbildung von GVA auf GPA auf HPA umsetzen. (vgl. [8])

2.1.5.1 Extended Page Tables

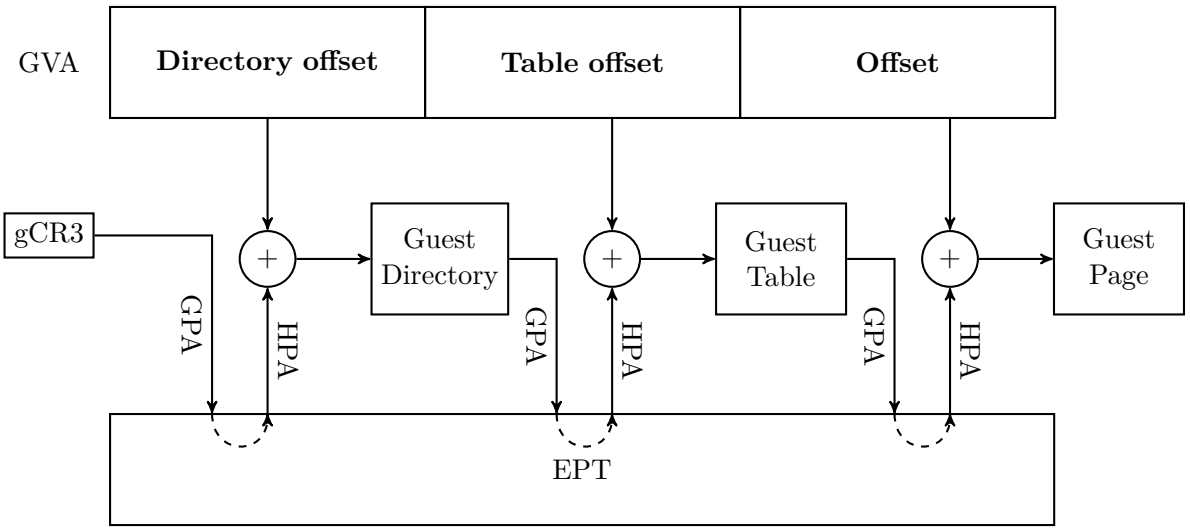


Abbildung 5: Adressübersetzung mit EPT[10]

Seit dem Westmere Modell[11] bieten Intel-Prozessoren Hardwareunterstützung für die sogenannte Second Level Address Translation, bekannt als EPT.

Second Level Address Translation fügt der MMU eine weitere Ebene zur Adressübersetzung hinzu. Diese erlaubt die Übersetzung von virtuellen Gastadressen in physische Gastadressen ohne VM Exit in den Hypervisor in Hardware zu codieren. Dies reduziert die Anzahl der VM Exits bei Page Faults in einer virtuellen Maschine deutlich, da nun viele Zugriffe aus der virtuellen Machine direkt in Hardware erledigt werden können.

Wie in Abbildung 5 dargestellt ist, können mittels EPT GPAs direkt in Hardware in HPAs umgewandelt werden. Der Gast hält, analog zur nicht virtualisierten Ausführung, Page Directories und Page Tables vor. Somit können aus GVAs wie beim normalen Paging-Verfahren direkt HPAs in Hardware bestimmt werden.

[10] Sinngemäß entnommen aus [5].
[11] ca. Ende 2009 von Intel entwickelte Mikroarchitektur

2.1.5.2 Memory Mapped I/O

Bei MMIO handelt es sich um Input/Output-Schnittstellen, die auf physische Adressen des Systems gemappt werden. Zugriffe auf diese Adressen müssen bei der Virtualisierung gesondert behandelt werden. Für die virtuellen Adressen, die auf MMIO-Speicher verweisen, existieren keine physischen RAM Pages, weshalb jeder Zugriff zu einem Page Fault führt.

Im Gegensatz zu normalen Page Faults müssen MMIO Page Faults auf jeden Fall durch den Emulator behandelt werden. Diesem werden die entsprechenden zugegriffenen Adressen mitgeteilt, so dass er die Emulation des zugehörigen Gerätes durchführen kann. Für virtuelle Adressen, die auf MMIO verweisen, werden keine Mappings in der Shadow MMU angelegt.

2.1.6 Virtual Machine Control Structure

Um die nötige Funktionalität für VM Exits und VM Entries anzubieten, stellen Intel-Prozessoren die Virtual Machine Control Structure (VMCS) zur Verfügung. Ein Hypervisor unterhält im Normalfall je eine Struktur für jede virtuelle Maschine. Außerdem kann er für jede virtuelle CPU, die einer virtuellen Maschine zugeteilt ist, eine eigene VMCS unterhalten. Nach dem Erstellen der VMCS erfolgen keine direkten Zugriffe auf die Struktur durch den Hypervisor. Für diese Zugriffe bieten Intel-Prozessoren die Instruktionen `VMCLEAR`, `VMPTRLD`, `VMREAD` und `VMWRITE`.

Byte Offset	Contents
0	Bit 0-30: VMCS Revision Identifier Bit 31: shadow-VMCS indicator
4	VMX-abort indicator
8	VMCS data

Tabelle 1: Format der VMCS Region[12]

2.1.6.1 VMCS Data

Der Datenbereich der VMCS Region ist aufgeteilt in sechs logische Gruppen:

1. **Guest-state area**: Zum Sichern des Prozessorstatus bei VM Exits und Laden des Prozessorstatus bei VM Entries.
2. **Host-state area**: Zum Laden des Prozessorstatus bei VM Exits
3. **VM-execution control fields**: Einstellungen für das Prozessorverhalten im VMX non-root Modus.
4. **VM-exit control fields**: Einstellungen für VM Exits.
5. **VM-entry control fields**: Einstellungen für VM Entries.
6. **VM-exit information fields**: Informationen zu VM Exits, Ursache und Art des Exits.

[12] Sinngemäß entnommen aus [7, Kap. 24, S. 24-2].

Für diese Arbeit sind insbesondere die *VM-Execution Control Fields* und die *VM-Exit Control Fields* von Interesse. Diese kontrollieren, welche Operationen im *VMX non-root* Modus zu VM Exits führen. Weiterhin sind die *VM-exit Information Fields* von Bedeutung, um nötige Informationen zu einem aufgetretenen Exit zu erhalten und verarbeiten.

2.1.6.2 VM-Execution Control

Die VM-Execution Control ermöglicht dem Hypervisor zu entscheiden, wie sich der virtuelle Prozessor bei bestimmten Ereignissen verhält. Dies beinhaltet zum Beispiel das Ausführen bestimmter Instruktionen. Dieses Feld ist aufgeteilt in zwei einzelne 32-Bit-Felder, bei denen jedes Bit jeweils ein Flag darstellt, das die jeweilige Funktionalität kontrolliert.

Bit Position	**Name**	**Beschreibung**
...	...	...
24	Unconditional I/O Exiting	Entscheidet, ob die Ausführung von I/O Instruktionen zu einem VM Exit führen.
...	...	...
27	Monitor Trap Flag	Aktiviert das Monitor Trap Flag.
...	...	...
31	Activate secondary controls	Entscheidet, ob das zweite 32-Bit-Feld der VM-Execution Controls beachtet wird.

Tabelle 2: Auszug aus den Primary Processor-Based VM-Exeution Controls[13]

2.1.7 Nested Virtualization

Moderne Prozessoren ermöglichen es, einen Hypervisor innerhalb einer virtuellen Maschine zu betreiben. Diese Funktionalität wird als Nested Virtualization bezeichnet. Somit ist es möglich, eine virtuelle Maschine innerhalb einer virtuellen Maschine auszuführen, indem der äußere Hypervisor (L0) für den inneren Hypervisor (L1) die für die Virtualisierung notwendige Funktionalität emuliert. Dies umfasst unter anderem die `VMLAUNCH`-/`VMRESUME`-Instruktionen.

Neben der Emulation einiger Instruktionen wird durch die Nested Virtualization auch beim Speichermanagement eine weitere Abstraktionsebene eingeführt. Diese ist für die Übersetzung von virtuellen Adressen des inneren Gastes (Nested guest virtual address (NGVA)) zu physischen Adressen des Hosts (HPAs) notwendig. Dies erfolgt durch die Speicherung des Mappings von virtuellen Adressen des inneren Gastes auf physische Adressen des äußeren Gastes in den Page Tables des äußeren Gastes. (vgl. [8])

[13] Sinngemäß entnommen aus [7, Kap. 24, S. 24-9 - 24 -10].
[14] Sinngemäß entnommen aus [11].

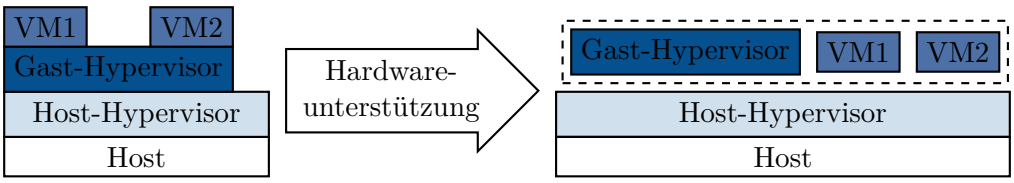


Abbildung 6: Nested Virtualization mit Hardwareunterstützung[14]

2.1.7.1 Nested VM Exit

Um die Nested Virtualization zu realisieren, ist es notwendig, dass VM Exits aus dem inneren Gast an den äußeren Hypervisor weitergeleitet werden. Dieser entscheidet dann, ob er selbst diesen Exit behandelt, oder ihn an den inneren Hypervisor zur Bearbeitung weitergibt. Dies ermöglicht dem äußeren Hypervisor unter anderem, bestimmte Zugriffe zu emulieren oder zusätzliche Funktionalität bereitzustellen.

Das heißt entsprechend, dass der Host sich über jeglichen VM Exit sowohl aus dem inneren als auch aus dem äußeren Gast informieren lassen kann. Dies ermöglicht, ein Gesamtbild beider virtueller Maschinen zu erstellen und so auch die Kommunikation zwischen beiden beobachten zu können.

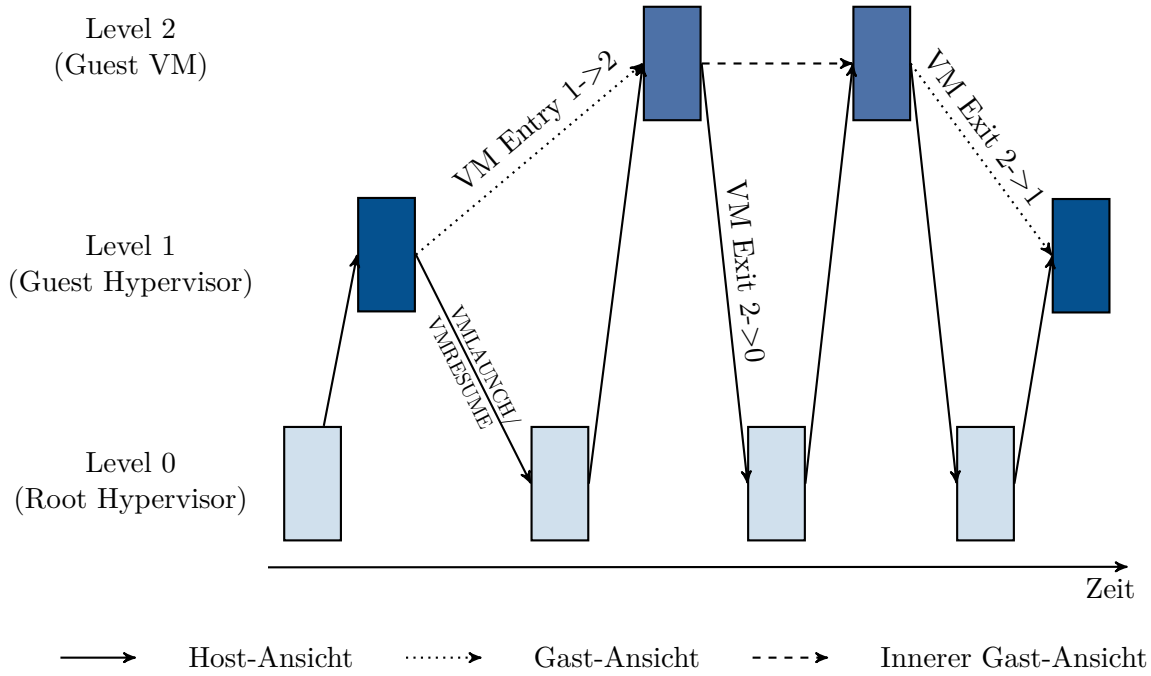
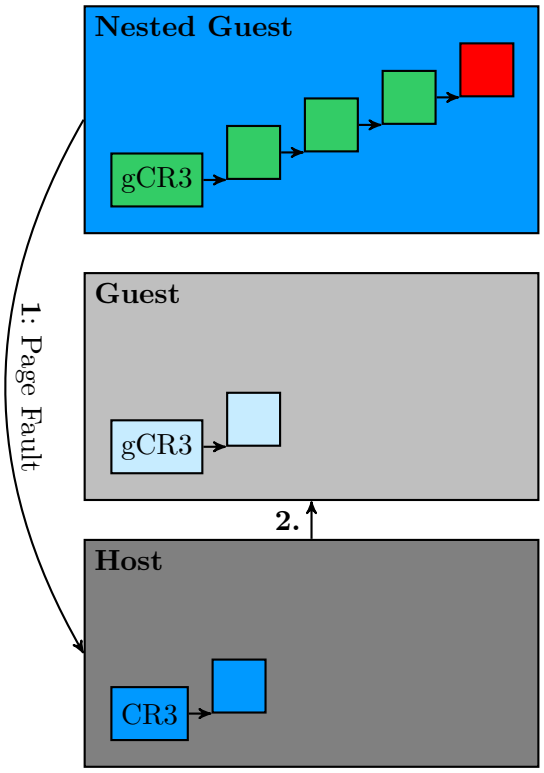


Abbildung 7: VMX Nested transitions[15]

[15] Sinngemäß entnommen aus [6].

2.1.7.2 Nested Shadow MMU

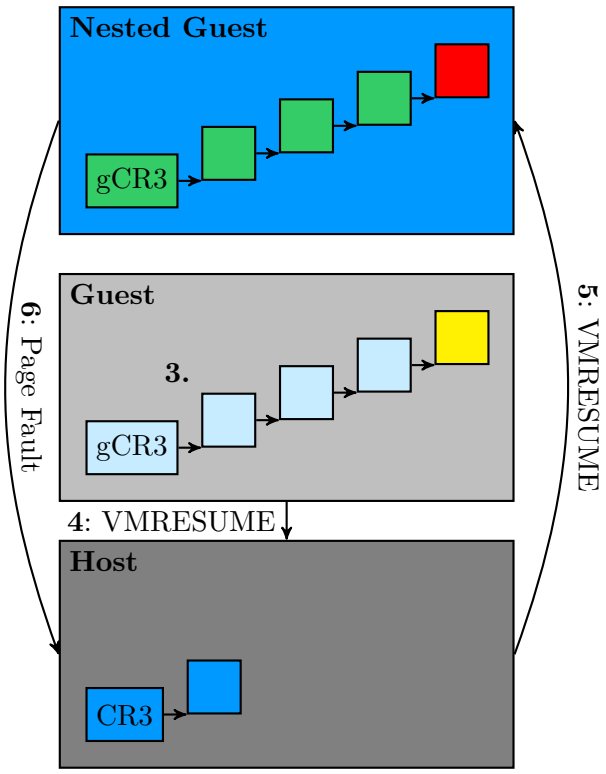
Durch das Hinzufügen einer inneren virtuellen Maschine wird auch die Auflösung von Speicheradressen um eine weitere Abstraktionsebene erweitert. Im Folgenden wird der Ablauf hierzu ausgehend von leeren Shadow Page Tables in Host und Gast beschrieben. [16]



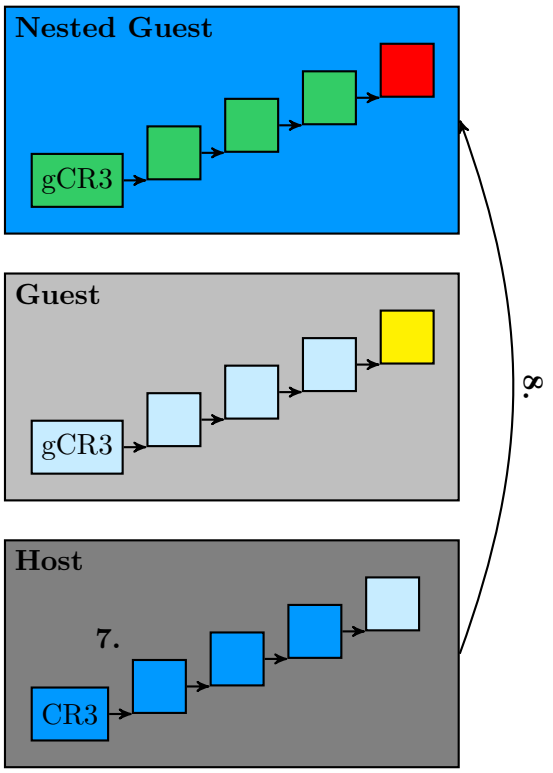
1. Der innere Gast greift auf eine seiner virtuellen Adressen zu. Dies führt zu einem Page Fault, da diese noch nicht auf eine physische Adresse gemappt ist. Der zum Page Fault gehörende VM Exit wird vom Host abgefangen.
2. Der Host gibt den Page Fault direkt an den äußeren Gast weiter, da dieser kein Mapping für die inneren virtuellen Adressen hat.

Abbildung 8: Übersetzung von virtuellen Adressen des inneren Gastes (Teil 1)

[16] Der Ablauf wurde sinngemäß aus [5] und [8] übernommen.



3. Der äußere Gast bearbeitet den Page Fault und trägt die Übersetzung der inneren virtuellen Adresse zu einer GPA in seine Shadow Page Table ein.
4. Der äußere Gast führt ein `VMRESUME` aus, welches einen VM Exit in den Host verursacht.
5. Der Host emuliert die `VMRESUME`-Operation des äußeren Gastes, so dass die Ausführung des inneren Gastes fortgesetzt wird.
6. Der innere Gast führt die Instruktion, die zu 1. führte, erneut aus, wodurch wieder ein Page Fault ausgelöst wird, der vom Host abgefangen wird.



7. Der Host trägt mit Hilfe der Shadow Page Table des Gastes die Übersetzung der virtuellen Adresse des Gastes zu einer Hardware physischen Adresse in seine Shadow Page Table ein.
8. Die Ausführung des inneren Gastes wird fortgesetzt. Da nun ein gültiges Mapping besteht kann der innere Gast ohne Page Fault auf die virtuelle Adresse zugreifen.

Abbildung 8: Übersetzung von virtuellen Adressen des inneren Gastes[17]

[17] Sinngemäß übernommen aus [5]

2.1.7.3 Nested EPT

Standardmäßig bietet EPT keine Hardwareunterstützung für Nested Virtualization. Trotzdem bietet die Verwendung für Nested Virtualization häufig Vorteile. KVM setzt Nested EPT softwareseitig um und erzielt so teils deutliche Performance-Unterschiede (siehe hierzu Abschnitt 5.2).

Statt EPT wie bei der einfachen Virtualisierung zur Abbildung von NGPAs auf HPAs zu nutzen, wird bei der Nested Virtualization EPT genutzt, um NGVAs auf NGPAs zu übersetzen. Somit kann der innere Gast seinen virtuellen Adressbereich komplett selbst verwalten. Die Abbildung von NGPAs auf HPAs wird über Shadow Page Tables implementiert.

Der Performancevorteil von Nested EPT ergibt sich aus der Tatsache, dass virtuelle Maschinen häufig deutlich langlebiger sind als normale Prozesse. Somit sind Page Faults beim Zugriff auf NGVAs deutlich häufiger als bei Zugriffen auf NGPAs. Durch die Verwendung von Nested EPT wird die Anzahl der durch diese Page Faults verursachten VM Exits deutlich verringert. (vgl. [10])

2.2 QEMU / KVM

Dieses Kapitel beschreibt die Kombination von QEMU und der im Linux-Kernel integrierten Virtualisierungsinfrastruktur KVM.

2.2.1 QEMU

QEMU (kurz für englisch: "Quick Emulator") ist ein unter der GPLv2 lizenzierter Open Source Emulator und Anbieter für virtuelle Maschinen. QEMU nutzt die dynamische Übersetzung von Prozessorinstruktionen, also die Übersetzung von Prozessorinstruktionen einer anderen Architektur (zum Beispiel ARM) zu Prozessorinstruktionen der Architektur des Hostsystems (zum Beispiel x86), um die Emulation zu ermöglichen. Alternativ kann QEMU den Code der virtuellen Maschine, mit Unterstützung durch das KVM-Kernelmodul oder den XEN Hypervisor, direkt auf der Host Maschine ausführen. (vgl. [13])

Neben der Emulation/Virtualisierung von Prozessoren bietet QEMU die Möglichkeit, andere virtuelle Hardware, wie zum Beispiel Netzwerkkarten oder Festplatten, sowie unzählige Peripheriegeräte, zu emulieren und so ein komplettes virtuelles System zur Verfügung zu stellen.

2.2.2 KVM

KVM (kurz für "Kernel-based Virtual Machine") ist ein Kernelmodul des Linuxkernels, das seit der Version 2.6.20 integriert ist. Es stellt Virtualisierungs-Infrastruktur im Kernel bereit. Um diese Infrastruktur bereitzustellen, nutzt KVM die Virtualisierungserweiterungen moderner Prozessoren (Intel VT beziehungsweise AMD-V). KVM selbst stellt nur eine Kernelspace-Unterstützung zur Verfügung, jedoch ist eine Userspace-Komponente seit Version 1.3 in QEMU enthalten. Da KVM keine eigene emulierte Hardware anbietet, ermöglicht die Kombination mit QEMU die Möglichkeit hardwareunterstützte Virtualisierung durchzuführen. KVM stellt also ausschließlich den für Virtualisierung benötigten Hypervisor im Kernelspace bereit.

2.2.2.1 Modul Struktur

Das KVM-Modul bietet die wesentliche Funktionalität für die Virtualisierung. Um Hardwareunterstützung nutzen zu können, ist jedoch zusätzlich entweder das KVM-Intel-Modul oder das KVM-AMD-Modul nötig. Diese können nur geladen werden, wenn das KVM-Modul geladen ist und nutzen für ihre Funktionalität vom KVM-Modul bereitgestellte Funktionen.

Hierbei bietet das KVM Modul grundlegende Funktionalität, zum Beispiel die Implementierung der Shadow MMU. Das Intel- beziehungsweise AMD- spezifische Modul bietet die Hardwareunterstützung für verschiedene Funktionalität, zum Beispiel die Nested Virtualization und EPT.

2.2.2.2 Kernel Parameter

Die Funktionalität des KVM-Moduls lässt sich über verschiedene beim Boot oder beim Laden des Moduls übergebene Parameter konfigurieren. Im Rahmen dieser Arbeit sind insbesondere die Parameter `nested` und `ept` interessant. Diese aktivieren die Unterstützung der entsprechenden Features.

So ist für die Verwendung von Nested Virtualization zwingend notwendig den Parameter `nested`, der standardmäßig deaktiviert ist, beim Laden zu setzen.

2.2.3 Emulation von Geräten

Um dem Gast Hardware anzubieten, stellt QEMU für verschiedene Hardware verschiedene Emulatoren zur Verfügung. Diese bilden möglichst genau das Verhalten der realen Hardware, die sie emulieren, ab, da der Gast mit ihr kommunizieren kann wie mit realer Hardware.

Um die Funktionalität von emulierter Hardware zur Verfügung zu stellen, nutzt QEMU die von KVM weitergegebenen VM Exits bei bestimmten Speicherzugriffen, Portzugriffen und MMIO. Die derzeit im Gast verfügbare Hardware muss sich für Port- und Speicherbereiche registrieren und kann dann bei Zugriffen auf diese die Funktionalität bereitstellen.

QEMU bietet der emulierten Hardware eine API, die wichtige Funktionalitäten implementiert. So existieren Hilfsfunktionen für DMA, sowie Funktionen für einheitliche Netzwerk- und Dateizugriffe. Für die korrekte Umsetzung der Hardware ist jedoch jeder Emulator selbst verantwortlich.

2.2.3.1 Implementierung eines Test Devices

Für die weitere Verwendung innerhalb des Projektes wurde ein Testgerät zur Emulation durch QEMU auf Basis der QEMU-API entwickelt. Das entwickelte Gerät bietet die folgende Funktionalität:

- Port I/O, lesend und schreibend, über einen Port.
- Memory Mapped I/O in zwei verschiedenen Speicherbereichen:
  - Speicher für freie Ein- und Ausgaben.
  - Speicher zum Ablegen einer physischen RAM-Adresse, um direkten Speicherzugriff zu ermöglichen.
- Direkter Speicherzugriff auf eine über MMIO übergebene Speicheradresse.
- Persistenz beim Speichern und Laden von Snapshots.

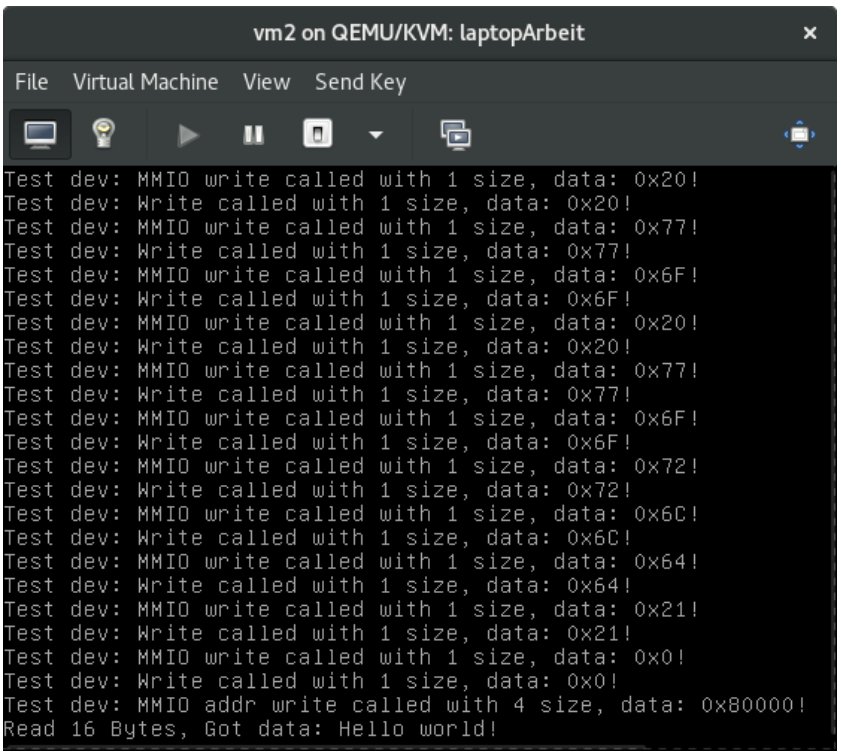


Abbildung 9: PCI Test Device auf der äußeren virtuellen Maschine

2.2.4 QEMU auf der Kommandozeile

Es existieren einige Werkzeuge, die QEMU über eine grafische Oberfläche ansprechen (insbesondere *libvirt* ist hier zu erwähnen). QEMU ist jedoch primär ein Kommandozeilenprogramm, das über verschiedene Parameter komfortabel anpassbar ist.

Einer der wichtigsten Parameter bei der Verwendung von QEMU auf einem System mit der gleichen Prozessorarchitektur wie die virtuelle Maschine ist `-enable-kvm`. Dieser aktiviert die KVM-Unterstützung und bietet so einen deutlichen Geschwindigkeitsvorteil. Weiterhin lassen sich Eigenschaften der emulierten Hardware über verschiedene Parameter beeinflussen. Der Parameter `-m` steuert die Größe des virtuellen Arbeitsspeichers, der Parameter `-hda` definiert, welches Image als primäre Festplatte verwendet werden soll. Weiterhin besteht die Möglichkeit, beliebige Peripheriegeräte über den Parameter `-device` an die virtuelle Maschine zu übergeben. Möchte man QEMU komplett ohne grafische Oberfläche nutzen, so ist es bei einigen Betriebssystemen, möglich ein TTY über einen seriellen Port zur Verfügung zu stellen und per `-nographic` die grafische Ausgabe zu deaktivieren. QEMU bietet auch die Möglichkeit mittels `-chardev` ein TTY zum Beispiel per Netzwerk bereitzustellen.

Auch das Laden von Snapshots ist über einen Parameter direkt zum Start der virtuellen Maschine möglich. Hierfür existiert der Parameter `-loadvm`.

Um die Hardwareunterstützung für Nested Virtualization in QEMU nutzen zu können, ist es weiterhin nötig, das CPU Feature „VMX“ an den Gast durchzureichen. Der verwendete Prozes-

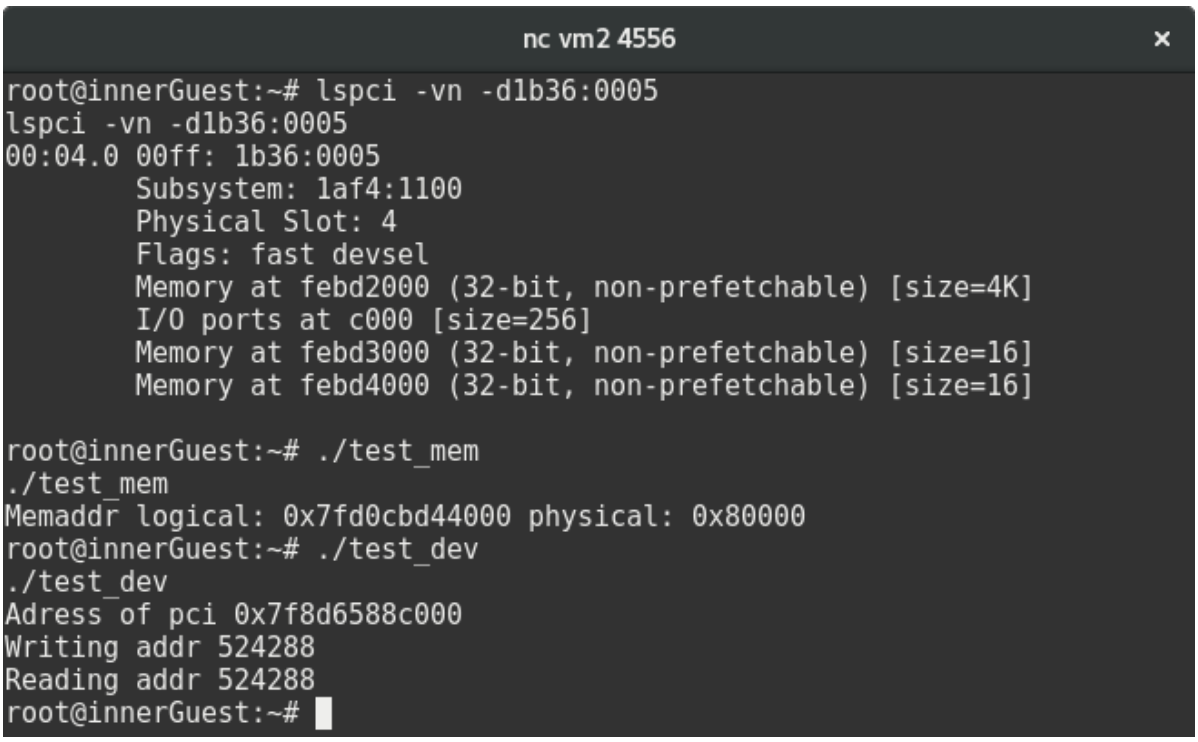


Abbildung 10: PCI Test Device auf der inneren virtuellen Maschine

sor kann bei QEMU über den `-cpu` Parameter konfiguriert werden. Um Features hinzuzufügen beziehungsweise zu entfernen, wird als Wert `+Feature` oder `-Feature` gesetzt.

Ein Aufruf von QEMU über die Kommandozeile könnte zum Beispiel wie folgt aussehen:

```
qemu -enable-kvm -m 512 -cpu host,+vmx -device pci-testdev -hda ./debian.qcow2
```

2.2.5 Image-Dateien

QEMU nutzt zur Bereitstellung virtueller Festplatten sogenannte Image Dateien. Diese Dateien entsprechen dem Inhalt der virtuellen Festplatte, können jedoch verschiedene Formate annehmen. Bei der Verwendung von QEMU sind die Formate `raw` und `qcow2` die gängigsten.

2.2.5.1 raw

Raw-Dateien sind einfache Binärdateien, die eine Eins-zu-Eins-Abbildung der virtuellen Festplatte darstellen. Im Allgemeinen sind sie genauso groß wie die virtuelle Festplatte. Sie können aber auf Dateisystemen, die Sparse Files unterstützen, weniger Speicher in Anspruch nehmen. Raw-Dateien sind der Standard-Dateityp für QEMU-Images.

2.2.5.2 QCOW2

QCOW2-Dateien (kurz für QEMU copy-on-write 2) sind Images, die eine dynamische Größe haben, die sich dem Füllstand der virtuellen Festplatte anpasst. Diese Anpassung ist unabhängig davon, ob das verwendete Dateisystem Sparse Files unterstützt. Das QCOW2-Format bietet die Möglichkeit, virtuelle Festplatten mittels AES zu verschlüsseln und mittels `zlib` zu komprimieren. Weiterhin unterstützt QCOW2 das Sichern mehrerer Systemzustände virtueller Maschinen (Snapshots), sowie die Verwendung von Overlay Images. (vgl. [15])

2.2.5.3 Snapshots

QEMU unterstützt das Sichern von Systemzuständen virtueller Maschinen in separate Dateien (bei beliebigem Image-Format), sowie das Sichern direkt in der virtuellen Festplatte (bei der Verwendung von qcow2). Ein solcher sogenannter Snapshot speichert den Zustand der gesamten virtuellen Maschine. Dies beinhaltet die Inhalte von CPU-Registern, RAM, virtuellen Festplatten und die Zustände von Geräten[18].

Snapshots können über den QEMU-Monitor im laufenden Betrieb erstellt und auch geladen werden. Außerdem gibt es die Möglichkeit, einen Snapshot mittels des Parameters `loadvm` direkt beim Start der virtuellen Maschine zu laden. So lässt sich zum Beispiel durch das Erstellen eines Snapshots direkt nach dem Boot der virtuellen Maschine der Bootvorgang durch das Laden des Snapshots überspringen.

2.2.5.4 Overlay Images

Bei der Verwendung von qcow2 Images ist es möglich, sogenannte Overlay Images zu erstellen. Dies sind Images, die auf einem sogenannten Backing File basieren und Änderungen relativ zu diesem speichern. Dies hat unter anderem den Vorteil, dass das ursprüngliche Backing File unverändert bleibt und somit weiterhin als Ausgangspunkt für neue virtuelle Maschinen dienen kann. Weiterhin sind Overlay Images im Allgemeinen deutlich kleiner als das Original-Image, da nur Veränderungen gespeichert werden.

Ein Nachteil der Verwendung von Overlay Images ist, dass diese, sobald das Backing File verändert wird, nicht mehr funktionieren. Deshalb ist es wichtig, Änderungen an dem Backing File zu vermeiden.

2.2.5.5 qemu-img

QEMU bietet zum Erstellen und Verwalten von Images das Kommandozeilentool `qemu-img` an. Relevant ist für diese Arbeit nur das qcow2-Format und die Verwendung von Overlay Images. Ein neues qcow2-Image kann mit dem Befehl

```
qemu-img create -f qcow2 debian.img 10G
```

[18] Vorraussetzung hierfür ist, dass dies vom jeweiligen Geräteemulator implementiert ist.

erstellt werden.

Soll nun ausgehend von diesem Image ein Overlay Image erzeugt werden, so wird hierzu der Befehl

```
qemu-img -f qcow2 -o backing_fmt=qcow2 -b debian.img debian_overlay.img
```

verwendet. Dieses Overlay Image lässt sich dann wie ein ganz normales qcow2-Image verwenden. Die Datei `debian.img` sollte nun nicht mehr verändert werden.

2.2.6 QEMU-Monitor

Während der Ausführung einer virtuellen Maschinen bietet QEMU die QEMU-Monitor Console an. Diese kann Informationen zur laufenden virtuellen Maschine anzeigen und verschiedene Aspekte davon manipulieren.

So bietet der Monitor über den Befehl `savevm` die Möglichkeit im laufenden Betrieb Snapshots zu erstellen und diese über `loadvm` wieder zu laden. Über den `info` Befehl kann der Nutzer Informationen über die virtuelle Maschine, zum Beispiel aktuelle Register und RAM-Inhalte abfragen. Auch das Herunterfahren und Neustarten einer virtuellen Maschine ist über den Monitor möglich.

Der QEMU-Monitor wird standardmäßig über eine Tastenkombination auf der Konsole aufgerufen, in der der QEMU-Prozess ausgeführt wird. Dies kann jedoch hinderlich sein, wenn auf dieser Konsole weitere Informationen angezeigt und verarbeitet werden sollen, weshalb QEMU den `-monitor` Parameter anbietet, um die Ausgabe umzuleiten.

2.2.6.1 Migration

QEMU bietet die Möglichkeit eine Migration von virtuellen Maschinen zwischen Hosts durchzuführen. Hierfür bietet der QEMU-Monitor den `migrate` Befehl an, der zum Beispiel das Verschieben einer virtuellen Maschine über eine TCP-Verbindung erlaubt.

Für diese Arbeit ist ein weiteres Feature der Migration von Interesse. Der Status einer virtuellen Maschine kann über „pseudo-migration“ in eine externe Datei gespeichert werden. Dies entspricht der Erstellung von Snapshots mit dem Befehl `savevm`, der jedoch kein Speichern in eine externe Datei erlaubt. Um dies zu realisieren wird `migrate` mit dem Parameter `exec` aufgerufen:

```
migrate "exec: cat > vmstate"
```

Dieser gesicherte Zustand kann dann, ohne dass die virtuelle Maschine ausgeführt wird, analysiert, bearbeitet und wieder geladen werden. Hierzu bietet sich zum Beispiel das Tool `lqs2mem`[19] an.

[19] `https://github.com/juergh/lqs2mem.py`

2.3 Fuzzing

Fuzzing, von Fuzz-Test („fuzz“ englisch: unscharf), bezeichnet eine Softwaretechnik zum automatisierten Finden von Implementationsfehlern und Sicherheitslücken in Software. Diese Technik wird realisiert, indem der Eingabeschnittstelle der zu testenden Software unerwartete, veränderte oder ungültige Daten übermittelt werden. Ziel des Fuzzings ist, es Abstürze oder Fehlverhalten der Applikation hervorzurufen. (vgl. [12])

2.3.1 Grundlegender Aufbau

Fuzzing wird im Allgemeinen in sechs Phasen aufgeteilt (vgl. [16]).

1. Identifikation der Zielsoftware
2. Identifikation der Eingabeschnittstelle
3. Generierung von Daten für die Eingabeschnittstelle
4. Übertragung der generierten Daten an die Eingabeschnittstelle
5. Überwachung der zu testenden Software. Dies dient der automatischen Erkennung von Abstürzen oder Fehlverhalten in der Software.
6. Untersuchung der gefundenen Fehler auf ihre Ausnutzbarkeit

Hierbei werden die Phasen drei bis fünf wiederholt, um mögliches Fehlverhalten mit verschiedenen Eingaben zu identifizieren. Phase sechs wird erst nach einem erfolgreichen Fuzzing-Durchgang durchgeführt.

2.3.2 Fuzzing in diesem Projekt

Als zu testende Software wurden in diesem Projekt Hypervisoren identifiziert. Die Eingabeschnittstelle, die gefuzzt werden soll, stellt die Kommunikation zwischen einer virtuellen Maschine und ihrer emulierten Hardware dar. Diese Arbeit beschäftigt sich im wesentlichen mit den Phasen vier, fünf und sechs. Insbesondere wird in den folgenden Abschnitten dargelegt, wie die Übertragung der generierten Daten an die emulierte Hardware erfolgt.

Die generierten Daten sind im wesentlichen die Daten, die vom Gast an die emulierte Hardware versendet wurden. Diese werden jedoch nach zufälligen Mustern verändert, um so zufällige Daten zu erzeugen.

3 Anforderungen

3.1 Zu testende Elemente

Um ein effizientes Fuzzing von Virtualisierungsumgebungen zu ermöglichen, wurden im Vorhinein einige Anforderungen an das entwickelte Projekt gestellt. Das zu entwickelnde Projekt soll es ermöglichen, jegliche Kommunikation zwischen einer virtuellen Maschine und ihrem Hypervisor zu beobachten und zu manipulieren. Dies beinhaltet sowohl die Kommunikation zwischen emulierter Hardware und der virtuellen Maschine, als auch jegliche weitere Zugriffe auf Daten, die der Gast manipulieren kann, durch die Virtualisierungsumgebung. Dies beinhaltet insbesondere:

- Portzugriffe durch den Gast
- MMIO-Zugriffe durch den Gast
- Speicherzugriffe auf den RAM des Gastes durch die Virtualisierungsumgebung, insbesondere DMA

Speziell wird in diesem Projekt der Fokus auf die Speicherzugriffe vom Hypervisor auf den Gast gelegt.

3.2 Angreifermodell

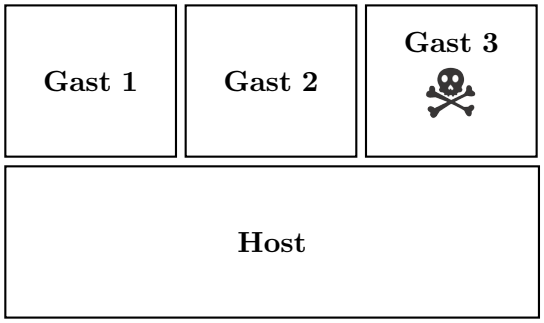


Abbildung 11: Angreifermodell[20]

Das Angreifermodell, dessen Angriffsvektoren durch die zuvor beschriebenen Elemente abgedeckt sind, ist in Abbildung 11 dargestellt. Ein Angreifer besitzt die volle Kontrolle über eine virtuelle Maschine[21] auf einem Host. Auf diesem Host werden gegebenenfalls weitere virtuelle Maschinen ausgeführt.

Der Angreifer kann in seiner virtuellen Maschine alle Kommunikation mit emulierter Hardware steuern. Er kann beliebige Portzugriffe und MMIO-Zugriffe durchführen und den RAM seiner virtuellen Maschine beliebig verändern. Über diese Schnittstellen versucht er, durch das

[20] Totenkopf Quelle: `https://www.iconfinder.com/icons/192531/crossbones_skull_spooky_icon`
[21] In diesem Fall Gast 3.

Auslösen von Fehlern in der Virtualisierungsumgebung, die Kontrolle über den Host oder die anderen virtuellen Maschinen zu erlangen.

3.3 Limitierungen bestehender Lösungen

Wie in Abschnitt 1.4 beschrieben, existieren bereits einige Lösungen zum Fuzzen von Virtualisierungsumgebungen. Diese weisen jedoch Limitierungen gegenüber der hier entwickelten und vorgestellten Lösung auf.

Das Projekt XenPwn (vgl. [19]) bietet die Möglichkeit, die paravirtualisierte Hardware des Hypervisors XEN zu testen. Hierbei spezialisiert es sich auf Memory Tracing. Damit ist das Projekt zum einen auf den Test von paravirtualisierter Hardware beschränkt und betrachtet zum anderen Portzugriffe und MMIO nicht. Des Weiteren beschränkt sich das Projekt ausschließlich auf das Testen des Hypervisor XEN.

Viele andere Lösungen, zum Beispiel XenFuzz (vgl. [1]), beschränken sich auf das Fuzzen von Hypercalls. Diese werden in dem hier vorgestellten Projekt nicht getestet.

Andere Projekte arbeiten mit Anwendungen, die speziell für den genutzten Gast entwickelt werden (siehe zum Beispiel [17]). Diese Projekte sind demnach nicht unabhängig davon, welches Betriebssystem im inneren Gast verwendet wird, und in den meisten Fällen auch nicht davon, welche Virtualisierungsumgebung genutzt wird.

4 Umsetzung

Um ein effektives und effizientes Fuzzing von Virtualisierungsumgebungen zu betreiben, wird die Nested Virtualization genutzt. Hierbei wird die Eigenschaft genutzt, dass im inneren Gast verursachte VM Exits immer zuerst dem äußeren Hypervisor mitgeteilt werden. Dieser entscheidet, ob sie dem inneren Hypervisor mitgeteilt werden. Somit sind im äußeren Hypervisor sowohl die VM Exits des inneren als auch des äußeren Gastes zu sehen.

Um das Fuzzing durchzuführen, wurde der Kernel des äußeren Hypervisors (Hostsystem) so gepatcht, dass Kommunikation zwischen dem inneren Hypervisor und dem Gast erkannt und an den Fuzzer im Userspace weitergeleitet werden kann. Der zu testende Hypervisor ist in diesem Szenario der innere. Die Kommunikation zwischen ihm und dem inneren Gast kann beobachtet und manipuliert werden. Der entwickelte Kernel Patch führt Änderungen am KVM Code durch, weshalb als äußerer Hypervisor KVM zum Einsatz gebracht werden muss. Der innere Hypervisor ist hingegen theoretisch nicht eingeschränkt.

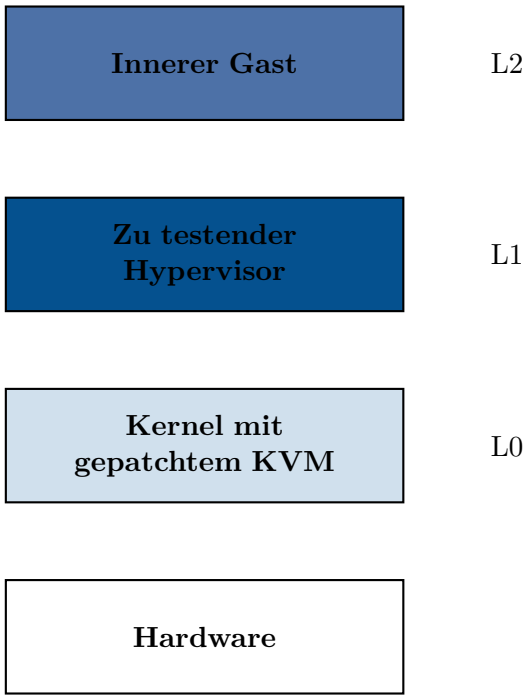


Abbildung 12: Struktur der Fuzzing Umgebung[22]

4.1 Struktur

Zur Umsetzung der gewünschten Funktionalität wurde der Kernel 3.16.7 eines Debian Jessie gepatcht. An Stellen, die relevante Zugriffe durch den inneren Hypervisor darstellen, können diese im gepatchten Kernel untersucht werden. Diese Zugriffe werden gegebenenfalls an den Fuzzer,

[22] Sinngemäß entnommen aus [3] und entsprechend der Verwendung im Projekt erweitert.

der im Userspace implementiert ist, weitergeleitet. An den betreffenden Stellen im Kernel wird dann auf Reaktionen aus dem Userspace gewartet und diese entsprechend umgesetzt.

4.2 Erkennung von Port I/O

Das Ausführen von I/O-Instruktionen in einer virtuellen Maschine führt bei Intel-CPUs zu einem VM Exit, falls eine der beiden Voraussetzungen

1. Das Flag `use I/O bitmaps` ist nicht gesetzt, aber `unconditional I/O exiting` ist gesetzt.
2. Das Flag `use I/O bitmaps` ist gesetzt und für mindestens einen der zugegriffenen Ports ist das entsprechende Bit in der Bitmap gesetzt.

erfüllt ist. (vgl. [7, Appendix C, S. C-2])

Dies ist auch bei der Nested Virtualization der Fall, so dass Portzugriffe aus dem inneren Gast zu VM Exits in den äußeren Hypervisor führen.

In KVM ist das unconditional I/O exiting Flag standardmäßig gesetzt, weshalb jeder Zugriff auf I/O Ports zu einem VM Exit führt. Zu jedem dieser Zugriffe fallen einige interessante Daten an:

- Der zugegriffene Port
- Die Breite des Zugriffs
- Ob es sich um einen lesenden oder schreibenden Zugriff handelt
- Gegebenenfalls welche Daten geschrieben wurden

Interessant sind in diesem Fall nur die Zugriffe aus dem inneren Gast, da diese eine Kommunikation mit der durch den inneren Hypervisor emulierten Hardware darstellen. Portzugriffe des äußeren Gastes werden für das Fuzzing ignoriert.

4.3 Erkennung von Speicherzugriffen

Ohne die Verwendung von Extended Page Tables führt jeder Page Fault in der äußeren virtuellen Maschine zu einem VM Exit. Hierbei gilt es jedoch zu unterscheiden, ob es sich bei dem Speicherzugriff um einen normalen Zugriff auf den RAM der äußeren virtuellen Maschine handelt, oder ob der innere Hypervisor auf den Gast der inneren virtuellen Maschine zugreift.

Um diese beiden Arten von Zugriffen zu unterscheiden, wurde ein Tracking der vom inneren Gast verwendeten GFNs eingeführt. Hierzu wird bei einem Zugriff des inneren Gastes auf seinen Speicher die zum Zugriff gehörende Guest Frame Number (GFN) gesichert. Da der erste Zugriff auf eine Speicherseite immer zu einem Page Fault führen muss, werden im laufenden Betrieb so alle vom inneren Gast verwendeten Speicherseiten gesichert.

Findet nun ein Zugriff durch die äußere virtuelle Maschine auf einen dieser Frames statt, so handelt es sich mit hoher Wahrscheinlichkeit um einen Zugriff auf den RAM des inneren Gastes. Interessante Daten zu diesem Zugriff beinhalten:

- Zugegriffene physische/virtuelle Addresse
- Zugegriffene GFN
- Schreibender oder lesender Zugriff
- Derzeit vorliegende Daten an dieser Adresse im RAM des inneren Gastes

Ein Hindernis bei der Erkennung der Speicherzugriffe ist, dass nicht jeder Zugriff auf eine Adresse zu einem Page Fault führt. Nachdem auf eine virtuelle Adresse zum ersten Mal zugegriffen wurde, wird für diese ein Shadow Mapping angelegt. Die MMU wird daraufhin so konfiguriert, dass weitere Zugriffe auf derselben Page nicht zu Page Faults führen. Dies verhindert die Erkennung dieser Zugriffe. Um dies zu umgehen, muss das angelegte Shadow Mapping für diese Page gelöscht werden, um einen erneuten Page Fault zu erzwingen.

Das Löschen des Shadow Mappings kann jedoch nicht während des VM Exits erfolgen, der durch den ersten Page Fault verursacht wird. Die virtuelle Maschine würde sich dann in einer Schleife von Page Faults befinden, da immer wieder dieselbe Adresse zu einem Page Fault führt. Auch das Löschen des Shadow Mappings beim nächsten beliebigen VM Exit ist nicht zwingend zielführend, da zwischen diesen beiden VM Exits weitere Speicherzugriffe innerhalb derselben Page erfolgen könnten. Das Verhindern von Shadow Mappings führt zum gleichen Problem. Aus diesem Grund wird das Monitor Trap Flag bei einem solchen Speicherzugriff gesetzt. Bei dem nach der nächsten Instruktion verursachten VM Exit kann das Shadow Mapping gelöscht werden, so dass ein erneuter Zugriff auf die Speicherpage zu einem Page Fault führt.

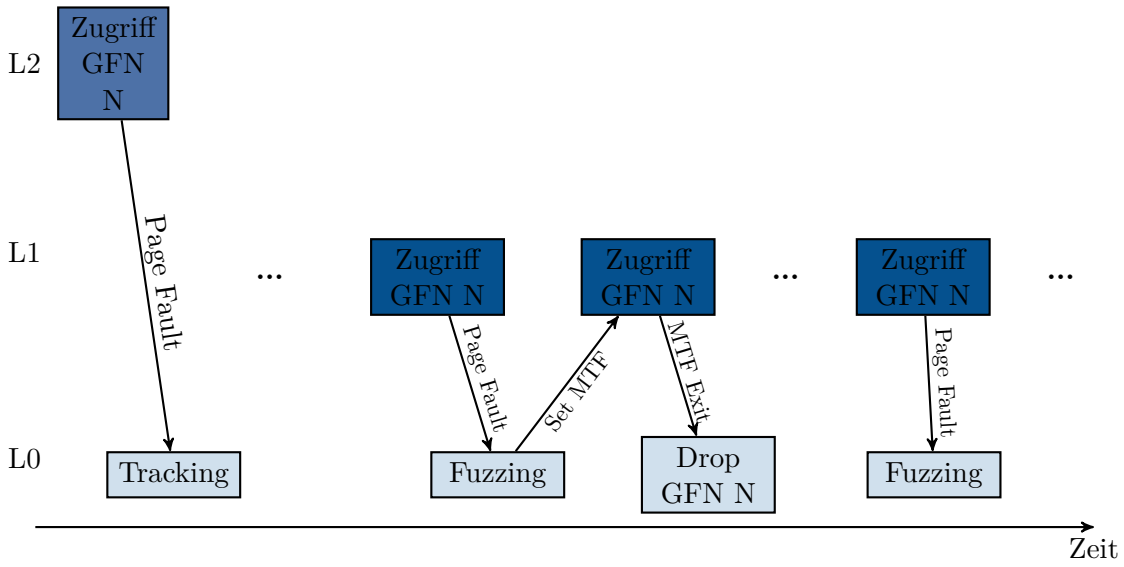


Abbildung 13: Erkennung von Speicherzugriffen

4.4 Schnittstelle zum Userspace

Der eigentliche Fuzzer wird mittels einer Schnittstelle im Kernel im Userspace implementiert. Tritt einer der zuvor genannten Zugriffe auf, so wird im Kernel ein Event erzeugt und an den Userspace gesendet. Der Fuzzer im Userspace schickt eine bestimmte Struktur zurück, die die durchzuführende Reaktion beschreibt.

Die Kommunikation mit dem Userspace erfolgt über Multicast Netlink Sockets. Dies ermöglicht es, einen Fuzzer zu entwickeln, der mit mehreren Threads arbeitet. Dadurch wird die Ausführung und das Fuzzen mehrerer virtueller Maschinen mit möglichst geringem Geschwindkeitsverlust ermöglicht.

4.4.1 Zuordnung von Gästen zu Fuzzer Threads

Die Ausführung einer virtuellen Maschine beginnt immer mit der Erstellung einer Virtual CPU (vCPU). In diesem Projekt wurde die Erstellung von vCPUs erweitert, so dass ihr eine Multicast-Gruppe zugeordnet wird. An diese Multicast-Gruppe werden alle Informationen zu diesem Gast gesendet. Der Fuzzer im Userspace kann so pro Thread eine Multicast-Gruppe auswählen, in der dieser liest. Durch die klare Zuordnung zu einer Multicast-Gruppe kann verhindert werden, dass dem zuständigen Thread Informationen entgehen.

Auch wenn jedem Gast eindeutig eine spezielle Multicast-Gruppe zugeordnet ist, gilt diese Eindeutigkeit nicht umgekehrt. In einer Multicast-Gruppe können Informationen zu mehreren Gästen gesendet werden. Ein Fuzzer Thread behandelt in diesem Fall mehrere Gäste. Um diese eindeutig trennen zu können, wird bei jeglicher Information zu einer virtuellen Maschine die eindeutige ID der vCPU mitgegeben. So kann der Fuzzer mehrere Gäste auf einem Multicast-Socket trennen.

4.4.2 Allgemeine Struktur

Über die Netlink Sockets werden C structs gesendet, die eine Type-Value-Struktur haben. Es wird hierbei zwischen Ereignis- (Event) und Reaktionsnachrichten (Reaction) unterschieden. Diese Unterscheidung erfolgt anhand des `nlmsg_type` der Netlink-Nachricht. Jede Nachricht enthält weiterhin einen Typ, der eindeutig bestimmt, um welche Art von Event beziehungsweise Reaction es sich jeweils handelt.

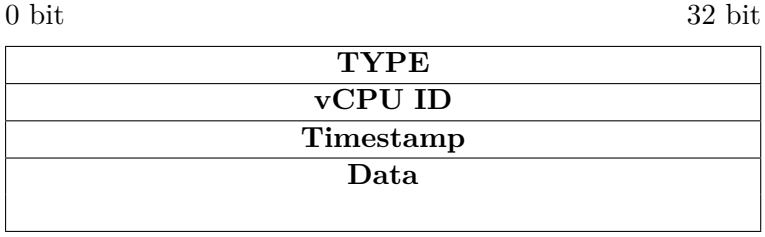


Tabelle 3: Grundlegende Nachrichtenstruktur

Die Länge des `DATA`-Feldes ist hierbei abhängig vom Typ der jeweiligen Nachricht.

TYPE kann bei Event-Nachrichten die folgenden Werte annehmen:

Value	Event type	Beschreibung
0	MMIO	Es wurde ein MMIO-Zugriff durch den inneren Gast durchgeführt.
1	PORTIO	Es wurde ein Port-Zugriff durch den inneren Gast durchgeführt.
2	MEMACCESS	Es wurde eine Speicherzugriff des äußeren Gastes auf den Speicher des inneren Gastes durchgeführt.
3	DONE	Die Ausführung des inneren Gast wurde beendet.

Tabelle 4: Auflistung der Event-Typen

TYPE kann bei Reaction-Nachrichten die folgenden Werte annehmen:

Value	Reaction type	Beschreibung
0	NONE	Ignoriere dieses Event.
1	SKIP	Überspringe die aktuelle Instruktion.
2	CHANGERAM	Ändere die übergebene Speicherstelle auf den übergebenen Wert.
3	CHANGERAX	Ändere den Inhalt des RAX-Registers auf den übergebenen Wert.

Tabelle 5: Auflistung der Reaction-Typen

4.4.3 Beispiel einer Kommunikation

In Abbildung 14 und Abbildung 15 sind beispielhaft Ausschnitte einer möglichen Kommunikation zwischen dem KVM-Modul und dem Userspace-Fuzzer zu sehen.

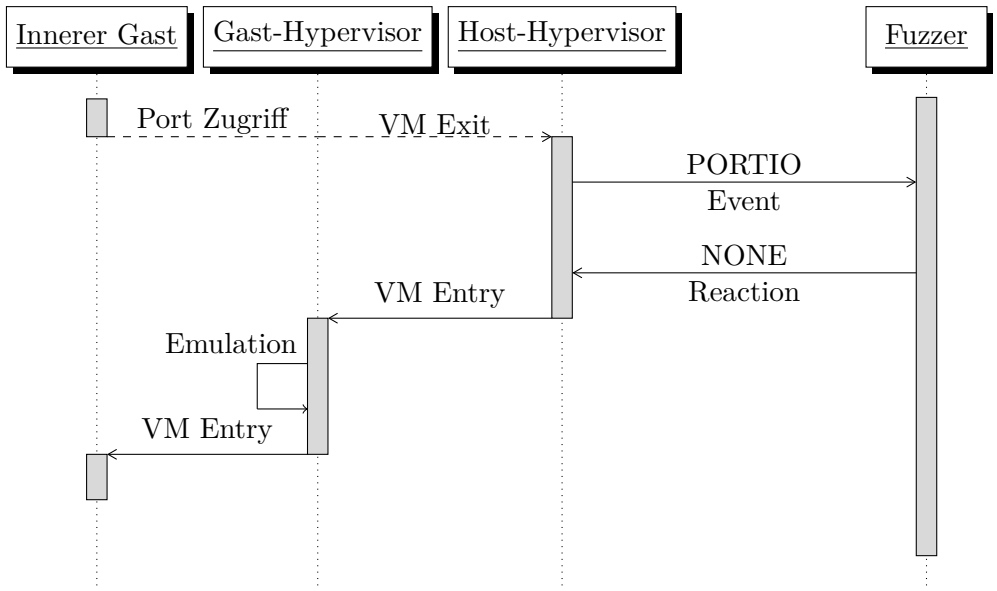


Abbildung 14: Erster beispielhafter Ablauf einer Fuzzing-Kommunikation

Abbildung 14 zeigt einen Port-Zugriff durch den inneren Gast. Dieser erreicht jedoch nicht den Gast-Hypervisor, sondern führt zu einem VM Exit in den Host-Hypervisor. Dieser generiert hieraus ein `PORTIO`-Event für den Fuzzer und sendet es diesem über den Netlink Socket zu. Der Fuzzer kann nun im Userspace auf dieses Event reagieren. Im dargestellten Beispiel reagiert er mit einer `NONE` Reaction und sendet diese über den Netlink Socket zurück an den Kernelspace. Das KVM-Modul führt daraufhin einen VM Entry in den Gast-Hypervisor durch. Dieser führt die Emulation des Port-Zugriffs durch und führt einen VM Entry in den inneren Gast durch. Danach wird die Ausführung des inneren Gastes bis zum nächsten Event fortgeführt.

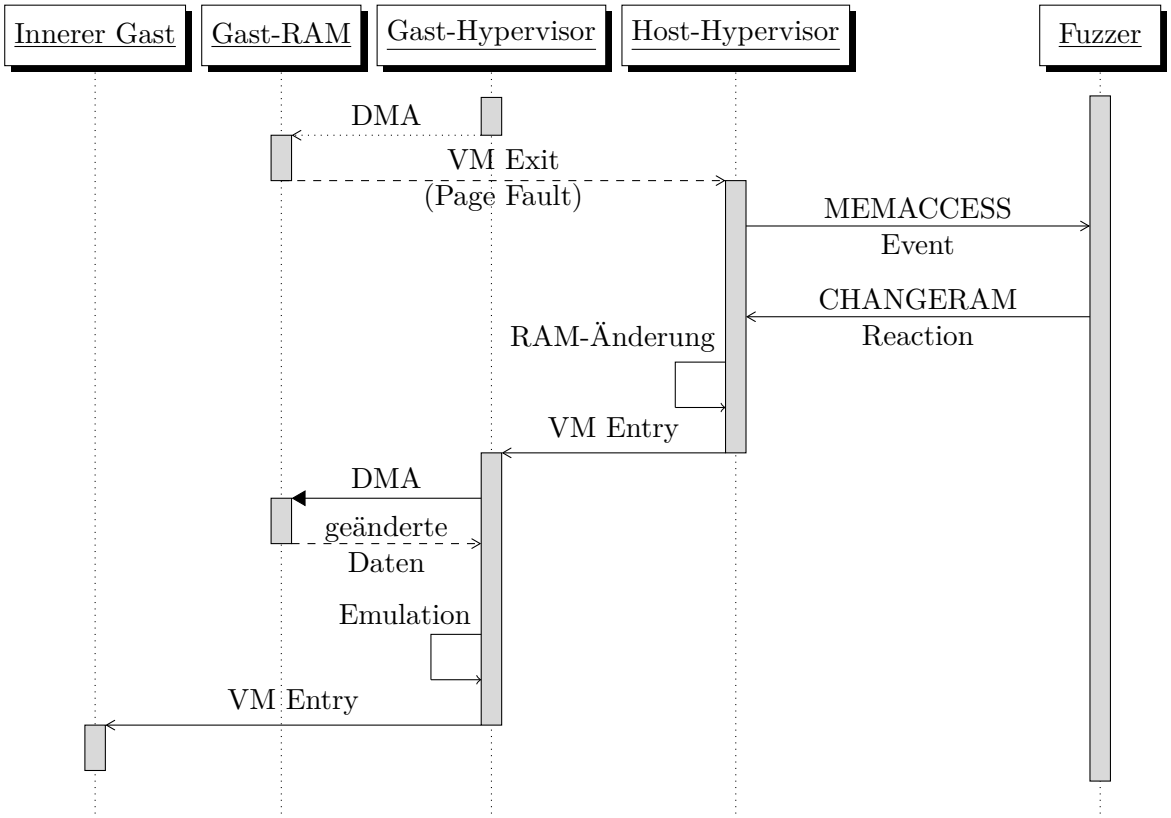


Abbildung 15: Zweiter beispielhafter Ablauf einer Fuzzing-Kommunikation

Abbildung 15 zeigt einen direkten Zugriff auf den Speicher des inneren Gastes durch die emulierte Hardware. Dieser Zugriff führt, da die zugegriffene Adresse noch nicht gemappt ist, zu einem Page Fault. Dieser wiederum führt zu einem VM Exit in den Host-Hypervisor. Dieser generiert ein `MEMACCESS` Event, das über den Netlink Socket an den Fuzzer im Userspace gesendet wird. Dieser reagiert mit einer `CHANGERAM` Reaction auf dieses Event. Nachdem diese Reaction das KVM-Modul erreicht hat, wird der Speicher des inneren Gastes entsprechend der Daten in der Reaction verändert.

Nach dem darauffolgenden VM Entry in den Gast-Hypervisor versucht dieser, erneut auf den RAM des inneren Gastes zuzugreifen. Da nun ein Mapping besteht, liest er die zuvor geänderten Daten aus dem RAM. Dann emuliert er die nötige Hardware-Funktion und führt einen VM Entry in den inneren Gast durch, der seine Ausführung bis zum nächsten VM Exit fortführt.

Um die Übersichtlichkeit zu wahren, wurde in Abbildung 15 die Nutzung des Monitor Trap Flags nicht dargestellt. Dessen Nutzung wird in Abschnitt 4.3 und Abschnitt 4.5.3.1 genauer erläutert.

4.5 Umsetzung im Kernel

Die Erkennung der an den Userspace gesendeten Events erfolgt im erweiterten KVM-Kernel-Modul. Wird ein Event erkannt, so wird es, wie zuvor beschrieben, über einen Netlink Socket dem Fuzzer mitgeteilt. Danach wird die Ausführung des Gastes blockiert, um dem Fuzzer die Möglichkeit zu geben, das Event zu verarbeiten und die entsprechende Reaction zurückzusenden. Diese Blockade erfolgt durch einen Mutex im KVM Code, der erst freigegeben wird, wenn eine entsprechende Reaction-Nachricht empfangen wurde.

4.5.1 Überspringen von Instruktionen

Wird die Reaction `SKIP` gesendet, so soll die aktuelle Maschineninstruktion übersprungen werden. KVM bietet hierfür Funktionalität, die den Instruction Pointer der virtuellen Maschine manipuliert, so dass dieser auf die nächste Instruktion zeigt. Damit wird effektiv die aktuelle Instruktion übersprungen.

Ein Problem bei diesem Weg, Instruktionen zu überspringen, ist, dass dies bei Port Ein- und Ausgaben häufig zu Abstürzen führt. Darum wird das Überspringen von Instruktionen bei Port I/O gesondert gehandhabt. Statt den Port I/O VM Exit an den äußeren Gast und den darin laufenden Hypervisor weiterzugeben, wie es normalerweise geschieht, wird der VM Exit ignoriert, jedoch als durch den äußeren Hypervisor abgearbeitet markiert. Dies hat zur Folge, dass der innere Gast glaubt, seinen Port Zugriff durchgeführt zu haben, dieser jedoch nie die emulierte Hardware erreicht.

Das Überspringen von Instruktionen kann unter anderem dazu genutzt werden, um eine bestimmte Kommunikation zwischen innerem Gast und innerem Hypervisor zu unterbinden.

4.5.2 Änderungen des RAX-Registers

Wird die Reaktion `CHANGERAX` gesendet, so wird der Inhalt des RAX-Registers der virtuellen CPU verändert. Hierfür bietet KVM geeignete Funktionen. Auch andere Register lassen sich problemlos verändern, dies ist jedoch in der aktuellen Version dieses Projektes nicht vorgesehen.

Die Veränderung des RAX-Registers sorgt dafür, dass die Daten, die über I/O-Instruktionen auf Ports der emulierten Hardware geschrieben werden, verändert werden können. Auch anderes Verhalten könnte in Zukunft über diese Reaktion verändert werden, gegebenenfalls durch die Erweiterung auf beliebige Register.

4.5.3 Änderungen des RAMs

Wird die Reaktion `CHANGERAM` vom Userspace-Fuzzer gesendet, so soll der RAM des inneren Gastes an einer bestimmten Adresse verändert werden. Der Fuzzer kann angeben, an welcher Adresse dies geschehen soll und welche Daten dort hinterlegt werden sollen. KVM bietet zum Verändern des Gast-RAMs Funktionen, mit denen dies problemlos möglich ist.

Das Verändern von RAM-Inhalten wird als Hauptfunktionalität zum Fuzzen genutzt. Insbesondere DMA kann hierüber gefuzzt werden, da der Speicher, der vom Hypervisor gelesen wird, direkt verändert werden kann. Dies sorgt dafür, dass der Hypervisor genau das liest, was vom Fuzzer vorgegeben wird.

4.5.3.1 Wiederherstellung von Daten

Veränderungen im RAM des inneren Gastes, wie sie zum Beispiel beim Anwenden der `CHANGERAM`-Reaction entstehen, führen häufig kurz darauf zu einem Absturz dieses Gastes. Dies liegt daran, dass sich in seinem Speicherbereich fehlerhafte beziehungsweise unerwartete Daten befinden. Um die Anzahl solcher Abstürze deutlich zu reduzieren, wird der geänderte Speicher nach der nächsten Instruktion des inneren Hypervisors auf seinen ursprünglichen Wert zurückgesetzt. Dies führt dazu, dass der innere Hypervisor im Speicher seines Gastes den geänderten Wert liest, der Gast selbst jedoch beim nächsten Zugriff wieder den ursprünglichen Wert liest.

Um diese Wiederherstellung zu realisieren, wird erneut das Monitor Trap Flag genutzt. Dieses wird im Falle einer externen Speicheränderung gesetzt und der ursprüngliche Inhalt der geänderten Speicherstelle gesichert. Bei dem auf die nächste Instruktion folgenden VM Exit wird der Inhalt wiederhergestellt, ohne dass der innere Gast selbst dies mitbekommt.

4.6 Logging

Um beim Fuzzen aufgetretene Probleme reproduzieren zu können, müssen jegliche Informationen, die anfallen, in einem definierten Format gesichert werden. Hierbei fallen Daten in hohem Volumen an. Diese müssen möglichst performant gesichert werden, da sonst die Ausführgeschwindigkeit des Fuzzers und damit der virtuellen Maschine reduziert wird. Hierzu wurde ein Logger geschrieben, der die aufgetretenen Events und die darauffolgenden Reaktionen im Speicher hält. Er hält diese so lange vor, bis er darüber informiert wird, dass diese auf der Festplatte persistiert werden sollen.

Um den Logger zu informieren, dass die Informationen nun gespeichert werden sollen, wurde zusätzlich zu den bestehenden Events ein weiteres hinzugefügt. Dieses informiert den Fuzzer darüber, dass eine innere virtuelle Maschine ihre Ausführung (ob aus eigenem Anlass oder im Fehlerfall) beendet hat. Der Fuzzer informiert dann wiederum den Logger hierüber.

```
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99ff8, gva
    7f44b0399ff8, data = 6760810100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99ff8, gva
    7f44b0399ff8, data = 6760810100000000000000000000000
2016-05-19 10:34:39 - REACTION: RAMCHANGE, id 3307, gpa 33b99ff8, data
    676181010101010101010101010101
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99880, gva
    7f44b0399880, data = 6740AF0100000000000000000000000
2016-05-19 10:34:39 - EVENT: MEMACCESS, id 3307, gfn 33b99, gpa 33b99ff8, gva
    7f44b0399ff8, data = 6760810100000000000000000000000
```

Abbildung 16: Auszug aus einem Log File

4.7 Umgebung

Damit das Fuzzing immer unter möglichst gleichen Bedingungen abläuft, wurden einige Vorkehrungen getroffen, die dies gewährleisten. Hierzu wurden einige Skripte entwickelt, die das Aufsetzen der Umgebung deutlich erleichtern.

Auf einem physischen System sollen mehrere virtuelle Maschinen gleichzeitig ausgeführt werden. Diese sollen alle auf die gleiche Art konfiguriert sein. Zusätzlich sollen hierbei möglichst wenig Ressourcen verbraucht werden. Deshalb bietet es sich an, alle virtuellen Maschinen ausgehend von einer virtuellen Maschine aufzubauen. Hierzu werden Overlay Images ausgehend von dieser Maschine erzeugt um einen möglichst geringen Speicheraufwand zu haben. Die Erstellung der virtuellen Maschinen ausgehend von einer Ausgangsmaschine wurde in einem Skript automatisiert, das die folgenden Schritte für jede zu erstellende virtuelle Maschine durchführt:

1. Erstellen eines Overlay Images
2. Starten der virtuellen Maschine
3. Konfiguration von Hostname, IP- und MAC-Adressen
4. Erstellen eines Snapshots, um die Bootzeit der Maschine zu verkürzen

5. Herunterfahren der virtuellen Maschine

Um alle virtuellen Maschinen in einen identischen Zustand zu bringen, wäre es nötig, Punkt vier schon mit dem Backing File durchzuführen. Allerdings ist das Laden von Snapshots, die mit dem Backing File erstellt wurden, nur sehr schwer möglich. Nach dem beschriebenen Verfahren befinden sich alle virtuellen Maschinen in einem ähnlichen Zustand, der als Ausgangspunkt für das Fuzzing genutzt wird.

4.8 Ausführung

Um das Fuzzing möglichst automatisiert und parallelisiert ablaufen zu lassen, wurde auch die Ausführung der virtuellen Maschinen durch ein Skript automatisiert. Dieses Skript führt bis zum Abbruch durch den Benutzer in einer Schleife für jede virtuelle Maschine folgende Schritte aus:

1. Laden des bei der Erstellung der virtuellen Maschine erzeugten Snapshots
2. Starten der inneren virtuellen Maschine mittels QEMU mit dem Parameter `-loadvm` um den auf der ursprünglichen virtuellen Maschine erstellten Snapshot zu laden
3. In einer Schleife: Ausführen von Befehlen in der inneren virtuellen Maschine
4. Beenden der äußeren virtuellen Maschine

Schritt drei wird hierbei so lange ausgeführt, bis entweder die innere virtuelle Maschine ihre Ausführung aufgrund eines Fehler beendet oder QEMU auf der äußeren virtuellen Maschine einen Fehler verursacht. Um einen sauberen Zustand zu gewährleisten, wird in beiden Fällen die äußere virtuelle Maschine vom erstellten Snapshot wiederhergestellt.

4.9 Reproduzierbarkeit

Um die Ergebnisse des Fuzzings geeignet auswerten zu können, ist es nötig, den Ablauf in der inneren virtuellen Maschine zu reproduzieren. Dies kann nur erreicht werden, indem der innere Gast die exakt gleichen Maschinenbefehle erneut ausführt. Hierzu wird als Ausgangspunkt die in Abschnitt 2.2.6.1 beschriebene Technik zur Sicherung des aktuellen Zustands des inneren Gastes genutzt. Der beschriebene Lösungsansatz ist hierbei für linuxbasierte innere Gäste entwickelt worden.

Wird über die „pseudo-migration“ der Status in eine Datei gesichert, so finden sich dort alle relevanten Informationen zur virtuellen Maschine. Dies umfasst den aktuellen RAM-Inhalt, sowie die aktuellen Register-Inhalte.

Bei der angelegten Datei handelt es sich um ein Binärformat, das in verschiedene Sections aufgeteilt wird. Dieses Format ist nur sehr spärlich dokumentiert und es existiert kein Werkzeug, um dieses zu parsen oder zu manipulieren. Deshalb wurde ein solches Tool in Grundzügen entwickelt.[23]

[23] Als Grundlage für die Entwicklung wurde das Tool `lqs2mem`(`https://github.com/juergh/lqs2mem.py`) verwendet, das ein Parsen der RAM Section in Snapshot Files ermöglicht.

4.9.1 Migration format

Die Binärdatei beginnt mit einem von QEMU generierten Header, der Informationen zum genutzten Speicherformat, sowie zu den einzelnen Sections enthält. Nach diesem Header folgen die einzelnen Sections, wie etwa der RAM oder der VRAM.

Die RAM Section ist eine Abbildung des physischen Speichers der virtuellen Maschine. Hierbei wird der Speicher in Form von Pages gesichert. Im Binärformat beginnt jede Page mit einem acht Byte langen Wert, der die Adresse dieser Page im physischen Speicher, sowie einige Flags definiert.

Verwendete Flags sind hier:

Flag	**Name**	**Beschreibung**
0x1	FULL_SAVE	Der komplette Speicher ist in der Datei am Stück abgelegt. (Nicht mehr genutzt)
0x2	COMPRESS_PAGE	Die Page ist komprimiert abgelegt, das nächste Byte beschreibt den Inhalt.
...	...	...
0x08	FULL_PAGE	Die Page ist komplett in der Datei abgelegt.
0x10	EOS	End of Stream. Der Migrationsstream ist an dieser Stelle unterbrochen, die nächste Page wird nachfolgend gesendet.
0x20	CONTINUE	Die RAM Section wird mit dieser Page fortgesetzt.
...	...	...

Tabelle 6: Page Flags in der Migrationsdatei

Ist das Flag `FULL_PAGE` angegeben, so befindet sich der Speicherinhalt dieser Page vollständig folgend in der Binärdatei. Ist die Seite jedoch komprimiert, so ist nur das nächste Byte von Bedeutung. Dieses gibt an, mit welchem Byte die Speicherseite gefüllt ist.[24]

In einer weiteren Section befinden sich die Register-Inhalte der virtuellen CPU. Diese sind in loser Folge ohne Trennung in der Binärdatei abgelegt.

0 32

RAX	RCX	RDX	RBX
...	RBP	RSI	RDI
R8	R9	R10	R11
R12	R13	R14	R15
RIP	...		

Tabelle 7: Register-Reihenfolge in der Migrationsdatei

[24] Dieses Byte entsprach in allen durchgeführten Tests 00 und damit einer nicht genutzten Seite.

4.9.2 Manipulation des aktuellen Zustands

Um einen beim Fuzzing geloggten Ablauf zu reproduzieren, muss die Migrationsdatei entsprechend manipuliert werden. Die virtuelle Maschine soll beim Laden der Datei den Ablauf erneut ausführen, indem die gleichen Maschineninstruktionen erneut ausgeführt werden. Theoretisch wäre es ausreichend, den Speicher, auf den der Instruction Pointer zeigt, zu verändern und dort neue Maschinenanweisungen zu hinterlegen.

Jedoch bestehen dabei zwei Probleme. Zum einen ist der Return Instruction Pointer als virtuelle Adresse angegeben, das Speicherabbild in der Migrationsdatei ist jedoch am physischen Speicher ausgerichtet. Somit ist es nur durch Auslesen der Adress-Mappings möglich, die nötige physische Adresse zu bestimmten. Zum anderen kann der Instruction Pointer auf eine beliebige virtuelle Adresse zeigen, die sich unter Umständen direkt am Ende einer Page befindet. In diesem Fall wäre der Platz für die neuen Instruktionen unter Umständen nicht gegeben.

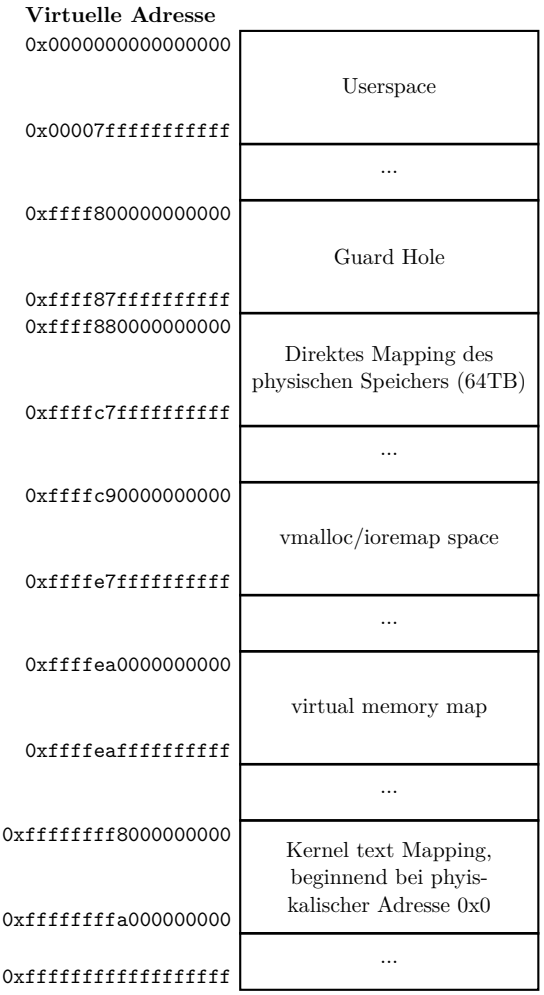


Abbildung 17: Virtual Memory Map eines Linuxsystems[25]

[25] Sinngemäß entnommen aus [9].

Um beide Probleme zu lösen, wird der Instruction Pointer auf eine andere Speicheradresse umgelegt. Hier bietet sich eine Adresse an, die in einen bestimmten physischen Speicherbereich gemappt wird. Dies ist bei einem Linuxsystem unter anderem die Kernel Text Area.[26] Diese beginnt bei der virtuellen Adresse `0xffffffff80000000` und endet bei `0xffffffffa0000000` und ist im physischen Speicher durchgehend von der Adresse `0x0` gemappt (vgl. [9]).[27]

Nun legt man Maschinenanweisungen in diesem Speicherbereich in der Migrationsdatei ab und ändert den Return Instruction Pointer auf die entsprechende Adresse. Dann führt das Gast-System diese Instruktionen nach dem Laden der Migrationsdatei aus. Somit ist es möglich, ausgehend von einem Migrationsstand beliebige Instruktionen im Gast auszuführen. Somit besteht ein Ausgangspunkt, um ein Fuzzing Ergebnis zu reproduzieren, auch wenn einige weitere Faktoren dabei relevant sind.

[26] Dies gilt nur unter der Vorraussetzung, dass Kernel Address Space Layout Randomization deaktiviert ist.
[27] Dies gilt nur für ein System mit mehr als einem Gigabyte RAM. In jedem Fall sind jedoch einige der Speicherpages direkt in den physischen RAM gemappt.

5 Ergebnisse

Da zum Zeitpunkt dieser Arbeit die Entwicklung des Projektes noch nicht vollständig beendet ist, ist davon auszugehen, dass sich die Performance noch deutlich verbessern lässt. Weiterhin ist davon auszugehen, dass bei weiterem Testen Fehler und Sicherheitslücken in Virtualisierungsumgebungen zu finden sind.

5.1 Limitierungen

Mit der entwickelten Schnittstelle kann die Kommunikation zwischen einer virtuellen Maschine und ihrer emulierten Hardware beobachtet und manipuliert werden. Erste Ergebnisse zeigen, dass dies funktioniert und die Kommunikation verändert werden kann. Trotzdem besitzt die entwickelte Lösung derzeit noch einige Einschränkungen (siehe hierzu auch Abschnitt 7).

Zum einen ist es derzeit nur schwierig nachzuvollziehen, wie ein beim Fuzzing entstandener Fehler verursacht wurde. Obwohl eine sehr präzises Logging aller anfallenden Informationen erfolgt, ist nicht direkt nachvollziehbar, welche Veränderung den Fehler ausgelöst hat. Dies ist dadurch bedingt, dass in den meisten Fällen mehrere Änderungen bei einem Fuzzing-Durchlauf durchgeführt werden. Dabei ist nicht klar, welche von diesen zum Fehler geführt hat. Um diese Einschränkung zu verringern, muss die Reproduzierbarkeit der Fuzzing-Ergebnisse verbessert werden, so dass bei einem Fehler die Kommunikation entsprechend erneut durchgeführt werden kann. Dann könnte durch ein Testen der verschiedenen Veränderungen festgestellt werden, welche von diesen zum beobachteten Fehler führen.

Eine weitere Einschränkung betrifft die fehlende Möglichkeit zu entscheiden, welche emulierte Hardware gefuzzt werden soll. Die Schnittstelle im Kernel leitet derzeit alle Speicherzugriffe aus dem äußeren auf den inneren Gast an den Userspace weiter. Der Fuzzer im Userspace kann nicht ohne Weiteres unterscheiden, von welcher emulierten Hardware diese Zugriffe ausgehen. Um spezielle emulierte Hardware fuzzen zu können, wäre es nötig, zumindest im Userspace zusätzliche Informationen unter anderem über DMA-Adressen dieser Hardware zu erlangen.

5.2 Performance

Klassisch wird die Performance von Fuzzing-Anwendungen in Tests pro Sekunde gemessen. In diesem Projekt ist es jedoch schwierig, eine solche Aussage zu treffen. Es werden nicht ausgehend von einem Seed beliebige Eingaben generiert und permutiert, bei denen jede einen kompletten Test darstellt. Stattdessen wird ein existierender Datenstrom manipuliert. Dieser stellt die Hardware-Kommunikation zwischen Gast und Hypervisor dar. Diese Kommunikation verläuft für zwei Fuzzing-Durchgänge unter Umständen sehr unterschiedlich und kann auch nach sehr unterschiedlichen Zeiten zum Ende kommen. Deshalb scheint die Einteilung des Fuzzings in einzelne Tests in diesem Fall nicht sinnvoll.

Um trotzdem eine Indikation über die mit diesem Projekt erreichte Performance zu geben, wurden einige Benchmarks durchgeführt. Diese bieten einen gewissen Einblick in die Gesamtperformance des Systems. Sie sind jedoch abhängig von äußeren Faktoren, wie etwa der Last auf dem Hostsystem.

Die Benchmarks wurden jeweils mit vier verschiedenen Systemen durchgeführt:

- Host Maschine mit unmodifiziertem Kernel und abgeschaltetem EPT
- Host Maschine mit unmodifiziertem Kernel und eingeschaltetem EPT
- Gepatchter Kernel mit eingeschaltetem Tracking von Portzugriffen Diese werden an den Fuzzer im Userspace gesendet, der für den Test immer mit `NONE` Reactions antwortet
- Gepatchter Kernel mit eingeschaltetem Tracking von Port- und Speicherzugriffen, inklusive Droppen von GFNs. Der Fuzzer im Userspace antwortet auf diese Events mit `NONE` Reactions.

Alle Benchmarks werden in einer inneren virtuellen Maschine auf dem jeweiligen Host durchgeführt.

Ein Benchmarking beim tatsächlichen Fuzzing, also mit Veränderungen des RAMs oder der Manipulation von Portzugriffen, ist nicht möglich, da der innere Gast hierdurch in seiner Ausführung zu instabil wird (siehe hierzu Abschnitt 5.3).

Weiterhin sollten die Benchmark-Ergebnisse mit Vorsicht betrachtet werden, da sie zwar eine Indikation für die Ausführgeschwindigkeit des inneren Gastes geben, aber nicht unbedingt repräsentativ für die Durchführung des Fuzzings sind. Hierbei ist nicht die Ausführgeschwindigkeit des inneren Gastes von Interesse, sondern die Menge der gefuzzten Daten. Diese lassen sich jedoch nicht vergleichbar mit einem unmodifizierten System testen.

5.2.1 I/O Tests mit FIO

Um die I/O Performance der Nested Virtualization mit Verwendung des Fuzzers zu testen, wurde das Tool FIO[28] verwendet. Im Folgenden wurde hierfür immer derselbe Befehl zum Ausführen von FIO genutzt:

```
fio --randrepeat=1 --ioengine=libaio --direct=1 --gtod_reduce=1 --name=test --filename=test
              --bs=4k --iodepth=64 --size=4G --readwrite=randrw --rwmixread=75
```

Hierdurch wird eine 4-Gigabyte-große-Testdatei nach einem reproduzierbaren Format erzeugt, auf die nach einem zufälligen Muster zu 75 Prozent lesend und zu 25 Prozent schreibend zugegriffen wird. Dies wird mit bis zu 64 I/O-Operationen gleichzeitig durchgeführt. Dieser Test soll beliebige Zugriffe auf die Festplatte simulieren.

Wie aus Tabelle 8 ersichtlich, ergibt sich aus der Verwendung von EPT bei I/O-Instruktionen kaum ein Vorteil. Dies ist dadurch bedingt, dass EPT für die Verwaltung von Speicheradressen zuständig ist, was für I/O wenig relevant ist.

[28] `https://github.com/axboe/fio`, verfügbar im Debian stable Repository

Kernel	Lesen	Schreiben
Unmodifiziert	406 KB/s	135 KB/s
Unmodifiziert mit EPT	405 KB/s	135 KB/s
Tracking von Port I/O	400 KB/s	135 KB/s
Tracking von Port- und Speicherzugriffen	117 KB/s	39 KB/s

Tabelle 8: Testergebnisse FIO

Die Identifikation von Portzugriffen und deren Weiterleitung an den Userspace Fuzzer hat ebenfalls nur einen sehr geringen Einfluss auf die Performance bei I/O. Hierbei wurden auch nur relativ wenige Portzugriffe festgestellt und bearbeitet.

Bei der vollständigen Identifikation aller DMA- und Portzugriffe lässt sich jedoch ein deutlicher Performanceunterschied messen. Beim Lesen und Schreiben handelt es sich hier jeweils ungefähr um eine Verschlechterung um Faktor 3,5. Dieser Unterschied ergibt sich unter anderem aus den zusätzlichen VM Exits, die für das Löschen der Adressmappings nötig sind.

5.2.2 Kernel compile benchmark

Für einen weiteren realistischeren Test wurde auf dem Gast in den verschiedenen Konfigurationen das Kompilieren eines Kernels durchgeführt. Hierzu wurde der folgende Ablauf verwendet:

```
1 make mrproper
2
3 # Hier wird sofort exit und Speichern ausgewählt
4 make menuconfig
5
6 make dep
7 time make bzImage
```

Kernel	Laufzeit (in Minuten)
Unmodifiziert	51:14.672
Unmodifiziert mit EPT	14:04.028
Tracking von Port I/O	65:58.561
Tracking von Port- und Speicherzugriffen	830:51.392

Tabelle 9: Testergebnisse Kernel compile

Wie in Tabelle 9 zu erkennen, ist hier ein deutlicher Performancegewinn durch die Verwendung von EPT zu erkennen. Beim Kompilieren werden häufig neue Prozesse gestartet. Diese erzeugen neue virtuelle Adressen. Dadurch kann der grundlegende Vorteil von Nested EPT, die selbstständige Verwaltung seine Adressraums durch den innere Gast, genutzt werden. Somit ergibt sich hier ein Performanceunterschied von ungefähr Faktor 3,6.

Bei aktivierter Identifikation von Portzugriffen zeigt sich bereits ein kleiner Performanceunterschied, der jedoch mit ungefähr 30 Prozent eher gering ausfällt.

Bei der vollständigen Identifikation von Port- und Speicherzugriffen hingegen zeigt sich eine deutliche Verschlechterung der Performance. Hier ist mit einem Faktor von etwa 16,2 deutlich erkennbar, dass dies einen Performanceunterschied verursacht.

Diese deutliche Verschlechterung ist ebenfalls unter anderem dadurch bedingt, dass beim Kompilieren viele Prozesse erzeugt werden. Diese habe jeweils ihren eigenen Adressbereich, der natürlich, wie in Abschnitt 4.3 beschrieben, getrackt werden, muss um Speicherzugriffe zu erkennen. Zusätzlich werden durch die Verwendung des MTF eine Menge zusätzlicher VM Exits verursacht, die die Ausführung zusätzlich verlangsamen.

5.3 Ergebnisse des Fuzzens

Zum Zeitpunkt dieser Arbeit steht eine umfassende Auswertung der Fuzzing-Ergebnisse noch aus. Es wurden bisher einige verschiedene Verhaltensweisen beim Fuzzen festgestellt. Diese lassen sich grob in zwei Bereiche einteilen. Zum einen sind dies Abstürze, Fehler und Timeouts im inneren Gast. Zum anderen sind dies Fehlermeldungen von QEMU beziehungsweise der emulierten Hardware.

	Kernel Version	**QEMU Version**
Host	3.16.7 (modifiziert)	2.5.0
Äußerer Gast	3.16.0-4	2.5.0
Innerer Gast	3.16.0-4	-

Tabelle 10: Fuzzing Umgebung

Die aufgetretenen Fehler und Meldungen entstanden bei der Verwendung von QEMU Version 2.5.0 mit aktivierter KVM-Unterstützung im äußeren Gast. Bei der äußeren virtuellen Maschine handelte es sich um ein Debian Jessie mit Kernel 3.16.0-4. Das auf dem Host verwendete QEMU liegt in der Version 2.5.0 vor. Im Inneren Gast wurde ebenfalls ein Debian Jessie mit Kernel 3.16.0-4 ausgeführt. Die angeschlossene emulierte Hardware war die Virtio Standardhardware für Festplatten und Netzwerkkarten.

5.3.1 Innerer Gast

Die durch das Fuzzing ausgelöste Verhaltensänderungen der Emulatoren haben ihrerseits Einfluss auf den Gast. Diese mittelbare Beeinflussung durch das Fuzzing kann zu Fehlern im Gast führen. Insbesondere handelt es sich hierbei um:

- Kernel BUGs
- Kernel Panics
- Kernel Null Pointer Dereferences
- Kernel Paging Requests

Diese Ergebnisse stellen erwartungsgemäß den größten Teil der Fuzzing-Ergebnisse dar. Da aber nicht das Verhalten des inneren Gastes beobachtet werden soll, ist die Auswertung dieser Abstürze nur von geringer Relevanz.

5.3.2 QEMU und emulierte Hardware

Beim Fuzzen der Kommunikation zwischen der emulierten Hardware und dem inneren Gast kommt es jedoch auch zu Fehlermeldungen von QEMU. Auffällig ist insbesondere, dass QEMU in einem solchen Fall sofort die Ausführung der virtuellen Maschine beendet. Wird also ein Fehler in der Kommunikation festgestellt, so wird die Virtualisierung zumeist direkt beendet.

Insbesondere traten beim Fuzzen die folgenden Fehlermeldungen auf:

- Emulation Failure
- Virtio Error
  - Fehler beim Mappen von MMIO Speicher
  - Ungültige Größen für Buffer
  - Ungültige/falsche Deskriptoren
  - Übergabe falscher Indizes

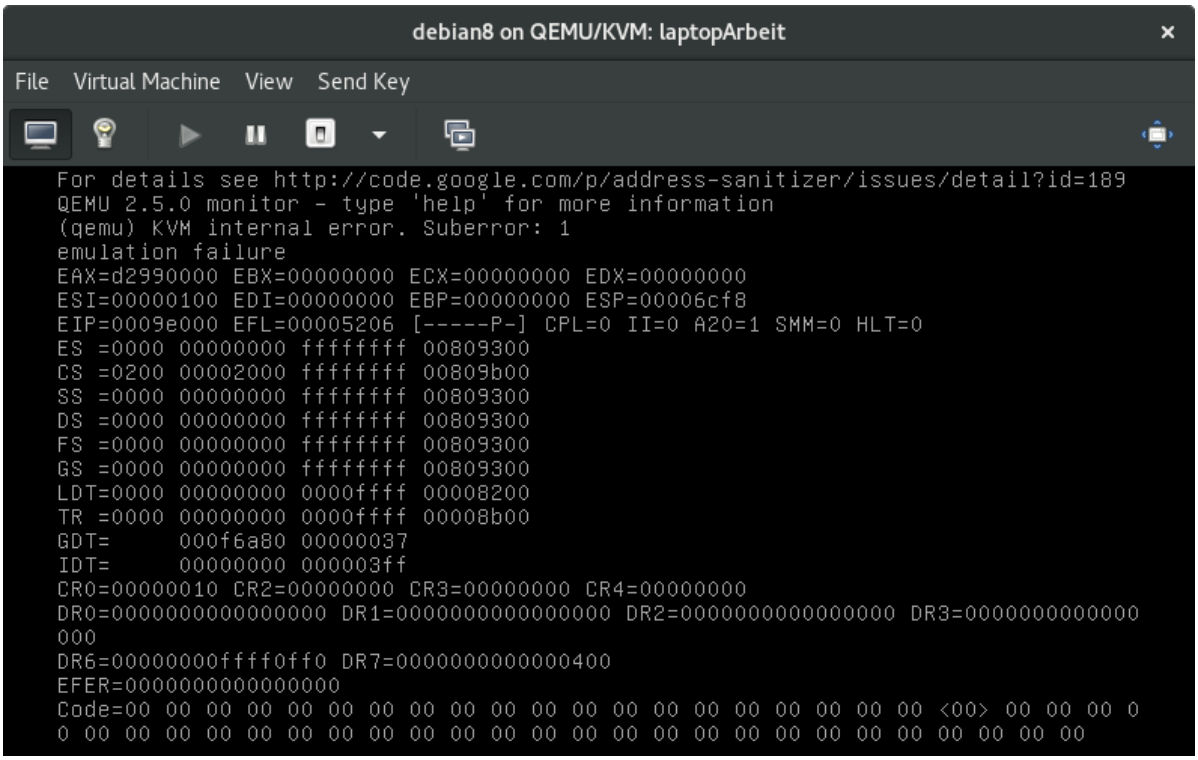


Abbildung 18: Emulation Failure

Emulation Failures können in QEMU aus verschiedensten Gründen auftreten. Häufig handelt es sich hierbei um Fehler, die die Emulation einer Instruktion betreffen. Dies tritt zumeist auf, wenn einer Instruktionen ungültige oder unerwartete Daten übergeben werden. Dies ist beim Fuzzing-Ansatz unter Umständen gegeben, da unerwartete Daten an den verwendeten Speicheradressen im RAM liegen können. Emulation Failure-Fehler können jedoch auch unter anderen Umständen entstehen, die im Nachgang dieser Arbeit weiter analysiert werden.

Die aufgetretenen Virtio-Fehler hingegen stellen jeweils die veränderte Kommunikation zwischen innerem Gast und emulierter Hardware dar. Die geänderten Speicherstellen enthalten hierbei für den Virtio-Treiber unerwartete Daten, die zu einer Fehlermeldung und daraufhin zum Beenden der Ausführung führen. Auch hier bietet sich eine weitere Analyse der gefundenen Probleme an.

Ein weiterer beim Fuzzen auftretender Fehler waren Interrupts, deren Interrupt Request (IRQ)-Nummer negativ war. Diese werden von der emulierten Hardware an den Gast gesendet. Negative IRQs sind bei physischer Hardware nicht vorgesehen und sollten somit bei standardkonformer emulierter Hardware auch nicht auftreten. Auf dem Host System löst dies eine Warnung in KVM aus. Dies deutet auf einen noch nicht weiter analysierten Fehler in QEMU hin. Mögliche Ursachen stellen eine falsche Eingabeüberprüfung, sowie ein Überlauf dar.

6 Fazit

In der Vergangenheit wurden immer wieder Sicherheitslücken und Fehler in Virtualisierungsumgebungen gefunden. Diese stellen eine reale Gefahr für jegliche Umgebung dar, die Virtualisierung nutzt. Dies gilt insbesondere für das Cloud Computing. In Einzelfällen kam es gar dazu, dass aus virtuellen Maschinen der Host komplett kompromittiert wurde.

Das in dieser Arbeit beschriebene Projekt bietet einen neuen, automatisierbaren Ansatz, um solche Schwachstellen generisch zu suchen. Es erlaubt ein generisches Fuzzing der Kommunikation mit emulierter Hardware. Es ist hierbei konzeptionell unabhängig davon, welche Virtualisierungsumgebung und welches innere Betriebssystem genutzt wird. Hierbei kann die Kommunikation zwischen Gast und emulierter Hardware nach Belieben manipuliert werden, um so Fehler in den emulierten Geräten hervorzurufen.

Die ersten Testläufe des Projektes zeigen, dass es möglich ist, die Kommunikation zwischen Gast und emulierter Hardware zu manipulieren. Hierbei wurden verschiedene Abstürze des inneren Gastes und Fehlermeldungen der Virtualisierungsumgebung ausgelöst. Hierzu musste weder das verwendete Betriebssystem im inneren Gast noch der zu testende Hypervisor modifiziert werden.

Derzeit lassen sich trotz der generischen Umsetzung keine anderen Hypervisoren als QEMU/KVM mit dem Projekt testen. Dies ist aber in Zukunft vorgesehen und prinzipiell auch möglich.

Auch wenn das Potential des Projektes somit noch nicht vollständig ausgeschöpft wird, so ist die grundlegende Funktionalität erfolgreich umgesetzt worden. Damit ist der gewählte Lösungsansatz funktionsfähig. Das Projekt stellt natürlich keinen Ersatz für ausführliche Tests von emulierten Geräten bei der Entwicklung dar, kann diese aber effizient ergänzen.

7 Ausblick

Extended Page Tables

Die Verwendung von Extended Page Tables erlaubt es, die Anzahl von VM Exits aufgrund von Page Faults deutlich zu reduzieren. Dies erhöht Ausführgeschwindigkeit von virtuellen Maschinen. Somit würde die Verwendung von EPT auch das Fuzzing beschleunigen, da dies natürlich von der nativen Geschwindigkeit der virtuellen Maschinen abhängig ist.

Ein weiterer Vorteil der Verwendung von EPT ist, dass es nicht mehr nötig ist, nach jedem Zugriff auf den Speicher des inneren Gastes jegliche Shadow Mappings zu löschen. EPT bietet die Möglichkeit Speicheradressen als nicht zugreifbar zu markieren. Hierdurch wird beim Zugriff ein VM Exit ausgelöst, der zum Fuzzen genutzt werden kann. Auch dadurch würde sich die Ausführgeschwindigkeit der virtuellen Maschine beim Fuzzen erhöhen, da das Neuanlegen der gelöschten Mappings Ressourcen fordert, die sich so verringern lassen.

Eine Herausforderung an der Erweiterung des Codes auf die Verwendung mit EPT stellt das Tracking der Memory Pages des inneren Gastes dar. Aufgrund der durch EPT behandelten Page Faults wird kein VM Exit ausgelöst, der derzeit zum Tracken genutzt wird.

Fuzzing anderer Hypervisoren

Konzeptionell können beliebige Hypervisoren als innere Virtualisierung genutzt werden. Da derzeit nur ein relativ geringer Anteil des Codes fest auf die Verwendung von QEMU/KVM als inneren Hypervisor ausgelegt ist, besteht die Möglichkeit, den Code mit relativ geringem Aufwand zum Testen anderer Hypervisor zu erweitern. So könnten die gleichen Untersuchungen auch an Hypervisoren wie VMware oder XEN durchgeführt werden.

Die Nested Virtualization in KVM bietet derzeit offiziell nur Unterstützung für KVM als inneren Gast. Jedoch ist inoffiziell auch Support für andere Hypervisoren, wie zum Beispiel XEN oder VMWare vorhanden, der aber in der für dieses Projekt verwendeten Kernel Version 3.16 noch nicht funktionsfähig ist. Auch in der neusten Kernel Version 4.6 [29] ist ohne die Aktivierung von EPT keine Ausführung von inneren virtuellen Maschinen möglich. Bei aktiviertem EPT lassen sich jedoch auch virtuelle Maschinen zum Beispiel in Virtualbox starten.

Sollte das Projekt auf die Verwendung von EPT erweitert werden, reicht diese Änderung unter Umständen bereits aus, um auch weitere Hypervisoren auf Fehler zu überprüfen.

Reproduzierbarkeit

Ein signifikantes Problem stellt die Reproduzierbarkeit der Ergebnisse dar. Sowohl in der inneren als auch in der äußeren virtuellen Maschine wird ein vollständiges Betriebssystem ausgeführt, das wiederum eine Menge unterschiedlicher Prozesse ausführt. Aufgrund dieser Komplexität ist es kaum möglich, das Verhalten, das während einer Ausführung der inneren virtuellen

[29] Stand: 31.05.2016

Maschine abläuft, zu reproduzieren. Um die Ergebnisse des Fuzzing jedoch verwendbar zu machen, muss die Möglichkeit bestehen, dieses Verhalten verlässlich zu wiederholen.

Der in Abschnitt 4.9 beschriebene Ansatz ist derzeit noch nicht vollständig ausgearbeitet und ausgereift. Jedoch zeigen erste Prototypen, dass dieser grundsätzlich funktioniert und die Reproduzierbarkeit zumindest in Teilen ermöglicht.

Ein bisher noch ungelöstes Problem stellt jedoch dar, dass so nur die Ausführung der inneren virtuellen Maschine kontrolliert werden kann. Die äußere virtuelle Maschine jedoch handelt weiterhin nicht immer reproduzierbar.

Interrupts

Interrupts dienen dazu, das Betriebssystem über Ereignisse der Hardware zu informieren. Sie behandeln somit keine Daten des Betriebssystems und werden deshalb im aktuellen Projektstand nicht betrachtet. Da sie jedoch trotzdem eine unter Umständen für das Verständnis der Kommunikation wichtige Information darstellen, sollten sie in Zukunft auch dem Fuzzer mitgeteilt werden.

Interrupts stellen auch für die Reproduzierbarkeit eine Hürde dar, da sie unter Umständen asynchron erfolgen und so schwer zu reproduzieren sind.

Memory Mapped I/O

In der aktuellen Version des Projektes ist die Erkennung und das Fuzzing von MMIO-Zugriffen nicht immer möglich. Die Zugriffe führen zwar zu Page Faults, jedoch handelt es sich entsprechend um eine virtuelle Adresse. Diese lässt nicht ohne weiteres darauf schließen, dass es sich um einen MMIO-Zugriff handelt.

Bei der Verwendung von EPT lösen MMIO-Zugriffe jedoch eine so genannte EPT Misconfiguration aus, die einen VM Exit auslöst. Somit können MMIO-Zugriffe unter der Verwendung von EPT erkannt werden. Dies hat den Vorteil, dass hier direkt die GPAs ausgelesen werden können und so der Zugriff einfacher ist.

Performance

Wie in Abschnitt 5.2 ersichtlich, ist die Performance des Projektes noch nicht befriedigend. Auch wenn die derzeitige Performance ausreichend ist, um Tests damit durchzuführen, sollte diese in Zukunft noch deutlich verbessert werden.

Die Ergebnisse der Performance-Benchmarks lassen hoffen, dass die Verwendung von EPT einen deutlichen Geschwindigkeitsvorteil bringt. Darüber hinaus existieren noch einige andere Bereiche, in denen die Performance verbessert werden könnte.

So ist zum Beispiel noch ein Performancegewinn durch die Optimierung des Trackens von Speicherpages zu erwarten. Auch die Optimierung der Schnittstelle zum Userspace und des Fuzzers im Userspace ist möglicherweise lohnenswert.

Einschränkung zu fuzzender Informationen

Derzeit wird jegliche Kommunikation zwischen dem inneren Gast und seinem Hypervisor gefuzzt. Jedoch kann es von Vorteil sein, das Fuzzing auf die Kommunikation mit bestimmter emulierter Hardware einzuschränken. Hierzu wäre es unter anderem sinnvoll, Informationen zu DMA vom DMA Controller abzufragen und so interessante Speicheradressen für bestimmte Hardware zu identifizieren. So würde zum einen speziell bestimmte emulierte Hardware getestet, zum anderen würde ein Teil der Abstürze des inneren Gastes reduziert.

Der Nachteil an dieser Erweiterung besteht darin, dass dies spezifisch für das Betriebssystem im inneren Gastes implementiert werden müsste. Somit würde für diese Tests der generische Anteil dieser Lösung geringer ausfallen. Deshalb ist die Implementierung dieser Erweiterung abzuwägen.

Literatur

[1] Sören Bleikertz. „Fuzzing the Xen Hypervisor“. 2010.
URL: https://www.openfoo.org/blog/xen-fuzz.html.

[2] Intel Corporation. „INTEL 80386 PROGRAMMER'S REFERENCE MANUAL“. 1987.
URL: http://css.csail.mit.edu/6.858/2015/readings/i386.pdf.

[3] Bandan Das, Yang Z Zhang und Jan Kiszka.
„Nested Virtualization - State of the art and future directions“.
URL: http://www.linux-kvm.org/images/3/33/02x03-NestedVirtualization.pdf.

[4] Wikimedia Foundation. „Privilege rings for the x86, along with their common uses“. 2007.
URL: https://commons.wikimedia.org/wiki/File:Priv_rings.svg.

[5] Xiao Guangrong. „KVM MMU Virtualization“. 2012. URL:
http://os.51cto.com/down/ppt/2012ckernel/07IBM%20XiaoGuangrong%20kvm-mmu.pdf.

[6] Intel. „Micro VMMs and Nested Virtualization“.
URL: http://tce.webee.eedev.technion.ac.il/wp-content/uploads/sites/8/2015/09/BC_Micro-VMMs-and-Nested-Virtualization.pdf.

[7] *Intel® 64 and IA-32 Architectures Software Developer's Manual.*
Bd. 3: System Programming Guide. 2016.

[8] „Linux kernel documentation“. In: Kap. The x86 kvm shadow mmu.
URL: https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt.

[9] „Linux kernel documentation“. In: Kap. x86_64 Memory Map.
URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[10] Gleb Natapov. „Nested EPT to Make Nested VMX Faster“. 2013.
URL: http://www.linux-kvm.org/images/8/8c/Kvm-forum-2013-nested-ept.pdf.

[11] „Nested virtualization for the next-generation cloud“.
URL: http://www.ibm.com/developerworks/cloud/library/cl-nestedvirtualization/.

[12] Open Web Application Security Project. „Fuzzing“. 2016.
URL: https://www.owasp.org/index.php/Fuzzing.

[13] „QEMU Wiki“. In: URL: http://wiki.qemu.org/Main_Page.

[14] Inc. Red Hat. „Red Hat Enterprise Linux 3: Introduction to System Administration“. 2003.
URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Introduction_to_System_Administration/s1-memory-virt-details.html.

[15] Thomas Ritzau. „QEMU, KVM, XEN & LIBVIRT“. 2011. URL: http://qemu-buch.de.

[16] A.G. Michael Sutton und P. Amini. *Fuzzing - Brute Force Vulnerability Discovery.*
Addison Wesley Pub. Co. Inc., 2007.

[17] Qinghao Tang. „Virtualization Device Emulator Testing Technology“. 2016.
URL: https://cansecwest.com/slides/2016/CSW2016_Tang_VirtualizationDeviceEmulatorTestingTechnology.pdf.

[18] Virtualbox. „Virtualbox Manual“.
URL: http://www.virtualbox.org/manual/ch10.html#idp13729504.

[19] Felix Wilhelm. *Discover software vulnerabilities.* 2015. URL: https://os.itec.kit.edu/downloads/ma_2015_wilhelm_felix__discover_software_vulnerabilities.pdf.

Alle Weblinks wurden am 15. August 2016 auf ihre Erreichbarkeit geprüft.