MinVisor: Provable Machine Protection with Optional Fidelity

Committee:

Mike Dahlin, Supervisor

Warren Hunt

Michael Walfish
MinVisor: Provable Machine Protection with Optional Fidelity

by

Michael Joseph McCoyd, B.S.

Thesis

Presented to the Faculty of the
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

BACHELOR OF SCIENCE, TURING SCHOLARS HONORS

The University of Texas at Austin

May 2012
To my parents and teachers,

who taught me so much.
Acknowledgments

I am very grateful for the mentoring and feedback from my advisor Mike Dahlin. Mike has taught me to find the key aspect of the problem and focus on solving that. I want to thank the other members of my committee, Warren Hunt and Michael Walfish, for their detailed feedback and for challenging the assumptions I had made and forgotten. Our hypervisor verification group of Mike Dahlin, Bill Young, Deepak Goel, and Robert Krug contributed constructive discussion on this work, without which it would not be where it is. As my systems teammate, Deepak Goel has contributed a great deal to the system side of this project. Deepak enabled debugging of AMD-V in QEMU, made the partial move to newer hardware, added PXE download ability to MinVisor, and is working on PXE, PCI, and memory map issues for running ttyLinux under MinVisor. I want to thank the faculty and students of the Turing Scholars Honors Program, especially Calvin Lin, who made it possible for me to join that group. It has been a great home these past few years. This work was supported by the National Science Foundation under Grant No. CNS-0917162, and by Raytheon under contract 200901296.

Michael Joseph McCoyd

The University of Texas at Austin
May 2012
MinVisor: Provable Machine Protection with Optional Fidelity

Michael Joseph McCoyd, B.S.
The University of Texas at Austin, 2012

Supervisor: Mike Dahlin

The hypervisors securing our expanding cloud services are not as secure as we would like. Vulnerabilities in resource protection have allowed unprivileged guests to change arbitrary host data. The relatively smaller size of hypervisors does not seem to guarantee their safety. Yet formal methods of mechanical program verification have recently verified a 9,300 line operating system, thus it seems plausible to verify a small hypervisor.

If we are to verify a hypervisor, verifying protection seems the place to start, and not fidelity or speed. To help hypervisor verification, I isolate the protection characteristic of a hypervisor into a separate layer, a minvisor. MinVisor runs two guests, the Fidelity and the Guest, both untrusted. It uses the guest-host barrier for protection, provides minimal device access to the Fidelity, and tasks the Fidelity with providing an accurate x86 virtual machine to the Guest. This architecture facilitates verification by providing a smaller trusted computing base (TCB) and allows change to the virtual machine fidelity without change to the verified TCB. MinVisor, with the Fidelity provided, runs a Guest that can echo input on its transparently virtualized serial port, thus providing an initial validation of the architecture. Our group is developing a formal method for mechanical verification of hypervisors and hopes to verify MinVisor’s machine code, using a proof skeleton and machine model simplification I provide. MinVisor presents a starting point for hypervisor
verification in two dimensions by providing 1) a working protection and device access layer and 2) a skeleton of how to structure its verification.

MinVisor is loaded by the AMD64 BIOS and comprises 1,890 lines of C code and 568 lines of assembly; less than 400 lines set up its protections and provide its runtime.
Contents

Acknowledgments v

Abstract vi

List of Tables xi

List of Figures xii

Chapter 1 Introduction 1

Chapter 2 Scope 6

Chapter 3 Background 8

3.1 AMD64 ISA resources . 8

3.2 AMD-V protection mechanisms . 9

3.3 AMD-V fidelity mechanisms . 14

Chapter 4 Architecture 17

4.1 Overview . 17

4.2 Protection . 18

4.2.1 Memory address space . 18

4.2.2 I/O address space and model specific registers (MSR) . 21

4.2.3 Uncleared resources . 21

4.2.4 Restoration of host state . 21

4.2.5 System management and long modes . 21

4.3 Authentic Virtual Machine . 22
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>AMD64 general programming registers and protection by AMD-V or MinVisor</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>AMD64 system programming registers and protection by AMD-V or MinVisor</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>AMD64 modes, instructions, address spaces and protection by AMD-V or MinVisor</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>AMD64 general programming registers and fidelity by AMD-V or MinVisor</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>AMD64 system programming registers and fidelity by AMD-V or MinVisor</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>AMD64 modes, instructions, address spaces and fidelity by AMD-V or MinVisor</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>MinVisor code size</td>
<td>19</td>
</tr>
<tr>
<td>7.1</td>
<td>Execution flow of VMRUN, guest, and #VMEXIT</td>
<td>40</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>MinVisor architecture</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>AMD-V hardware virtualization support</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Mappings of guest to physical memory addresses</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Runtime execution flow</td>
<td>24</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis addresses creating trusted hypervisors for our computing services. It presents the architecture and successful partial implementation of the protection aspect of a hypervisor for a simple x86-based machine. We present a proof skeleton for the mechanical verification of the protection layer, and a simplification of the machine model that may make verification easier. MinVisor’s mechanized proof appears feasible and we believe MinVisor can serve as the base for development and proof of a verified and more capable hypervisor.

Many computing service providers, such as Amazon and Rackspace, provide their clients the convenience of their own virtual machine and operating system. Hypervisors, such as VMWare and Xen [3], allow the client the illusion of their own machine, without the costs of a dedicated physical machine. The hypervisor allows multiple guest operating systems to concurrently share the resources of a single physical machine without interfering with one another. This resource sharing is used to maximize machine utilization, while the isolation property is useful in data center security. Both Amazon and Rackspace use versions of the Xen hypervisor for resource virtualization and as part of the security for their web services [1, 14].

Securing one hypervisor for the cloud is easier than securing all of the more complex operating systems run as guests. Operating system kernels are often very large. The Linux 3.2 kernel has just under of 10,000,000 code lines, with 5,600,000 lines in the driver directory and 120,000 in the kernel directory [10].\footnote{According to the SLOCCount tool [20].} The Xen 4.1.2 hypervisor has 207,000 source code lines, with 82,000 lines in the x86 arch directory, exclusive of tools, but including drivers.
Though smaller than a full size operating system, Xen is still a very large code base with a history of security vulnerabilities. A 2008 Xen exploit allowed a normal guest running a paravirtualized frame buffer to run arbitrary code in dom0, bypassing NX, ASLR and SELinux [19]. This exploit gave access to hardware resources, with the ability to read or change any disk data.

Proving the correctness of a hypervisor, instead of just relying on its relative simplicity, provides a much stronger assurance of security. Verification of a 80,000 line hypervisor is a difficult task because of the size of the code and the complexity of the machine model. The mechanical verification of a small hypervisor is plausible as formal methods tools have been found capable of proving the 9,300 line seL4 microkernel [8] based on its source code. The small NOVA [16] hypervisor has 9,000 lines of core services, and 27,000 lines of management and drivers. SecVisor [15] is smaller at 2,000 lines of code yet focuses on protecting a modified Linux guest kernel and not the machine resources from an arbitrary guest. Yet the verification of SecVisor found it to be vulnerable [7], so even such a small code base can benefit from a formal machine verification.

MinVisor was created within a project to develop a method for mechanical verification of a hypervisor based on its machine code, thus avoiding the need to trust the compiler of the hypervisor. MinVisor provides a starting point for application of that method to a code base small enough for an initial verification, and one that should soon provide useful capabilities.

Popek and Goldberg [13] define a virtual machine monitor (hypervisor) as having three essential characteristics resource control (protection), speed, and an environment essentially identical to the original machine (fidelity). Hypervisors serve different roles and thus have different design choices. A common role is the data center virtualization of resources among many guests, with creation and migration of guests. This is a focus on their usefulness, not on assurance of their security. At the other end of the spectrum is protection of the hardware from a single known malicious guest, such as in a cyber range [6] studying known malware. No matter what the role of a hypervisor, all its capabilities depend on its ability to protect itself and the machine from the guests. Proving success at protection is an essential start toward proving the safety of a hypervisor that also includes guest management and a broad offering of machine resources. Protection is also amenable to absolute proof, while speed and fidelity are matters of degree. Thus our
project simplified its goal to mechanical verification of hypervisor protection, as a first step
toward a mechanically verified hypervisor.

My role in our project has been to create a hypervisor suitable as a starting test
case for use of our verification method. I decided our initial verification would be simpler if
protection was separated from fidelity and speed. Using an architecture similar to NOVA,
fidelity was placed in an untrusted guest, while a thin layer, a minvisor, provides protection
and paravirtualized device access, Figure [1.1]

To concisely refer to MinVisor’s two guests, “guest” is used for a guest in general,
“the Guest” is used for the guest under observation or test, “the Fidelity” is used for the
guest that provides an authentic environment to the Guest. The terms “Guest” and “Fi-
delity” are capitalized titles applied to the roles two types of guests play in the architecture,
as such they should usually appear after an article. The exception to this usage is section
or column titles that are capitalized. Use of uncapsitized “fidelity” appears in phrases
such as “the machine fidelity is improved.”

Figure 1.1: MinVisor architecture

MinVisor protects itself and the machine from all guests and provides paravirtualized devices
to the Fidelity guest(s). Each Guest has its own instance of a Fidelity, on which it depends for
speed and a transparently virtualized machine. MinVisor does not create bidirectional isolation
within a Fidelity-Guest pair, as the Fidelity may need full access to the Guest. MinVisor is
intended to create isolation between Fidelity-Guest pairs, independent of any Fidelity or Guest,
but only one pair is currently supported.

MinVisor is a minimal protection layer loaded by the BIOS and running on bare
AMD-V hardware, it runs one pair of untrusted 32-bit guests, the Fidelity and the Guest. It
provides a paravirtualized serial port to the Fidelity, and supports the Fidelity in providing
a fully virtualized serial port to the paired x86-based Guest, it protects itself and the
machine resources from all guests. Protection and Guest isolation depend on MinVisor alone. Speed and an authentic virtual machine depend on MinVisor and the Fidelity paired with the Guest. MinVisor, and its boot loader, total 2,458 lines of C and assembly; the code that sets up its protections and comprises its runtime is less than 400 lines. Once its protection is verified, we intend to add additional resources, such as a network card, and support for more Guests. The verification will be advanced with each addition to MinVisor, thus building up a proven and more full featured hypervisor in stages.

MinVisor itself does not ensure that the Guest runs well; that is part of the fidelity aspect of a hypervisor, but not crucial for its security. The challenge of a hypervisor is to secure the host, and provide guest isolation, while allowing guests to run productively. This is often aided by separating the protection sensitive code, in a privileged core, from emulation or fidelity code at a lower privilege level. Some hypervisors have guests which they trust more, e.g. Xen dom0 [3], or untrusted emulation layers running within host mode but as user processes, e.g. NOVA [16]. MinVisor relies on securing just the guest-host barrier, and thus attempts to simplify its verification. The Fidelity allows a potentially large untrusted code base to provide whatever fidelity is needed from the virtual machine, without requiring an increase in the trusted computing base, once essential paravirtualized device access is provided.

Our project’s hope is to contribute a proof of MinVisor that would advance the security of hypervisors beyond assumed security. Our project hopes to provide a method of machine code verification that has proven that a small layer provides the protection characteristic of a hypervisor. Proof that MinVisor properly establishes its nested page tables was done on the y86 [5]. At that time MinVisor expected to provide all three of the hypervisor characteristics of protection, fidelity, and speed. However it had only advanced to protecting its own memory from direct access by the guest.

My contribution is the creation of a protection layer that protects itself and the machine, and provides a base for a hypervisor. I provide the skeleton of a proof that MinVisor does provide such protection, and a simplification of the machine model to facilitate such proof. I provide a simple Fidelity that demonstrates virtualization of the serial port. MinVisor and the Fidelity run on real hardware and allow a Guest to echo its serial port input with no guest access to hardware devices. MinVisor is useful now as a stepping stone to a proof of hypervisor correctness. The architecture I created should support MinVisor’s
expansion to form part of a more capable hypervisor. With the addition of a timer and network card, MinVisor could soon be useful as a host for Linux, or as a cyber range laboratory “glove box” in which to run and observe known malware.
Chapter 2

Scope

This chapter describes MinVisor’s job and places it in the larger context of what is needed for a broadly useful and mechanically verified hypervisor.

MinVisor’s job is to provide paravirtualized devices and ensure that control of the machine can always be returned to the host, in a clean state, by “pressing” a virtual “Big Red Button”, e.g. sending a special packet or entering a particular combination of keys on the keyboard, regardless of what the guest does. With an increasingly complex software stack and tool chain, it is important that we have a tool at its base that we can trust through verification. MinVisor is small enough that mechanical verification of its machine code appears feasible, thus avoiding issues with trusting the tools used to compile it [18].

If MinVisor provided the basis for a complete and mechanically verified hypervisor, it could secure our computing resources more reliably. A mechanically verified hypervisor requires efforts in two dimensions, hypervisor capability and mechanical verification. MinVisor, with its provided proof skeleton and Fidelity, provides only a start in each dimension. It is our group’s intent to expand in both these dimensions, in synchronized stages, building up a mechanically verified hypervisor that provides an authentic x86 with network access.

**Capability** We used a definition of a hypervisor as having the essential characteristics of protection, speed, and fidelity [13]. Our proof skeleton suggests that MinVisor provides the first essential characteristic, complete protection of machine resources. MinVisor only partially provides the next essential characteristic, a virtual machine “essentially identical
with the original machine" [13]. MinVisor provides an x86-based machine with only one para-virtualized device to the Fidelity. This does not give the Fidelity enough para-virtualized devices to create an x86 environment for the Guest. Yet the current MinVisor does demonstrate that the architecture supports para-virtualized devices. It seems plausible that MinVisor could expand to provide the core para-virtualized devices needed. Our group intends to add a timer, a minimal PCI bus, and a network card. These seem the key resources needed to run a Linux guest, or any arbitrary guest of comparable requirements. Speed, the last essential characteristic, depends on all hypervisor components. MinVisor’s use of minimal intercepts suggests that a minimal speed cost is possible. Speed and fidelity both depend on the Fidelity used. Our group hopes to build a MinVisor and a Fidelity providing both fidelity and speed. Yet MinVisor’s architecture supports use of any suitable Fidelity; a user provided Fidelity might provide customized fidelity, or logging, while accepting a high speed penalty.

**Verification** The proof skeleton is an outline for verification, but only that. It shows what resources must be examined, and suggests that the code is structured properly for protection, yet a complete machine model must still be developed. The formal methods side of our group is part of an effort to develop a model of the Intel AMD64 in ACL2. They will soon begin trying to verify MinVisor with their model. The provided simplification of the AMD-V VMRUN instruction may allow an initial verification that assumes a broader set of assumptions. MinVisor’s code is very cleanly separated into the sequential steps of init, protection, and runtime. A first verification could concentrate on just the protection setup and runtime as this is the code most crucial to protection. MinVisor is loaded by the BIOS and the current skeleton trusts the boot process. Loading MinVisor by a secure boot mechanism, such as Unified Extended Firmware Interface or Trusted Platform Module, would allow the proof to be stronger by relying on fewer assumptions.
Chapter 3

Background

This chapter enumerates the resources of the AMD64 ISA and summarizes the hardware virtualization mechanisms provided by the AMD-V extensions. MinVisor must protect the AMD64 ISA resources and can use the AMD-V extensions to simplify its task. The chapter describes how the hardware mechanisms support protection and fidelity.

To ensure that MinVisor protects the machine, it is helpful to enumerate the resources of the AMD64 ISA. This list can then be compared to the protections created by AMD-V and MinVisor. This chapter enumerates the AMD64 resources, the AMD-V hardware virtualization mechanisms, and MinVisor virtualization actions, but does not discuss the last. Chapter 4 will discuss MinVisor’s actions to virtualize the x86.

3.1 AMD64 ISA resources

MinVisor uses AMD64 as its host, and so must protect the AMD64 resources. The AMD64 resources [2] are listed on the left of tables 3.1, 3.2, and 3.3. This chapter only enumerates the resources, leaving their description to the manual. The AMD64 resources include processor registers, instructions, and access by the processor to memory and devices. We follow the ISA division of registers and instructions into general and system programming. For instructions, only a sample is listed of both those that are a protection concern and those with no special protection mechanisms. The tabular presentation of the resources, and the AMD-V effects on them, were created from data in the AMD manual [2], no authoritative table presentation of AMD64 resources, nor of the AMD-V effects on them,
3.2 AMD-V protection mechanisms

AMD-V provides hardware support for virtualization as an AMD64 extension, described in detail in [2]. AMD-V divides execution into host or guest mode, controlled by the VMRUN instruction and hardware intercepts. Any switch between guest and host is called *world shift*. The VMRUN instruction causes world shift from host to guest mode. A hardware *intercept* causes world shift from guest to host mode. AMD-V uses a virtual machine control block (VMCB) to hold information about a guest’s virtualization. The VMCB contains saved guest register state, information about the last intercept, control settings for instruction and other intercepts, a page table pointer, and permission bitmap pointers for the I/O address space and the model specific registers (MSR). Figure 3.1 highlights these mechanisms. What defines the guest is that if it attempts actions that have been restricted, an intercept is triggered, and the host can respond to the guest’s attempted action. Actions that might be restricted in the VMCB include access to memory, I/O ports, MSRs, and use of instructions or control registers.

Figure 3.1: AMD-V hardware virtualization support

The guest runs as a normal ring 0 or lower process. The guest start state is defined and guest actions restricted by settings in the VMCB. Red indicates machine state, here memory, affected by guest execution.
Table 3.1: AMD64 general programming registers and protection by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>ISA general registers</th>
<th>Host protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td>yes</td>
</tr>
<tr>
<td>EBX, ECX, EDX</td>
<td>yes</td>
</tr>
<tr>
<td>ESI, EDI, EBP</td>
<td>yes</td>
</tr>
<tr>
<td>ESP</td>
<td>yes</td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td>RAX to R15</td>
<td>yes</td>
</tr>
<tr>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td>RFLAGS</td>
<td>yes</td>
</tr>
<tr>
<td>Instruction ptr.</td>
<td></td>
</tr>
<tr>
<td>RIP</td>
<td>yes</td>
</tr>
<tr>
<td>Memory model</td>
<td></td>
</tr>
<tr>
<td>CS, DS, ES, SS</td>
<td>.SEL</td>
</tr>
<tr>
<td>FS, GS</td>
<td>.SEL</td>
</tr>
<tr>
<td>Streaming SIMD</td>
<td></td>
</tr>
<tr>
<td>YMM/XMM0-15</td>
<td></td>
</tr>
<tr>
<td>MXCSR</td>
<td></td>
</tr>
<tr>
<td>64bit media</td>
<td></td>
</tr>
<tr>
<td>x87 floating point</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*ISA general registers* lists the general programming registers the AMD64 ISA defines.

*AMD-V save and restore* describes AMD-V actions to save and restore the host state of registers. Registers are marked “yes” if they are restored to their saved state, on world shift to host. The 64-bit registers are marked N/A as guests are denied access to them. Four segment registers are restored, and if necessary their descriptors re-read.

*MinVisor restore or clear* describes MinVisor actions to protect registers not restored by AMD-V. Host registers are cleared if the guest could have changed them and AMD-V does not restore them. The general extension registers are ignored in this thesis and marked “ignored”.

*Reference AMD-V/code* lists where the AMD manual describes the protection action or MinVisor code creates it.

*Source:* ISA and AMD-V data adapted from [2].
Table 3.2: AMD64 system programming registers and protection by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>Category</th>
<th>ISA system registers</th>
<th>Host protection</th>
<th>Reference AMD-V/code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>CR0, CR3, CR4, CR2, CR1, CR5-7, CR8-CR15, XCR0</td>
<td>AMD-V save and restore yes</td>
<td>MinVisor restore or clear =0 protect.c</td>
</tr>
<tr>
<td>Descriptor tables</td>
<td>GDTR, IDTR, LDTR</td>
<td>descriptor =.SEL</td>
<td></td>
</tr>
<tr>
<td>Process level</td>
<td>CPL</td>
<td>yes (0) =.SEL</td>
<td>15.5.1,15.6 entry.S</td>
</tr>
<tr>
<td>Debug</td>
<td>DR0 to DR3, DR4, DR5</td>
<td>AMD-V save and restore yes</td>
<td>MinVisor restore or clear =0 protect.c</td>
</tr>
<tr>
<td>Task</td>
<td>TR</td>
<td>=.SEL</td>
<td>entry.S</td>
</tr>
<tr>
<td>MSR: all</td>
<td></td>
<td>MSR: control EFER yes</td>
<td>MSR: system linkage KernelGsBase STAR, LSTAR, CSTAR, SFMASK, SYSENTER_[CS,ESP,EIP] no guest write access 15.11, protect.c</td>
</tr>
</tbody>
</table>

Columns follow the format of table 3.1.

AMD64 defines but reserves CR1, CR5-7, DR4, and DR5. As they are reserved, they are neither restored by AMD-V nor cleared by MinVisor.

AMD-V requires the CPL be zero to use VMRUN; “restore” always sets it to zero. AMD-V provides a permission bitmap mechanism for the Model Specific Registers (MSRs) with read and write bits. An attempt to access a MSR whose bit is set causes a hardware intercept.

MinVisor denies long mode to the guest, thus CR8-15 and XCR0 can not change. MinVisor sets intercepts for all guest MSR writes.

*Source:* ISA and AMD-V data adapted from [2].
Table 3.3: AMD64 modes, instructions, address spaces and protection by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>Category</th>
<th>Detail</th>
<th>AMD-V mechanism</th>
<th>MinVisor settings</th>
<th>Reference AMD-V/code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating mode</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMM</td>
<td>real, v8086, protected long</td>
<td>CR0 intercept</td>
<td>static trust of SMM code</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR intercept</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Virtualization mode</strong></td>
<td>guest</td>
<td>instr. intercept</td>
<td>deny guest by VMRUN</td>
<td>svm.c</td>
</tr>
<tr>
<td>host</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>General purpose instructions</strong></td>
<td>ADD, CALL, ... CPUID</td>
<td>none</td>
<td></td>
<td>B.1</td>
</tr>
<tr>
<td><strong>System programming instructions</strong></td>
<td>CLGI VMCALL VMRUN</td>
<td>intercept</td>
<td></td>
<td>B.1, svm.c</td>
</tr>
<tr>
<td><strong>Memory address space</strong></td>
<td>RAM</td>
<td>mapping</td>
<td>to vanilla RAM</td>
<td>15.25, nested.c</td>
</tr>
<tr>
<td>ISA bus</td>
<td>mapping</td>
<td>to vanilla RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCI bus</td>
<td>mapping</td>
<td>to vanilla non-RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-RAM</td>
<td>mapping</td>
<td>to vanilla non-RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I/O address space</strong></td>
<td>Devices</td>
<td>permission</td>
<td>no guest access</td>
<td>15.10, protect.c</td>
</tr>
</tbody>
</table>

The ISA and reference columns follow the format of table 3.1. Only a sample of instructions is shown.

**AMD-V mechanism** describes the mechanisms AMD-V provides to virtualize resources. Intercepts can prevent entry to protected and long modes. A VMRUN instruction intercept can prevent creation of a nested virtual machine. A nested page table can control the mapping of guest memory addresses to machine physical addresses. Nested page faults cause a hardware intercept. A permission bitmap for the I/O address space can control access per address. An attempt to access an I/O address whose bit is set causes a hardware intercept.

**MinVisor settings** describes how MinVisor uses these mechanisms. It assumes trust of the SMM code, including that it can not be changed and it does not damage the machine or MinVisor. The guest is denied access to long mode and the creation of a nested virtual machine. Guest memory access is mapped away from sensitive RAM and the ISA and PCI buses. All I/O addresses are marked for intercept.

**Source:** ISA and AMD-V data adapted from [2].
AMD-V provides a mechanism for saving and restoring machine registers associated with the host or guest when world shift occurs. A page, pointed to by a MSR, stores the recent host register values, the VMCB stores the recent guest register values. On world shift to the guest, several machine registers are saved to the host save area and guest values are loaded from the VMCB into the machine registers. On world shift to the host, several machine registers are saved to the VMCB and host values are restored from the host save area to the machine registers. For the host, these registers are indicated in tables 3.1 and 3.2. Restoring these registers helps protect host mode execution, and the machine. Blank table cells under AMD-V reflect registers where the state created by the guest remains intact in the host mode, and where action may be needed to prevent unwanted influence by the guest on host execution.

AMD-V protects memory with an extra page table that is used after any guest address translation and before access to a physical memory address. Page faults in this nested page table create a world shift to host. The nested page table can be used to create a mapping of the guest memory address space away form sensitive host RAM or device buses. Each guest has its own nested page table. For each guest, recently used mappings are cached in the translation lookaside buffer. The nested page tables of modern hardware make it easier to create a hypervisor by removing the challenge of managing shadow page tables.

AMD-V protects the I/O address space with a bitmap that indicates if access to an address, and thus a device, should trigger an intercept. AMD-V protects the AMD64 model specific registers (MSR) with a bitmap with two bits, read and write permissions, per register. If the permissions are violated, the hardware generates an intercept, and control returns to host mode. Information about the attempted access is stored in a host data structure. This allows the host to block access, or to provide some virtual access. AMD-V protects some control registers and instructions with intercept flags in the VMCB. Enabling them causes an intercept on any use of the instruction or read/write of the control register. Table 3.3 summarizes the AMD-V virtualization mechanisms for modes, instructions, and address spaces.
3.3 AMD-V fidelity mechanisms

In addition to protection support, AMD-V provides mechanisms to help create an accurate virtual machine for the guest. AMD-V divides guest state management between VMRUN/#VMEXIT, which restore and save the core guest registers, and VMLOAD/VMSAVE which restore and save a few less common registers. The effect of world shift and VMLOAD/VMSAVE on guest resources is summarized in tables 3.4, 3.5, and 3.6. Blank cells under the guest reflect resources where the provision of an authentic machine to the guest may suffer if guest state is not saved and restored in software. AMD-V’s virtualization of real mode simplifies the Fidelity task of creating a virtual machine that includes real mode.

Table 3.4: AMD64 general programming registers and fidelity by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>ISA general registers</th>
<th>Authentic x86 virtual machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>State</td>
</tr>
<tr>
<td>32-bit</td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td>EBX, ECX, EDX</td>
</tr>
<tr>
<td></td>
<td>ESI, EDI, EBP</td>
</tr>
<tr>
<td></td>
<td>ESP</td>
</tr>
<tr>
<td>64-bit</td>
<td>RAX to R15</td>
</tr>
<tr>
<td>Flags</td>
<td>RFLAGS</td>
</tr>
<tr>
<td>Instruction ptr.</td>
<td>RIP</td>
</tr>
<tr>
<td>Memory model</td>
<td>CS, DS, ES, SS</td>
</tr>
<tr>
<td></td>
<td>FS, GS</td>
</tr>
<tr>
<td>Streaming SIMD</td>
<td>YMM/XMM0-15</td>
</tr>
<tr>
<td></td>
<td>MXCSR</td>
</tr>
<tr>
<td>64bit media</td>
<td>MMX0-7</td>
</tr>
<tr>
<td>x87 floating point</td>
<td>FPR0-7</td>
</tr>
<tr>
<td></td>
<td>FCW, FSW, FTW</td>
</tr>
<tr>
<td></td>
<td>RIP, RDP, Opcode</td>
</tr>
</tbody>
</table>

Columns follow the format of Table 3.1.

AMD-V describes AMD-V actions to save and restore the guest state of registers.

MinVisor describes MinVisor actions to save and restore the guest state of the remaining registers.

Source: ISA and AMD-V data adapted from [2].
Table 3.5: AMD64 system programming registers and fidelity by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>ISA system registers</th>
<th>Authentic x86 virtual machine</th>
<th>AMD-V save and restore</th>
<th>MinVisor save and restore</th>
<th>Reference AMD-V/code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR0, CR3, CR4</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>CR2</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>CR1, CR5-7 (reserved)</td>
<td></td>
<td>no guest access</td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>CR8-CR15</td>
<td></td>
<td></td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>XCR0</td>
<td></td>
<td></td>
<td>protect.c</td>
<td></td>
</tr>
<tr>
<td>Descriptor tables</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDTR, IDTR</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>LDTR</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.14</td>
</tr>
<tr>
<td>Process level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPL</td>
<td>yes*</td>
<td></td>
<td></td>
<td>15.5.1, 15.6</td>
</tr>
<tr>
<td>Descriptor tables</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDTR, IDTR</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>LDTR</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.14</td>
</tr>
<tr>
<td>Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.14</td>
</tr>
<tr>
<td>MSR: all</td>
<td></td>
<td></td>
<td>no guest write access</td>
<td>15.11, protect.c</td>
</tr>
<tr>
<td>MSR: control</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFER</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.5.1</td>
</tr>
<tr>
<td>MSR: system linkage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KernelGsBase</td>
<td>yes</td>
<td></td>
<td></td>
<td>15.14.4</td>
</tr>
<tr>
<td>STAR, LSTAR, CSTAR, SFMASK, SYSENTER_[CS,ESP,EIP]</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Columns follow the format of table 3.1.

AMD-V restores the guest CPL unless real mode is set, in which case it is forced to 0.

MinVisor fails to restore the DR0 to DR3 guest registers. The guest can read but not write MSR, which is not authentic.

Source: ISA and AMD-V data adapted from [2].
### Table 3.6: AMD64 modes, instructions, address spaces and fidelity by AMD-V or MinVisor

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>AMD-V mechanism</th>
<th>MinVisor settings</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating mode</td>
<td>SMM real, v8086, protected</td>
<td>CR0 intercept</td>
<td>no guest access</td>
<td>svm.c</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>MSR intercept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtualization mode</td>
<td>guest</td>
<td>instr. intercept</td>
<td>no guest access</td>
<td></td>
</tr>
<tr>
<td>General purpose</td>
<td>ADD, CALL, ...</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td>CPUID</td>
<td>intercept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System programming</td>
<td>CLGI</td>
<td>intercept</td>
<td>intercept</td>
<td>B.1, svm.c</td>
</tr>
<tr>
<td>instructions</td>
<td>VMCALL</td>
<td>intercept</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VMRUN</td>
<td>intercept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory address space</td>
<td>RAM</td>
<td>mapping</td>
<td>mapped to vanilla RAM</td>
<td>15.25, nested.c</td>
</tr>
<tr>
<td></td>
<td>ISA bus</td>
<td>mapping</td>
<td>none provided</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCI bus</td>
<td>mapping</td>
<td>none provided</td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-RAM</td>
<td>mapping</td>
<td>mapped to vanilla non-RAM</td>
<td></td>
</tr>
<tr>
<td>I/O address space</td>
<td>Devices</td>
<td>permissions</td>
<td>COM1 virtualized</td>
<td>15.10, protect.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>other write is no-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>other read returns 0</td>
<td></td>
</tr>
</tbody>
</table>

Columns follow the format of table 3.3.

The guest is provided no ISA or PCI bus and only a single device.

*Source:* ISA and AMD-V data adapted from [2].
Chapter 4

Architecture

This chapter describes how MinVisor uses the AMD-V mechanisms to provide protection and an environment for the Fidelity to create an authentic virtual machine for its Guest.

4.1 Overview

MinVisor’s job is to provide protection and enough paravirtualized devices to support the Fidelity. MinVisor does not worry about providing an authentic view of the hardware to the guests. MinVisor provides a paravirtualized machine to the untrusted Fidelity. It is up to the Fidelity to provide a fully virtualized machine to the Guest. MinVisor only: (A) protects itself and the machine, (B) provides minimal paravirtualized devices so the Fidelity can do its job, (C) passively passes messages back and forth between the Fidelity and the Guest, and (D) interposes on a small number of virtualized I/O channels so that an external message can return control of the machine. MinVisor uses the hardware host mode for protection and alternates between running the pair of guests as needed.

Booting

The BIOS loads MinVisor over the networked pre-execution environment (PXE), though disk boot is also available. The BIOS downloads a 32KiB image containing MinVisor’s boot-loader and ELF image. MinVisor’s boot-loader uses PXE to download two 32KiB images, for the Fidelity and the Guest, staging them after it in low memory. The boot-loader expands MinVisor’s ELF image to a starting point at 1MiB, and jumps to

\footnote{MinVisor’s use of PXE to download its guests is work by Deepak Goel}
the ELF’s entry point. After creating its own execution environment and page tables, MinVisor copies the guest images to the RAM addresses which will become 0x7C00 in the guests address space once nested paging is in effect, thus mimicking their PXE download. MinVisor then enters the code which establishes protections and finally enters its runtime loop of executing the guests. MinVisor trusts its boot process and any SMM code. It is expected that the boot process and MinVisor use a secure management network to download images. The Guest is assumed to use a physically separate network, once network access is implemented in MinVisor and the provided Fidelity.

Table 4.1 shows the code size of MinVisor’s components. The protection and runtime code are listed separately from the rest as their dependence on the init code is primarily just for the creation of a protected mode execution environment. For an initial verification, separation of the protection and runtime from the init code may simplify the proof by allowing the init code to be replaced by its post conditions.

4.2 Protection

To protect the hardware and itself, MinVisor uses the protections offered by AMD-V, but it must fill the remaining gaps. Setting the hardware requires: creating nested page tables that restrict guest memory addresses away from sensitive RAM or buses, denying direct access to the I/O address space, denying write of MSRs, and enabling AMD-V and the above protections. Fixing the gaps just requires clearing a few registers that are not returned to the host state on intercept.

4.2.1 Memory address space

The RAM that both guests can directly and indirectly affect is important for both protection and the provision of fidelity to the guest.

Address space protection Figure 4.1 shows the nested page table mappings that protect the machine and MinVisor from guest address space access. Both guests have access to only vanilla RAM, RAM whose address has no special meaning, and have all non-RAM addresses mapped to a single 2MiB non-RAM address range with no extra privileges, such as being the PCI bus. With no direct access to devices, protection of RAM from DMA access by devices is not needed.
### Table 4.1: MinVisor code size

<table>
<thead>
<tr>
<th>Directory</th>
<th>asm</th>
<th>C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot loader</td>
<td>350</td>
<td>38</td>
<td>388</td>
</tr>
<tr>
<td>inc</td>
<td>1224</td>
<td>1224</td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td>215</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>init</td>
<td>109</td>
<td>153</td>
<td>262</td>
</tr>
<tr>
<td>protect</td>
<td>184</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>runtime</td>
<td>109</td>
<td>76</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>568</td>
<td>1890</td>
<td>2458</td>
</tr>
<tr>
<td>Fidelity</td>
<td>71</td>
<td>148</td>
<td>219</td>
</tr>
<tr>
<td>Guest</td>
<td>121</td>
<td>61</td>
<td>182</td>
</tr>
</tbody>
</table>

Lines separate the boot loader, init, and protection sensitive code.

Half of the boot loader assembly is downloading the guests with BIOS PXE.

Include has x86 and AMD-V structure definitions.

Lib has high level printing and serial methods, and C wrappers of the ‘inb’ and “outb” instructions.

Init sets up MinVisor as an executable, and does most of the fidelity setup for the guests.

Protect sets up the protections. It uses, from inc and lib, two macros to format page table entries, a macro to flag physical addresses, two wrappers to read and write model specific registers, and debug print methods.

Runtime uses debug print methods from lib and definitions from svm.h.

The provided Fidelity and the experimental Guest use several include files and string printing and serial port management from lib. The provided Fidelity uses its own VMACALL implementations for “inb” and “outb” wrapper methods, The experimental Guest uses lib’s normal instruction wrapper implementation.

Sizes are according to the SLOCCount tool.
Direct RAM access  Nested paging maps the “physical” addresses of each guest onto the indicated RAM area and the non-RAM page. The Guest sees only its area while the Fidelity sees its and the Guest’s.

Indirect RAM effects  The Guest execution causes MinVisor to modify its stack and local variables during intercept processing. A concern is whether this can cause MinVisor to damage the machine or corrupt its protections. An example of state that is indirectly changed by guest execution is that VMEXIT stores the state of many machine registers into the guest save area of the VMCB and places the cause of the intercept into the EXITINFO area of the VMCB, thus both are modified by guest action. The effects of such guest changes to RAM are limited as MinVisor does not examine any guest state except for the details of VMCALLs.

Figure 4.1: Mappings of guest to physical memory addresses

The guest addresses that are intended to be guest RAM have linear maps to machine RAM that is away from sensitive machine addresses. All other guest address pages are mapped to a single machine page that is above RAM and has no special properties. The arrows indicate how representative guest pages are mapped onto machine pages. The Fidelity has 512MiB RAM, while the Guest has 256MiB RAM, as the Fidelity must have RAM for its own code and have access to the Guest’s RAM. The address spaces are 64GiB as the nested page tables use 2MiB physical address extension (PAE) pages. The guests may use PAE if they wish.
4.2.2 I/O address space and model specific registers (MSR)

Access to both the I/O address space and write access to model specific registers is denied to the guests with permission bit maps that are common to both guests. Denial of the I/O address space removes concerns for device DMA transfers accessing sensitive addresses. Denial of write access to MSRs restricts the guests from entering long mode or modifying other machine parameters.

4.2.3 Uncleared resources

There is a risk that changes by the guests to general or system registers could cause the host to take an action with undesired effects on the machine. To reduce this risk, MinVisor zeroes the guest’s changes to machine registers before processing the guest intercept. MinVisor clears or restores all registers not already cleared of guest state by the hardware, except for the general programming extension registers. This is shown in tables 3.1 and 3.2 with “=0” or “=.SEL” under “MinVisor”. The code clearing the registers does not read them. While MinVisor does not read most of these resources, clearing them makes the limited effects of guest execution on MinVisor more apparent.

4.2.4 Restoration of host state

Tables 3.1 and 3.2 suggests that, between AMD-V and MinVisor, a host defined state is restored to all AMD64 general purpose and system registers, except the extension and reserved registers. Table 3.3 suggests that guest memory access is mapped to machine addresses that can not damage the machine or MinVisor, and guest access to the I/O space, and thus guest access to devices, is denied. Thus what remains to be restored to a completely host defined state is just the memory of the guests, their VMCBs, and the device registers. This can easily be achieved when the host has control of the machine if we have limited guest access to devices.

4.2.5 System management and long modes

System management mode (SMM) has not been examined in depth. The manual describes that SMM can not be changed to use a different area of RAM from within SMM. As the boot SMM code is assumed to be part of our static trust, it is presumed that the
SMM code does not allow for such changes. An alternative is that appendix D of the AMD manual discusses secure SMM containerization using a SMM virtual machine.

Entry to long mode is denied to the guest by denying write access to MSRs, including EFER.LME. Long mode was restricted to limit the resources under guest control and thus to simplify the task of securing the machine and host from it, it is not required by the MinVisor architecture.

### 4.3 Authentic Virtual Machine

Though we have provided protection, MinVisor must still provide support for the Fidelity to provide an authentic virtual machine. To do this we allow the Fidelity to see the Guests RAM and provide the Fidelity with the information about each intercept of the Guest. The Fidelity is given access to the Guest’s RAM to allow emulation of any DMA transfers or to access any information needed for emulation, such as the current instructions opcode and arguments. Figure 4.1 shows the relationship between the Fidelity and the Guest address spaces within the machine address space. To assist the Fidelity in providing an authentic environment, the MinVisor setup code creates the following RAM layout and interrupts:

- BIOS service routines and data are in the low memory of each guest.
- Images for each guest are in their RAM areas with starting code at 0x7C00.
- All guest non-RAM addresses are mapped to a non-RAM range on the machine which should provide normal “memory bus without RAM” behavior to the guest.
- Each VMCB is set to start guest execution in real mode at 0x7C00.
- The Fidelity VMCB is set to intercept the VMCALL instruction.

**Execution pattern** Execution starts first with the Fidelity which sets any state needed and signals ‘done’, causing the first run of the Guest. Figure 4.2 shows the pattern of guest and host execution once the Guest is started.

**Devices** MinVisor provides minimal access to a serial port, paravirtualized for the Fidelity, transparently virtualized for the Guest. As the I/O address space and the ISA and PCI buses are inaccessible to the guest, devices are accessible only through VMCALLs to
the host. Only the COM1 serial port is virtualized. Access to COM1 is normally through
the “inb” and “outb” instructions to the I/O addresses 0x3F8 to 0x400, yet MinVisor
blocks all direct guest access to the I/O address space. Instead, COM1 must be accessed
using VMCALLs by the Fidelity on behalf of the Guest.

COM1 access is initiated by the Guest’s attempted access to a COM1 I/O address.
The intercept this generates is passed by MinVisor to the Fidelity. On examining the
EXITINFO in its copy of the VMCB, the Fidelity determines that the Guest is trying to
access the COM1 port. The Fidelity handles this by accessing its paravirtualized COM1
port. The COM1 access by the Fidelity is shown in Figure 4.2. In a struct in the Fidelity
RAM, the Fidelity stores the direction, port, and data of the COM1 use, places in eax the
value signaling a COM1 call, and does VMSCALL. Once control returns to the host, eax is
examined to determine that the intent of the of the VMSCALL is for COM1 access. If the
port is outside the COM1 range, control is returned to the Fidelity without affecting any
devices. Otherwise, the al and dx registers are set as needed for the COM1 access, ‘inb’ or
‘outb’ is executed by the host as needed, any incoming value is copied to the struct in the
Fidelity, and control is returned to the Fidelity.

This is the pattern of device access that MinVisor supports, of the Fidelity doing
management of the Guest’s device access and MinVisor providing only primitive device
access, and thus maintaining a smaller trusted computing base. By protecting itself and
the machine, providing paravirtualized device access, and a connection between the Fidelity
and the Guest, MinVisor is designed to provide a protection layer on which a hypervisor
can be built.
Both guests sit on top of the AMD-V hardware assisted virtualization and MinVisor. Memory affected by the Guest or the Fidelity execution is colored red or gray respectively. The “Intercept, register and inject state” box is red though it holds both data that the Guest generates and which is copied into Fidelity by MinVisor, and data that Fidelity changes or generates. Similarly the “The Guest” box is red but could have its data changed by the Fidelity. For clarity within the host, the save and restore of the Fidelity state is not shown.

The left of the figure shows the interaction of the Fidelity and Guest. The 1) intercept of a Guest protection violation causes the hardware to save the Guest register state and intercept exit information, in the VMCB. MinVisor then 2) copies the intercept details and the Guest register states into the Fidelity RAM. The Fidelity is then 3) run and examines the cause of the intercept, maybe accesses paravirtualized devices, maybe modifies the Guest RAM or the copy of the Guest register state, and returns control to the hypervisor with 4) a VMCALL ‘done’. MinVisor 5) copies the modified Guest register state and injection details back into the Guest VMCB without affecting protections, and uses 6) vmrun to cause the hardware to restore the altered Guest state and resume Guest execution. MinVisor uses fixed addresses within the Fidelity for these copy operations. It does not dereference any addresses provided by the guests.

The right of the figure shows the Fidelity paravirtualized driver architecture. In processing the Guest intercept, the Fidelity can make use of paravirtualized devices through a VMCALL interface. The a) VMCALL ‘device’ causes b) a check that the requested I/O port access is to COM1, c) copy of data from a Fidelity data structure to register eax, d) device access and external Guest communication, e) copy of any result back to the Fidelity RAM, and f) return to the Fidelity. MinVisor uses fixed addresses within the Fidelity for these copy operations. The only address provided by a guest that MinVisor accesses is the port provided in the I/O device VMCALL.

The bottom left of the figure shows an external signal which will cause an intercept and arbitrary host processing such as clearing all guest effects on the machine. Handling of this external signal is not yet implemented.
Chapter 5

Experiments

Experiments were run to see whether MinVisor and a Fidelity guest can provide the intended execution environment to a Guest, and thus form a hypervisor if MinVisor’s protection is proven. These experiment include work by Deepak Goel to setup QEMU to properly support SVM virtualization and to setup a PXE server for the real hardware.

5.1 Setup

Host machines Experiments of MinVisor were run natively on a Dell PowerEdge T105 with an AMD Quad-Core Opteron 1300, 8GiB RAM, a host network card, and a serial port. Experiments were also run on QEMU 15.0 running on an AMD FX-8150 with Asus Sabertooth 990FX motherboard, and 16GiB RAM. The machines were booted respectively with real or simulated PXE. QEMU presented to its guest (MinVisor) the architecture of QEMU’s host.

Virtualized machine The machine presented to the Guest was an x86 whose only device is a serial port. The authenticity of the machine was provided by a simple Fidelity which paravirtualized COM1 access, ignored all other I/O writes, and returned an appropriate bit-length zero for all other I/O port reads. The simulated machine ignores model specific register (MSR) write attempts. Thus its authenticity satisfied basic execution needs but was far from complete.

The Fidelity The Fidelity was a simple protected mode program I created that accesses the para-virtualized serial port when passed an intercept of the Guest trying to access
**The Guest**  The Guest was a simple protected mode program that polls the serial port and echoes twice any characters received.

### 5.2 Results

On real and QEMU hardware, the Guest echoes twice each character received on the machine serial port with no perceived response time lag. Logs show the intended execution.

### 5.3 Discussion

MinVisor and the provided Fidelity succeeded in providing some basic authenticity to a Guest at no perceivable speed decrease. That result suggests that if MinVisor is proven to provide protection it and a more developed Fidelity could form a hypervisor for a very simple machine.
Chapter 6

Proof Skeleton

This chapter outlines the skeleton of a proof that an external restore signal sent to the serial port restores the machine to a completely host defined state except for the general programming extension registers for SSE, 64-bit Media, and x87 Floating-Point. The external signal has not been implemented so the last conjecture of the skeleton is incomplete. The skeleton links the MinVisor implementation with the overall group goal of providing a hypervisor that has been formally proven correct. Such a formal proof would require a very detailed description of the ISA of the machine and far more precision than the informal skeleton presented here. The point of the skeleton is not to provide a proof but to describe how examination of the ISA and the machine code generated from the MinVisor source might be used by a formal and much more precise and detailed mechanically checked proof.

Two key assumption of the skeleton are that the machine implements the ISA, and that permanent change to the machine is not possible given access to only general and system registers, vanilla RAM, and the serial port.

The source files for the protect and runtime directories are referenced throughout the skeleton and are included in Appendix A. The entry to the protect code is init_protect(), which ends by calling start_runtime(). Thus there is no chance for other MinVisor code to mess up the protections before the guests are run. The minimal dependencies of the protect and runtime code are described after each source include statement and in table 4.1.

The outline of the skeleton is as follows.

1. Protection is based on the intercepts and memory maps created by the setup code.
2. The resources that must be cleared of guest effects are simplified by denying the guest

27
the use of long mode.

3. On any intercept, the effects of the guest are cleared from all core registers.

4. This makes it easier to establish that MinVisor’s runtime is minimally affected by guest execution.

5. That and the first point show that the intercepts and memory maps are preserved in the face of guest and runtime execution.

6. Finally, one could show that an external signal can clear the remaining effects of guest execution, absent the extension registers which this work has ignored for simplicity.

These points are expanded in more detail at the beginning of each of the following sections. Each section develops a conjecture which contributes to the overall skeleton. The conjectures have the same number as their section and the above outline.

A foundation of the skeleton is the AMD ISA defined by the AMD64 Architecture Programmer’s Manual [2]. Several parts of the ISA are crucial to any proof that MinVisor protects itself and the machine resources. These are stated in the skeleton as axioms with reference to their defining section in the manual. When they appear, that section is split into ISA and MinVisor subsections.

### 6.1 Setup Intercepts and Memory Mapping

The first step is to set data structures so the hardware will do as much as it can for us. The hardware allows us to direct all guest memory access away from sensitive areas and intercept guest execution when a range of dangerous actions are attempted. Exploiting this hardware capability reduces the software and proof effort.

The code to do so is simple. Two CPUID queries test that the hardware provides the needed support; setting a model specific register (MSR) bit enabling SVM. Filling two arrays with ‘1’ or “01” creates the permission bit maps. The nested page tables are more complex. Each page gets the same set of flags. Each table has one range that has a linear mapping, and a second where all guest pages are mapped to the same host page. For a bit of space efficiency, the second range reuses identical page directory tables. Finally, the VMCB points to these tables and sets the bits that enable them. The VMCB also sets the
bits to intercept the hardware interrupt and the CGIF instruction. According to the ISA, these steps set the hardware for the memory map and intercepts we want.

6.1.1 ISA

Definition 1. System sensitive memory addresses are: system BIOS in low memory 0-1MiB, VGA device in low memory 0-1MiB, ISA bus at 15-16MiB [12], PCI bus at 3-4GiB [17].

Axiom 1. If the guest’s VMCB enables MSR intercepts, an intercept is generated when the guest attempts to write to any MSR whose write, odd, bit is set in the MSR permission bit map (Vol. 2 section 15.11).

Axiom 2. If the guest’s VMCB enables I/O intercepts, an intercept is generated when the guest attempts to access any port whose bit is set in the I/O permission bit map (Vol. 2 section 15.10.1).

Axiom 3. If the guest’s VMCB enables nested paging, guest memory addresses access only host memory addresses mapped to by the nested page table (Vol. 2 section 15.25.1).

Axiom 4. If the guest’s VMCB enables an interrupt intercept, an intercept is generated when that interrupt occurs (Vol. 2 section 15.13).

Axiom 5. If the guest’s VMCB enables an instruction intercept, an intercept is generated when the guest attempts to execute that instruction (Vol. 2 section 15.9).

6.1.2 MinVisor

Proposed Lemma 1. The CPU is AMD and supports nested paging.

Argument. By mechanical analysis of init.c check_hardware_support().

Proposed Lemma 2. SVM is enabled in EFER.SVME.

Argument. By mechanical analysis of svm.c init_svm().

Proposed Lemma 3. An 8KiB bitmap at msr_perm_bitmap marks each odd bit set. It is aligned on a 4KiB boundary, in 0-15MiB, and disjoint from code and other data.

1 The location and size of the PCI hole depends on the BIOS and the demands of the installed cards. Our system has only a small PCI video card and PCI NIC.
Argument. By mechanical analysis of svm.c init.msrpm().

**Proposed Lemma 4.** A 12KiB bitmap at io_perm_bitmap marks each bit set. It is aligned on a 4KiB boundary, in 0-15MiB, and disjoint from code and other data.

Argument. By mechanical analysis of svm.c init.iopm().

**Proposed Lemma 5.** MinVisor code and data are in 0-15MiB, exclusive of data read or written in Fidelity.

Argument. By BIOS load and mechanical analysis of ELF layout init/init.ld, and remaining code.

**Proposed Lemma 6.** Nested page tables at fidelity_pdpt and guest_pdpt map all guest memory access away from 0-16MiB and 3-4GiB. They are in 0-15MiB and disjoint from code and other data.

Argument. By mechanical analysis of protect.c init_protect() and nested.c create_nested_pt().

**Proposed Lemma 7.** The hardware has a serial port at I/O addresses 0x3F8 to 0x3FF.

Argument. By mechanical analysis of init.c i386_init(), lib/serial.c.

**Proposed Lemma 8.** The 4KiB VMCBs at physical addresses fidelity_vmcb_pa and guest_vmcb_pa are aligned on 4KiB boundaries, in 0-15MiB, disjoint from code and other data, and:

1. Enable MSR intercepts and point to msr_perm_bitmap.
2. Enable IO intercepts and point to io_perm_bitmap.
3. Enable nested paging and point respectively to fidelity_pdpt and guest_pdpt.
4. Enable intercept of the hardware interrupt.
5. Enable intercept of the Clear Global Intercept Flag instruction.

Argument. By mechanical analysis of entry.S, protect.c init_protect(), and svm.c init_vmcb_protect().

**Conjecture 1.** A guest run with the VMCB at fidelity_vmcb_pa or guest_vmcb_pa:
1. Is intercepted on any attempt to write to MSRs.

2. Is intercepted on any attempt to access I/O ports.

3. Can not access MinVisor or sensitive system memory addresses using the CPU.

4. Is intercepted on any serial port interrupt.

5. Is intercepted on any attempt to clear the global interrupt flag.

Argument. By proposed lemmas 1, 2, and respectively:

1. Axiom 1, proposed lemmas 3, 8
2. Axiom 2, proposed lemmas 4, 8
3. Definition 1, axiom 3, proposed lemmas 5, 6, 8
4. Axiom 4, proposed lemmas 7, 8
5. Axiom 5, proposed lemma 8

6.2 Deny Guest Long Mode Resources

It is simpler to clear guest execution effects from system resources if we prevent the guest from entering long mode and accessing the larger set of long mode registers. This is an implementation expediency and not essential to the architecture. It could be removed to make the proof simpler.

The MSR EFER.LME bit must be set before long mode can be entered. Each guest save area clears this bit, and VMRUN then copies that value into the guest. As guests are denied write access to MSRs, it can not be changed. The section concludes with a list of the registers the guests may not change as a result.

6.2.1 ISA

Axiom 6. VMLOAD followed by VMRUN sets the below registers and begins guest mode execution. VMEXIT followed by VMSAVE saves the below registers. Both cases use the the guest save area of the VMCB pointed to by eax when VMRUN is called.
1. General programming: RAX, RSP, RFLAGS, RIP, CS, DS, ES, SS, FS, GS.


3. Model specific: EFER, KernelGsBase, STAR, LSTAR, CSTAR, SFMASK, SYSENTER.[CS,ESP,EIP]

(Vol. 2 sections 15.5.1, 15.14.4)

Axiom 7. To enable long mode, MSR EFER.LME must be set to 1. (Vol. 2 section 14.6).

Axiom 8. Long mode is required to access RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15, RFLAGS, RIP.

6.2.2 MinVisor

Proposed Lemma 9. MinVisor setup sets each VMCB with EFER.LME=0. MinVisor runtime does not change the VMCB EFER.

Argument. By mechanical analysis of protect.c init_protect(), svm.c init_vmcb_protect(), runtime/entry.S, intercepts.c.

Proposed Lemma 10. Fidelity and Guest can not access long mode.

Argument. MSR EFER.LME must be 1 to enter long mode, by axiom 7. MinVisor sets it to 0 in each VMCB, by proposed lemma 9. As MinVisor data, the guest can not change it, by conjecture 1. It is copied into the guest, by axiom 6. As a MSR, the guest can not change it, by conjecture 1.

Conjecture 2. Fidelity and Guest can not change: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15, RFLAGS, RIP.

Argument. By axiom 8 and proposed lemma 10.

Corollary. Fidelity and Guest might change: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, EIP.

Argument. Conjecture 2 does not restrict guest access to these corresponding 32-bit resources.
6.3 Runtime Clears Registers of Guest Execution Effects

To make a proof simpler, it is helpful to show that while MinVisor is running, the registers MinVisor might normally use have been cleared of guest execution effects. This also establishes an intermediate level where most guest effects have been cleared, on which an “all guest effects cleared” state can be built.

Clearing depends first on the host save area memory and MSR being protected from the guest, the above conjectures argue that they are. Immediately after #VMEXIT and VMSAVE, MinVisor assembly code unconditionally clears the remaining registers of interest. The code for this, between both guests, is 120 lines of non branching assembly. This leaves guest effects in only guest RAM, structures about guests, the serial port, and some extension registers.

6.3.1 ISA

Definition 2. The ISA defines AMD64 resources as:

1. General programming registers: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15, RFLAGS, RIP, CS, DS, ES, FS, GS, SS.
2. General programming extension registers: SSE registers, 64-bit Media registers, x87 Floating-Point registers.
4. Model specific registers: EFER, and others.
5. Memory address space.
6. I/O address space.

(Vol. 1 sections 2, 3.8, Vol. 3 section 2.3)

Axiom 9. VMRUN stores the host value of the below registers before loading guest state. #VMEXIT restores the value of the below registers after saving guest state. Both cases use the page pointed to by the MSR VM_HSSAVE_PA.

1. General programming: RAX, RSP, RFLAGS, RIP, CS.SEL, DS.SEL, ES.SEL, SS.SEL.

3. The model specific: EFER.

(Vol. 2 sections 15.5.1).

**Corollary.** The AMD64 registers not restored by #VMEXIT are:

1. General programming: RBX, RCX, RDX, RSI, RDI, RBP, R8-R15, FS, GS.

2. General programming extensions: Streaming SIMD, 64bit Media, x87 Floating point.

3. System programming: CR2, CR8, DR0-DR3, DR6, DR7, LDTR, TR.

4. Model specific: all except EFER.

**Axiom 10.** #VMEXIT stores in the VMCB intercept exit information and decode assist bytes describing the guest execution state. (Vol. 2 section 15.6)

### 6.3.2 MinVisor

**Proposed Lemma 11.** MSR VM_HSAVE_PA, the physical address of the host save area page, points to a 4KiB page in 0-15MiB that is disjoint from code and other data. The runtime code does not directly modify the contents of that page.

**Argument.** Inspection of kern/entry.S, svm.c init_host_save() and runtime/entrysvm.S and intercepts.c.

**Proposed Lemma 12.** On intercept, following VMSAVE and before further processing, MinVisor unconditionally saves the guest values of: EBX, ECX, EDX, ESI, EDI, and EBP to a guest register structure. MinVisor then unconditionally clears: EBX, ECX, EDX, ESI, EDI, EBP, FS, GS, CR2, DR0 - DR3, DR6, and DR7. MinVisor unconditionally sets: LDTR and TR.

**Argument.** By mechanical analysis of runtime/entry.S.

**Conjecture 3.** Before processing a guest intercept, the hardware and MinVisor have cleared all resources of guest execution effects except:

1. VMCB areas for: guest save area, intercept exit information, decode assist bytes describing guest execution state.
2. The guest register save structure.

3. General programming extension registers.

4. RAM mapped onto by the guest nested page table.

5. Any guest data in the serial port registers.

**Argument.** Of the resources in definition 2, those stored by VMRUN have been restored to their previous host values as the HSAVE area and its MSR were inaccessible to the guest, by axiom 9, proposed lemma 11, and conjecture 1. The guest could not modify long mode registers nor MSRs, by conjectures 1 and 2. MinVisor cleared the remaining CPU registers, by proposed lemma 12, except the general programming extension ones. Thus, only the listed resources retain effects of guest execution.

### 6.4 Runtime Minimally Affected by Guest Execution

The last step for normal execution is to show that MinVisor takes only very limited and safe actions as a result of guest execution. For a proof, this is the most crucial code as it is the only code that alternates with guest execution, aside from the blind assembly code actions to save, clear, and restore guest state on intercept.

All Guest intercepts are handled by blindly copying state into Fidelity and running Fidelity. The only Fidelity intercepts the code expects are use of the paravirtualized serial port to support the Guest and the signal that Fidelity is done and Guest should be run again. The default for all other Fidelity intercepts in to halt the machine. The C code in question is 90 modular single threaded lines with two switch statements, `VMCALL|other` and `done`|`device`|other, and two if statements, `port = COM1|other and inb|outb`. Its provisional trust might be reasonable after a detailed inspection.

**Proposed Lemma 13.** The runtime code uses only the VMCBs at `fidelity_vmcb_pa` and `guest_vmcb_pa` to run guests.

**Argument.** By mechanical analysis of entry.S.

**Conjecture 4.** The execution of guests affects the MinVisor runtime only by:

1. The occurrence of a Guest intercept.
2. The occurrence of a Fidelity intercept VMCALL with Fidelity eax signaling ‘done’

3. The occurrence of a Fidelity intercept VMCALL with Fidelity eax signaling ‘device’, and:
   (a) A Fidelity data structure listing an I/O port.
   (b) A Fidelity data structure signaling in or out.

4. The occurrence of any other intercept.

   Based on these it decides respectively to:

1. Copy the Guest register save structure and the VMCB into Fidelity and run Fidelity.

2. Copy the Guest register save structure and the VMCB event injection and non-MSR parts of the guest save area back to the Guest’s VMCB, and run the Guest.

3. Process a Fidelity serial port access.
   (a) Allow COM1 access if the port is in the range of COM1, or ignore the intercept.
   (b) Read or write a COM1 register.

4. Halt the machine.

Argument. By proposed lemma 13, conjecture 3, to simplify the proof, and mechanical analysis of runtime/entriesvm.S and intercepts.c.

6.5 Runtime and Guests Preserve Intercepts and Memory Maps

Having listed the actions the runtime takes, it is important that a proof point out that the runtime does not change the structures that establish protection, nor the MinVisor runtime code. Thus the cycle of execution can safely repeat as often as desired.

Conjecture 5. The guests and the runtime code do not modify the structures, created by the setup code, on which conjecture 7 depends.

Argument. By conjectures 3 and 4.
6.6 External Signal Clears Guest Effects on Machine

Any proof needs to show that an external signal can always bring the machine to host mode and clear it of guest execution effects. From the ISA, an intercept can be generated on external input, and conjecture[1] set that up. Normal normal intercept processing will clear most guest effects, excepting general programming extension registers. From there it is a matter of the runtime wiping the RAM and device register traces of the guest. This is unimplemented, but not a complex process.

6.6.1 ISA

Axiom 11. The ability to do the below does not allow permanent change to the machine.

1. Read and write general and system programming registers.

2. Read and write vanilla RAM and a memory address range away from all RAM and busses.

3. Read model specific registers, but not write them.

4. Read and write I/O addresses for the COM1 serial port.

6.6.2 MinVisor

Proposed Lemma 14. The runtime reads the serial port on intercept of a serial port interrupt.

Argument. By mechanical analysis of runtime/intercepts.c.

*****UNIMPLEMENTED*****

Proposed Lemma 15. When read of the serial port shows ten consecutive escape characters, the runtime resets the serial port and erases Fidelity and Guest: RAM, VMCBs, and register save structures.

Argument. By mechanical analysis of runtime/intercepts.c.

*****UNIMPLEMENTED*****
Conjecture 6. A external restore signal sent to the serial port restores the machine to a completely host defined state, except for general programming extension registers (SSE registers, 64-bit Media registers, x87 Floating-Point registers).

Argument. By conjectures 1 and 3 and proposed lemmas 14 and 15.
This chapter presents a simplification of the AMD-V ISA that may make an initial verification of MinVisor easier by simplifying the machine model required.

Although MinVisor greatly simplifies the code that must be analyzed to get a proof of hypervisor correctness, smaller and simpler code is not the end goal. Instead, once we can prove things about MinVisor we want to start adding new features and evolve it towards a more full featured but mechanically verified hypervisor. Although our approach has successfully simplified the code by isolating protection in a minvisor, the hardware remains complex. In fact, developing a sufficient model of the hardware for formal analysis is the main barrier to formally analyzing MinVisor at this point. Just as we have taken an iterative approach to the software - start small and then add features - it appears desirable to take an iterative approach to the hardware model.

Given the MinVisor design, a significant simplification to the hardware model is possible that still appears likely to yield a useful model for developing proofs about MinVisor. A difficulty in proving MinVisor’s protection is how to handle arbitrary guest execution. A full model of the transition to guest mode, the full range of guest execution, and return to host mode would require a detailed model of the virtualization hardware and all instructions available to the guest. Approaching all of this in a first proof may delay what could be a useful first approximation available by simplifying guest execution to a set of random resource changes and reflecting these in the hardware without modeling the AMD details of the shift to guest mode. This simplification is particularly beneficial in our project as the larger x86 modeling effort that our group is working alongside is focused on the Intel x86
hardware, while MinVisor development started with and has remained on AMD AMD64 hardware.

In terms of protection, the details of how the guest gets to a given changed machine state are not important. Modeling every sequential path of guest execution until the next intercept would be painful with a fixed guest, it would be infeasible for all hypothetical guests. Thus instead of modeling guest execution precisely as sequential operations, the guest might be modeled as a random change to a set of the resources it can reach. If no such set of random changes harms the machine or MinVisor, then protection has been successful. This approach allows making a claim for an arbitrary guest.

This model assumes that a chain of guest execution can be represented by a single change to the final machine resource state that the chain produces. This model would not represent well a timing based hardware attack based on a sequence of actions with specific delays between, all before the next intercept. But there is not direct guest access to devices with Minvisor, given the nested page tables and I/O permission bit maps. Thus, the assumption may be reasonable for a first proof.

The transition to and return from guest mode is defined in the ISA by the VMRUN instruction. The return phase of VMRUN uses the term #VMEXIT, borrowing the “#” exception notation. Execution of the guest occurs between these two parts of the VMRUN. The normal flow of VMRUN, guest execution, and #VMEXIT is show on the left half of table 7.1

Table 7.1: Execution flow of VMRUN, guest, and #VMEXIT

<table>
<thead>
<tr>
<th>ISA</th>
<th>“Do something random” guest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save some host register state to MSR page</td>
<td>same</td>
</tr>
<tr>
<td>Enable nested paging</td>
<td>same</td>
</tr>
<tr>
<td>Load some registers from VMCB</td>
<td>same</td>
</tr>
<tr>
<td>Jump to next guest instruction</td>
<td>n/a</td>
</tr>
<tr>
<td>Execute until intercept and #VMEXIT</td>
<td>Randomize registers/RAM</td>
</tr>
<tr>
<td>Save some guest register state to VMCB</td>
<td>Randomize VMCB guest save</td>
</tr>
<tr>
<td>Disable nested paging</td>
<td>same</td>
</tr>
<tr>
<td>Reload some host registers from MSR page</td>
<td>same</td>
</tr>
<tr>
<td>Execute host instruction following VMRUN</td>
<td>same</td>
</tr>
</tbody>
</table>

The left column summarizes the execution flow defined by the ISA. The right column describes what parts might be simplified if we treat guest execution as random resource changes. The line groupings are for VMRUN, guest execution, and #VMEXIT, in order.
The right column of table 7.1 shows how VMRUN might be simplified with a random execution guest model. To model guest execution, the random change must respect the restrictions created by the VMCB. These limit the resources the random change can reach. In the case of MinVisor, the guest is restricted to a specific range of RAM, can not write MSRs, can not access I/O ports, and can not modify long mode resources. Thus the randomization can only be applied to the remaining resources accessible by the CPU. A random advance of the clock should be included in these resources, to represent time spent on guest execution. As I/O ports are blocked, devices are not accessible.

To reflect #VMEXIT, the guest save and exit info areas of the VMCB should be modified. In reality this would reflect the corresponding machine registers, but that correspondence is included in the random change to both. A version of the VMRUN instruction with the simplifications from table 7.1 is shown in appendix A following the format of the AMD manual.

Further simplifications could be argued for, but risk defining away the problem. First is to ignore the need in the model for nested paging support, restricting the random RAM changes to only RAM mapped onto by the nested page tables. The second is to restrict resource changes to those that are not restored from the host save area, as the host save area page and the MSR pointing to it can not be changed under MinVisor’s restrictions.
Chapter 8

Design Choices and Lessons Learned

This chapter presents general observations from the development of MinVisor.

8.1 Design Choices

Real mode guest start Starting the guest in real mode as if by the BIOS simplified the interface between host and guest, by simplifying the initial conditions we needed to establish for the guest, and allowed a broader range of guests.

PXE boot PXE boot was Initially chosen to avoid MinVisor trace on the hard drive. With no guest hard drive, PXE boot matches what we want the guest to see. It and serial port prints make testing on real hardware simpler.

Fidelity Guest Prevention of guest access to sensitive machine resources stops all but the simplest of guests from running. The Fidelity allows the response to attempts to access sensitive resources to be created in a way that supports the Guest’s normal or expected execution without that code being in the protected base. MinVisor must provide only limited mediated access to devices to the Fidelity to support this.

x86 Guests The guests were limited to the X86 architecture to simplify protection of the host. Allowing an x86 guest seemed to provide a reasonable opportunity for attack,
while eliminating worry for the larger reach of long mode execution. Removing this guest restriction seems straightforward and could simplify the proof.

**Clearing guest state** A conservative approach is taken to protecting MinVisor from influence by the guest. The guest/host mode shift leaves a few resources holding copies of guest state. While MinVisor’s design does not access these resources, it is safer to clear them. This allows a simpler argument for MinVisors security, and some protection against implementation errors.

### 8.2 Lessons Learned

**Verification intuitions may be wrong** Intuitions on what would be easier to model in a proof may be wrong. The SecVisor code I started with walked the page table structure in creating them, in that it uses each link in the top level table to access and initialize the lower pages. This is a very general approach and make the location of each lower pages irrelevant. I simplified the code by assuring that the pages used for the lower level of the page table were contiguous, and just iterated through them as a group. This did not make the proof simpler, instead it broke the nice correspondence of how the page tables were constructed and how they were used, which had allowed one logical model to be used for both. It was not a big issue but an example of how tight software design and a verification effort may prefer different decisions.

**Creating a formal specification finds bugs** Developing the proof skeleton forced reexamination of the source and the cleanup of a few areas where issues had been missed or glossed over. First, the Fidelity was able to modify the Guests EFER flags and thus put the Guest in long mode despite the intended claim the long mode was denied to the Guest. Second, I had provided nested mappings for only 4GiB of guest address space, so it would be false to claim that a valid nested page map was always available.

**Build in stages** MinVisor started as the JOS boot loader and has had small pieces added in stages. The hardest part may have been getting any real mode guest to just execute after the VMRUN instruction. That step was taken building on a guest known to boot from the BIOS, host code known to successfully print to the serial port, and SecVisor’s
VMCB code for a protected mode guest. Solving one isolated problem at a time simplified development.
Chapter 9

Related Work

MinVisor differs from related work primarily in running only one Guest, providing just a protection layer, using just the guest/host barrier for protection, and in having a much smaller runtime trusted computing base.

**SecVisor**  SecVisor \cite{15} provided a starting point for our groups verification effort. The verification of its design with the Murϕ model-checker provided an example of hypervisor verification. Because of that verification, we started with SecVisor as the test case for our work. SecVisor’s design verification was based on a higher level abstraction of the code than the machine code verification our group is pursuing. SecVisor’s design goal is to protect the guest’s kernel from malicious code, and thus has more dependency on the guest than desired for a focus on protecting the hardware resources from arbitrary guests. Its use was dropped in favor of building up MinVisor from what had been learned from SecVisor. SecVisor begins its own execution after the boot loader and runs a modified Linux kernel on AMD hardware. It provides memory protection with either shadow or nested paging. SecVisor has a larger runtime code base than MinVisor. Its nested paging implementation has source lines of: initialization 729, header code 376, and runtime 1,030.

**NOVA**  NOVA \cite{16} runs multiple unmodified guests and places only core services in host kernel mode, with policies and drivers in host user mode. NOVA provides a separate user mode emulation module to each guest. MinVisor takes a similar approach, but places such emulation in the Fidelity. This reduces MinVisor’s protection and verification tasks to just the guest/host barrier, and not also the user/system one. NOVA runs on Intel VT or AMD-V, and provides for fully virtualized guests. NOVA has a host kernel mode of 9,000
lines of core services, and a host user mode with 27,000 lines of management and drivers.

**Palacios**  Palacios [9] is an OS independent hypervisor running on Linux and other operating systems. It runs unmodified guests on Intel or AMD virtualization hardware and is designed for high performance computing on top of a lightweight kernel. It is a highly configurable architecture working on top of different host operating systems and presenting different virtual machine configurations to guests. Palacios also moves the guest start point earlier to system reset, an approach that would allow a malicious guest to modify the “machine” BIOS and reboot. Similar to MinVisor, Palacios preallocates guest memory as physically contiguous regions, simplifying the virtual memory implementation. Palacios has a core of 15,000 lines and 9,000 lines of drivers.

**Related code**  MinVisor has benefited from reading or using code from several projects. JOS [4] provided MinVisor’s Makefile structure and the boot loader from which MinVisor was build up. SecVisor provided the initial nested page table creation and protected mode VMCB initialization. Xen [3] and Palacios provided general hypervisor examples. OSDev.org [11] provided real mode and I/O address space code and discussions on many issues. Kitten [9] provided some kernel example code.
Chapter 10

Conclusion

MinVisor advances the effort to create a mechanically verified hypervisor by providing an initial working implementation of a protection layer and a skeleton outline of its verification. MinVisor, as an implementation of the protection aspect of a hypervisor, has a proof outline, backed by the ISA, that suggests it provides protection of the ISA hardware and itself. A simplification of the ISA instruction VMRUN have been presented which may make a first proof more accessible by reducing the size of the machine model that must be created before proof begins. MinVisor’s Fidelity architecture allows changes to the authenticity of the virtual machine of a hypervisor without requiring reproof of its protection. MinVisor, and its provided Fidelity, run a simple protected mode Guest that echoes input on its transparently virtualized serial port. MinVisor is small enough for formal mechanical verification, with a total size of 2458 lines of C and assembly, and protection that primarily depends on less than 400 lines. MinVisor shows promise for near term utility as a server host for Linux if a timer and network card are virtualized.
Appendix A

Protection and Runtime Code

This appendix lists the full source code of the protection and runtime directories of Minvisor. Once this code is entered, it is the only host code that runs, absent a few small utility methods such as debug prints.

A.1 Protection

`.src/protect/cpuid.h`

```c
#ifndef CPUID_H
#define CPUID_H

static __inline void cpuid(uint32_t info, uint32_t *eaxp, uint32_t *ebxp, uint32_t *ecxp, uint32_t *edx)
{
    uint32_t eax, ebx, ecx, edx;
    asm volatile("cpuid"
                 : "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx)
                 : "a" (info));
    if (eaxp)
        *eaxp = eax;
    if (ebxp)
        *ebxp = ebx;
    if (ecxp)
        *ecxp = ecx;
    if (edxp)
        *edxp = edx;
}
#endif // CPUID_H
```

`.src/protect/protect.h`

```c
#ifndef PROTECT_H
#define PROTECT_H

// protect.c

void init_protect(void);
```

48
void create_nested_pt(u64* pdpt, u64* pdt, u32 addr_start, u32 ram_size);

/* Setup SVM protections */
#define include <inc/types.h> // svm structs
#define include <inc/svm.h>
#define include <inc/paging.h> // PAGE_SIZE_4K
#define include <inc/memlayout.h> // ADDR_RAM
#define include <inc/api_fidelity.h> // ADDR_FIDELITY_GUEST_VMCB

#define include <protect/protect.h> // create_nested_pt()
#define include <protect/svm.h> // svm protect declares
#define include <runtime/inline.h> // memmove() for fidelity

// Guest struct addresses
extern u32 guest_vmcb_label[];
extern u64 guest_pdpt[]; // top of tables
extern u64 guest_pdt[]; // 'mid' level table entries

// Fidelity struct addresses
extern u32 fidelity_vmcb_label[];
extern u64 fidelity_pdpt[]; // top of tables
extern u64 fidelity_pdt[]; // 'mid' level table entries

// Shared by both VMCB
extern u8 msr_perm_bitmap[];
extern u8 io_perm_bitmap[];

// Runtime
extern void start_runtime(void);

/* Memset helper. */
static void inline memset(void *v, int c, size_t n) {
  asm volatile("cld; rep stosb\n"
     : "D" (v), "a" (c), "c" (n)
     : "cc", "memory");
}

/* Setup protections and start.
 * Confirm hardware, setup SVM protections, start runtime.
 */
void init_protect(void)
{
  // Check hardware support
  check_hardware_support();

  // Setup host
  init_svm();
  init_host_save();

  // Setup nested page tables
  create_nested_pt(guest_pdpt, guest_pdt,
                   ADDR_GUEST_START, RAM_GUEST);
/* Setup nested paging. Shifts RAM addresses, map rest to empty bus. Only handle 32 bit PAE address space, so index space for 64GiB. */

#include <inc/types.h>
#include <inc/paging.h>  // pae_make_[pdpe|pde_big](), _PAGE, PAE,
#include <inc/memlayout.h>  // _pa(), ADDR_NORAMSPEC

// = = = = = = = = = = = = = Code = = = = = = = = = = = = =
void create_nested_pt(u64* pdpt, u64* pdt, u32 addr_start, u32 ram_size)
    __attribute__((section(".init_text")));

/* Map GUEST_RAM (<= 1GiB) to vanilla DRAM, map rest to one non RAM page. 
PRE: pdpt and pdt: exist, in .bss, disjoint from other things.
   pdt size = 64 * 8 bytes
   pdt size = 2 * PAGE_SIZE_4K  bytes

POST: pdpt: all valid, marking all directories present, reusing second pdt
   [63]  0x xxx 1001
 ... [1]  0x xxx 1001
   [0]  0x xxx 0001

  pdt:
  2 GiB
    | foo0 0000 foo0 00E7 (map all pages to same non RAM page)
    | foo0 0000 foo0 00E7
    | foo0 0000 foo0 00E7
    | foo0 0000 foo0 00E7

  1 GiB
    | foo0 0000 foo0 00E7
    | 0000 0000 0140 00E7
    | 0000 0000 0120 00E7

  */
```c
void create_nested_ppt(u64* pdpt, u64* pdt, u32 addr_start, u32 ram_size) {
    // TODO assert(ram_size <= ADDR_1GB);
    const u64 pdpt_flags = (u64)(PAGE_PRESENT); // 0x001
    pdpt[0] = pae_make_pdpe(.pa((u32)pdt), pdpt_flags);
    u32 i = 1;
    for (; i < PAE_PTRS_PER_PDPT; i++){
        pdpt[i] = pae_make_pdpe(.pa((u32)pdt + PAGE_SIZE_4K), pdpt_flags);
    }

    /* Page directories */
    const u64 flags_pdt = (u64)(PAGE_PRESENT | PAGE_RW | PAGE_USER | // USER or get intercepts.
                               PAGE_ACCESSSED | PAGE_DIRTY | // As will be eventually.
                               PAGE_PSE); // Point to 2MiB page.

    /* First pdt: 0 - 1GiB
    Guest RAM shifted up, non-RAM all mapped to one 2MiB non-RAM page. */
    u64 addr = 0; // u64 to compare w/ 4GiB
    #define range_to(below, entry) 
    for (; addr < below; addr += PAGE_SIZE_2M) { 
        pdt[addr>>PAGE_SHIFT_2M] = entry; 
    }
    range_to(ram_size, pae_make_pde_big(addr + addr_start, flags_pdt));
    range_to(ADDR_1GB, pae_make_pde_big(ADDR_NONRAMSPEC, flags_pdt));

    /* Second pdt: 1GiB -> 64GiB
    Mark the one, reused, pdt as pointing all addresses to the same page. */
    range_to(ADDR_2GB, pae_make_pde_big(ADDR_NONRAMSPEC, flags_pdt));
}
```

```c
#define SVMPROTECT_H
#define SVMPROTECT_H

// In protect/svm.c
void check_hardware_support(void) { ((section ("init.text")))
void init_host_save(void) { ((section ("init.text")))
void init_svm(void) { ((section ("init.text")))
void init_vmcb_protect(struct vmcb_struct* vmcb, u32 asid, u64* pdpt) { ((section ("init.text")))
#endif // SVMPROTECT_H
```

```c
/* Establish VMCB protections. */
#include <inc/types.h>
#include <inc/svm.h> // svm structs, bit names
#include <inc/memlayout.h> // .pa()
#include <inc/msr.h> // rdmsr(), wrmsr()
#include <inc/serial.h> // printf(), INFO()
```
#include <protect/cpuid.h>  // cpuid()

// ===Globals===
extern u32 host_save[];

// Shared by both VMCB
extern u8 msr_perm_bitmap[];
extern u8 io_perm_bitmap[];

// Locals
void init_vmcb_control_area(struct vmcb_struct* vmcb, u32 asid, u32 pdpt_pa);

/* Check required feature, halting if not present to ensure protection. */
#define CPU_CHECK(name, condition) 
   printf(name);
   if (condition) 
      printf("......[ok]\n");
   else {
      printf("......[failed]\n");
      asm__("hlt");
   }

void check.hardware.support(void)
{
    u32 eax, edx, ecx, ebx;

    /* check if AMD CPU */
    cpuid(0x00, &eax, &ebx, &ecx, &edx);
    CPU_CHECK("AMD CPU", ((ebx == AMD_STRING_DWORD1) &&
        (ecx == AMD_STRING_DWORD2) &&
        (edx == AMD_STRING_DWORD3)));

    /* check for extended instruction support */
    cpuid(0x80000000, &eax, &ebx, &ecx, &edx);
    CPU_CHECK("extended instruction", (eax >= 0x80000001));

    /* check if CPU supports SVM extensions */
    cpuid(0x80000001, &eax, &ebx, &ecx, &edx);
    CPU_CHECK("svm", (ecx & (1<<ECX_SVM)));

    /* check for nested paging support */
    cpuid(0x8000000a, &eax, &ebx, &ecx, &edx);
    CPU_CHECK("npt", (edx & (1<<EDX_NP)));
}

/* Setup host save area. */
void init_host_save(void)
{
    u64 hsave_pa;
    u32 eax, edx;

    /* Initialize the HSA */
    hsave_pa = (u64)_pa((u32)host_save);
    eax = (u32)hsave_pa;
    edx = (u32)(hsave_pa >> 32);
    wrmsr((u32)MSR_HSAVE_PA, eax, edx);
INFO("SVM HSA set up\f\r" );
}

void init_svm ( void ) {
    u32 eax , edx ;
    /*
    * Set up SVM hardware
    * SVM Control register
    *
    * Bits:   Sec SK Sec Min
    * Bits:   Load INIT Boot Visor
    * 0 DPD : 0 1 1 (1 - Disable hardware Debug Tool)
    * 1 R_INIT : 1 0 0 0 (0 - INIT normal) TODO or 1 for startup
    * 2 DIS_A20M: 0 1 0 0 (0 - allow normal masking if quest wants )
    * 3 LOCK : (not mess with for now)
    * 4 SVM_DIS : 0 0 (0 - ensure we can enable)
    * VM_RUN requires EFER_SVME set, which requires SVM_DIS clear.
    */
    rdmsr (( u32 )MSR_VM_CR, &eax , &edx );
    eax &\= ~ (0x00000016 );
    eax |\= 0x00000001 ;
    wrmsr (( u32 )MSR_VM_CR, eax , edx );

    /* SVM Enable in EFER – Extended Features Enable Register */
    rdmsr (( u32 )MSR_EFER, &eax , &edx );
    eax |\= (1 << EFER_SVME );
    wrmsr (( u32 )MSR_EFER, eax , edx );

    INFO("SVM: MSR \& EFER enabled\n" );
}

/*/ Initialize vmcb protections. */
void init_vmcb_protect( struct vmcb_struct* vmcb , u32 asid , u64* pdpt ) {
    vmcb->efer &= ~EFER_LME; // prevent entry to long mode
    init_vmcb_control_area (vmcb , asid, _pa ((u32)pdpt ));
}

/*/ Initialize vmcb control: protection of secvisor from guest. */
void init_vmcb_control_area( struct vmcb_struct* vmcb , u32 asid , u32 pdpt_pa ) {
    vmcb->cr_intercepts = 0;
    vmcb->dr_intercepts = 0;
    vmcb->exception_intercepts = 0;
    vmcb->general1_intercepts = 0;
    vmcb->general1_intercepts |\= (u32) (GENERAL1_INTERCEPT_IOIO_PROT |
                                GENERAL1_INTERCEPT_MSR_PROT );

    /* intercept all SVM instructions */
    vmcb->general2_intercepts |\= (u32) (GENERAL2_INTERCEPT_VMMCALL |
                                GENERAL2_INTERCEPT_VMMLOAD |
                                GENERAL2_INTERCEPT_VMSAVE |
                                GENERAL2_INTERCEPT_STGI |
                                GENERAL2_INTERCEPT_CLGI |
                                GENERAL2_INTERCEPT_SKINIT |
A.2 Runtime

/* Host code for entry and exit of guest mode. */
#include <inc/min_desc.h>  // SEL_NULL, SEL_TSS
#include <inc/mmu.h>      // STS_T16A

#define ENTRY(x) \
    .globl x;\
    .align ; \
    x:

/* Clear host values from general registers, for info security. */
/* not eax and esp as these are replaced by vmcb. */
#define CLEAR_HOST_GPR6 \
    xor  %ebx , %ebx ; \
    xor  %ecx , %ecx ; \
    xor  %edx , %edx ; \
    xor  %edi , %edi ; \
    xor  %esi , %esi ; \
    xor  %ebp , %ebp ; \

#define CLEAR_HOST_SEG2 \
    xorw %bx , %bx ; \
    movw %bx , %fs ; \
    movw %bx , %gs ; \

#define CLEAR_HOST_CR2 \
    xorl %ebx , %ebx ; \
    movl %ebx , %cr2 ; \

#define CLEAR_HOST_DRS5 \
    xorl %ebx , %ebx ; \
    movl %ebx , %dr0 ; \
    movl %ebx , %dr1 ; \

movl %ebx, %dr2 ;
movl %ebx, %dr3 ;
movl %ebx, %dr6 ;
movl %ebx, %dr7 ;

#define RELOAD_LDTR 
movl $SEL_NULL, %eax ;
1ldt %ax ;

/* Clear the TSS descriptor type busy bit, reload into TR */
((struct ldtss_desc *)mygdt)[SEL_TSS >> 3].type = STS_T16A;

#define RELOAD_TR 
mov mygdt + SEL_TSS + 5, %al ;
and $0x1f, %al ;
or $STS_T16A, %al ;
mov %al, mygdt + SEL_TSS + 5 ;
mov $SEL_TSS, %eax ;
1tr %ax ;

/* the order to save registers is important because we */
* reply on it to interpret some instruction during */
* interception .
* more details reference visor/intercepts-xxx.c */

// vmrun/exit handles eax, esp. Copies here are really the hosts....
#define GU_SAVE_ALL_GREGS 
movl %eax, guest_regs ;
movl %ecx, guest_regs + 4*1 ;
movl %edx, guest_regs + 4*2 ;
movl %ebx, guest_regs + 4*3 ;
movl %ebp, guest_regs + 4*5 ;
movl %esi, guest_regs + 4*6 ;
movl %edi, guest_regs + 4*7 ;

#define GU_RESTORE_ALL_GREGS 
movl guest_regs , %eax ;
movl guest_regs + 4*1, %ecx ;
movl guest_regs + 4*2, %edx ;
movl guest_regs + 4*3, %ebx ;
movl guest_regs + 4*5, %ebp ;
movl guest_regs + 4*6, %esi ;
movl guest_regs + 4*7, %edi ;

#define FI_SAVE_ALL_GREGS 
movl %eax, fidelity_regs ;
movl %ecx, fidelity_regs + 4*1 ;
movl %edx, fidelity_regs + 4*2 ;
movl %ebx, fidelity_regs + 4*3 ;
movl %ebp, fidelity_regs + 4*5 ;
movl %esi, fidelity_regs + 4*6 ;
movl %edi, fidelity_regs + 4*7 ;

#define FI_RESTORE_ALL_GREGS 
movl fidelity_regs , %eax ;
movl fidelity_regs + 4*1, %ecx ;
movl fidelity_regs + 4*2, %edx ;
movl fidelity_regs + 4*3, %ebx ;
movl fidelity_regs + 4*5, %ebp ;
movl fidelity_regs + 4*6, %esi ;
movl fidelity_regs + 4*7, %edi ;

55
/* Init:
 * Seed guest registers with values expected for BIOS boot.
 * For security we clear all guest registers to zero.
 */

/* On entry to guest:
 * No need to save host gprs, as contents not needed across vmrun.
 */

/* Note: On VMEXIT,
 * Hardware restores host eax, eip, esp, cs, ds, ...
 * We save guest registers.
 * Plus host eax, guest esp and eax already in vmcb from vmexit.
 * Use host stack for saving gpregs.
 * NOTE: Do not save MMX or XMM regs since these are not used by host.
 */

.section " .text"
/* Initial runs of fidelity and the guest are the same as later ones.
 * "Restoring:’ from bss zeroed values, clears the initial host values.
 */

ENTRY(start_runtime)

/* Intercept loop: Interleave running fidelity and guest. */
ENTRY(entry_top)

// Fidelity
/* Load extra state, and restore gregs. */
c-lgi
FI_RESTORE_ALL_GREGS
mov $fidelity_vmcb_label, %eax

/* Enter */
vmload
vmrun /* On intercept, return to execute next instruction below. */
vmsave /* Save extra state. */

FI_SAVE_ALL_GREGS /* Save guest general registers */

/* Clear guest effects from host */
CLEAR_HOST_GPR6
CLEAR_HOST_SEG2
CLEAR_HOST_CR2
CLEAR_HOST_DRS5
RELOAD_LDTR
RELOAD_TR
st-gi

/* Handle Fidelity intercept (by copy state to Guest). */
call handle_fidelity_intercept

/* If handle returned 0, run Guest. Else return to Fidelity. */
cmp $0, %al
jnz entry_top

// Guest
/* Load extra state, and restore gregs. */
c-lgi
GU_RESTORE_ALL_GREGS
mov $guest_vmcb_label, %eax

/* Enter */
vmload
vmrun /* On intercept, return to execute next instruction below. */
vmsave /* Save extra state. */

GU_SAVE_ALL_GREGS /* Save guest general registers */

/* Clear guest effects from host */
CLEAR_HOST_GPR6
CLEAR_HOST_SEG2
CLEAR_HOST_CR2
CLEAR_HOST_DRS5
RELOAD_LDT
RELOAD_TR
stgi

// Handle intercept (by copy state to fidelity)
call handle_guest_intercept

jmp entry_top

../../../src/runtime/inline.h

/* Inline helper functions. */
#endif INLINE_H
#define INLINE_H

/* Copy between disjoint ranges. Blindly assumes ranges are disjoint. */
static inline void
memmove(void *dst, const void *src, size_t n) {
    asm volatile("cld; rep movsb\n" :: "D" (dst), "S" (src), "c" (n) : "cc", "memory");
}
#endif // INLINE_H

../../../src/runtime/intercepts.c

/* Handle Fidelity and Guest intercepts. */

#include <inc/types.h>
#include <inc/serial.h>  // INFO()
#include <inc/pageing.h>  // PAGE_SIZE_4K
#include <inc/svm.h>     // svm structs and bit names
#include <inc/memlayout.h>  // ADDR_FIDEL_START
#include <inc/api_fidelity.h>  // ADDR_FIDEL_VFIDEL_VMEXIT_
#include <runtime/inline.h>  // memmove()

// External labels
extern u32 fidelity_vmcb_label[];
extern u32 guest_vmcb_label[];
extern u32 guest_regs[];

// Fidelity vmcb and VMCALL structure to choose execution path
struct vmcb_struct* fidelity_vmcb = (struct vmcb_struct*) fidelity_vmcb_label;
struct serial_call* io =
(struct serial_call*) (ADDR_FIDEL_START + ADDR_FIDEL_SERIAL_CALL);

// Copies of Guest resource state in Fidelity
struct vmcb_struct* guest_vmcb_copy =
(struct vmcb_struct*) (ADDR_FIDEL_START + ADDR_FIDEL_GUEST_VMCB);
void* const guest_greg_copy = (void*) (ADDR_FIDEL_START +
ADDR_FIDEL_GUEST_GREG);

// Fidelity handler helpers
int handle_vm_call();
void handle_vm_call_done();
void handle_vm_call_device();

// Guest vmcb to support save-area changes by Fidelity
struct vmcb_struct* guest_vmcb = (struct vmcb_struct*) guest_vmcb_label;

/* ———— Fidelity intercept handling ———— */
/* Handler for Fidelity intercepts, based on exit codes. 
   Return 0 if Fidelity is done, 1 if should rerun Fidelity. */
int handle_fidelity_intercept() {
    switch (fidelity_vmcb->exitcode) {
        case VMEXIT_VMCALL: return handle_vm_call();
        default:
            INFO("Host: Halting from non VMSCALL from fidelity.");
            __asm __volatile("hlt");
            return 1;  // retry Fidelity (and halt)
    }
}

/* Handle VMSCALL: Ready or Device access. */
int handle_vm_call() {
    // Advance Fidelity instruction past VMSCALL
    fidelity_vmcb->rip += 3;

    switch (fidelity_vmcb->rax) {
        case FIDEL_VMCALL_READY: handle_vm_call_done(); return 0;
        case FIDEL_VMCALL_SERIAL: handle_vm_call_device(); return 1;
        default:
            INFO("HOST: Bad fidelity VMSCALL.\n")
            __asm __volatile("hlt");
            return 1;  // retry Fidelity (and halt)
    }
}

/* Copy guest state info from fidelity, but not MSRs, 
copy injection fields. */
void handle_vm_call_done() {
    // Copy safe control area fields.
    guest_vmcb->tsc_offset = guest_vmcb_copy->tsc_offset;
    guest_vmcb->eventinj = guest_vmcb_copy->eventinj;

    // Copy parts of the guest save area back to real VMCB
    // Skipping the MSR registers, as we do not let the guest(s) change them.
    #define copy_vmcb(offset, size) \
        memmove((void*)guest_vmcb + VMCB_SAVE_OFFSET + offset, \
                (void*)guest_vmcb_copy + VMCB_SAVE_OFFSET + offset, size);
    copy_vmcb(VMCB_SAVE_ES_OFFSET, VMCB_SAVE_ES2EFR_SIZE);
    copy_vmcb(VMCB_SAVE_CR4_OFFSET, VMCB_SAVE_CR42STAR_SIZE);
copy_vmcb(VMCB_SAVE_CR2_OFFSET, VMCB_SAVE_CR2_SIZE);

    // Copy guest register save struct
    memmove(guest_regs, guest_greg_copy, sizeof(struct regs));
    }

    /* Read or write a COM1 I/O port based on struct in Fidelity RAM.
     * NOTE: only supports byte IO not word or dword as it should.
     */
    void handle_vm_call_device() {
        /* If outside COM1 port range, return, doing nothing. */
        if (io->port < IO_PORT_SERIAL_START ||
            io->port >= IO_PORT_SERIAL_START + IO_PORT_SERIAL_NUM) {
            return;
        }

        // Out to COM1
        if (io->type == 0) {
            asm("outb %0, %w1" : : "a" (io->a1), "d" (io->port) : );

            // In from COM1
        } else {
            asm("inb %w1, %0" : =a" (io->a1) : "d" (io->port) : );
        }
    }

    /* =========== Guest intercept handling =========== */

    /* Handle all Guest intercepts by copying resource state into Fidelity. */
    void handle_guest_intercept() {
        memmove(guest_vmcb_copy, guest_vmcb, PAGE_SIZE_4K);
        memmove(guest_greg_copy, guest_regs, sizeof(struct regs));
    }
Appendix B

VMRUN_V2

This appendix presents a modified version of the AMD-V VMRUN instruction in the format used by the AMD manual. VMRUN_V2 simplifies guest execution to a random change of a subset of machine state. A machine model using it may be suitable for an initial verification of MinVisor.

VMRUN_V2 Reflect effect of guest execution

Reflects the effect of entry to guest mode, arbitrary execution in guest mode, and exit from guest mode. The physical address of the virtual machine control block (VMCB) describing the guest is taken from the rAX register (the portion of rAX used to form the address is determined by the effective address size). The physical address of the VMCB must be aligned on a 4K-byte boundary.

VMRUN_V2 halts the processor if its preconditions are not met.

VMRUN_V2 saves a subset of host processor state to the host state-save area specified by the physical address in the VM_HSAVE_PA MSR. VMRUN_V2 then randomly changes the RAM accessible by the nested page table pointed to by the VMCB, registers accessible by the CPU based on the VMCB control area, and the VMCB guest save and exit info areas. The processor then reloads the host state, and continues execution of host code at the instruction following the VMRUN_V2 instruction.

This is a Secure Virtual Machine instruction. This instruction generates a #UD exception if SVM is not enabled. See Enabling SVM on page 425 in AMD64 Architecture Programmers Manual Volume 2: System Instructions, order# 24593.

The VMRUN_V2 instruction is not supported in System Management Mode. Processor behavior resulting from an attempt to execute this instruction from within the SMM handler is undefined. The Instruction does not reflect running SMM while in guest mode.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMRUN_V2</td>
<td>rAX 0F</td>
<td>Reflect a random intercepted guest execution</td>
</tr>
</tbody>
</table>
Preconditions

Instruction is called from protected mode, this and MSR write intercept prevent access to long mode. Host is in PAE paging mode. Machine supports nested paging. VMCB is set so that:

- Nested paging is enabled and the nested page table maps guest addresses onto memory address ranges X1, X2, ..., Xn.
- MSR intercept is enabled and the MSR permission bit map marks all MSR write accesses for intercept.
- I/O intercept is enabled and the I/O permission bit map marks all I/O accesses for intercept.

Action

IF (above preconditions fail)
    halt

IF ((MSR_EFER.SVME = 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]
    // This instruction can only be executed in protected mode with SVM enabled

IF (CPL != 0) // This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported system-physical address)
    EXCEPTION [#GP]

if (intercepted(VMRUN))
    #VMEXIT (VMRUN)

remember VMCB address (delivered in rAX)
save host state to physical memory indicated in the VM_HSAVE_PA MSR:
    ES.sel
    CS.sel
    SS.sel
    DS.sel
    GDTR.{base,limit}
    IDTR.{base,limit}
    EFER
    CR0
    CR4
    CR3
    // host CR2 is not saved
    RFLAGS
    RIP
    RSP
RAX

from the VMCB at physical address rAX, load control information: intercept vector
TSC_OFFSET
interrupt control (v_irq, v_intr_*, v_tpr)
EVENTINJ field
ASID

if (nested paging supported) /* which is a precondition*/
NP_ENABLE
if (NP_ENABLE = 1) /* which is a precondition */
nCR3

/* Do not load the guest state */

if (LBR virtualization supported)
LBR_VIRTUALIZATION_ENABLE
if (LBR_VIRTUALIZATION_ENABLE=1)
save LBR state to the host save area
DBGCTL
BR_FROM
BR_TO
LASTEXCP_FROM
LASTEXCP_TO
load LBR state from the VMCB
DBGCTL
BR_FROM
BR_TO
LASTEXCP_FROM
LASTEXCP_TO

/* Do not do guest state consistency check */

Execute command stored in TLB_CONTROL.

GIF = 1 // allow interrupts in the guest

/* Random guest start =======
Allow any guest change to machine state, expect those prevented by VMCB. */
Change any state of machine affected in
real, virtual 8086, or protected mode.
And/or cause any exception/intercept.
Except:
   Memory addresses map to physical address regions X1 .. Xn
   MSR may not be changed (cause VMEXIT and end of this phase)
   IO ports are inaccessible. (cause VMEXIT and end of this phase)
Randomly advance the clock.
/* Random guest END ======= */
GIF = 0

/*Randomly change the VMCB guest save area and EXITINFO fields */

clear EVENTINJ field in VMCB

prepare for host mode by clearing internal processor state bits: clear intercepts
   clear v_irq
   clear v_intr_masking
   clear tsc_offset
   disable nested paging
   clear ASID to zero

reload host state /* From HOST SAVE area */
   GDTR.{base,limit}
   IDTR.{base,limit}
   EFER
   CR0
   CR0.PE = 1 // saved copy of CR0.PE is ignored
   CR4
   CR3
   if (host is in PAE paging mode) /*which is a precondition */
      reloaded host PDPEs
   // Do not reload host CR2 or PAT
   RFLAGS
   RIP
   RSP
   RAX
   DR7 = all disabled
   CPL = 0
   ES.sel; reload segment descriptor from GDT
   CS.sel; reload segment descriptor from GDT
   SS.sel; reload segment descriptor from GDT
   DS.sel; reload segment descriptor from GDT

if (LBR virtualization supported)
   LBR_VIRTUALIZATION_ENABLE
   if (LBR_VIRTUALIZATION_ENABLE=1)
      load LBR state from the host save area:
         DBGCTL
         BR_FROM
         BR_TO
         LASTEXCP_FROM
         LASTEXCP_TO

if (illegal host state loaded, or exception while loading host state)
   shutdown
else
    execute first host instruction following the VMRUN

**Related Instructions**

VMRUN, VMLOAD, VMSAVE.

**rFLAGS Affected**

None.

**Exceptions**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virt 8086</th>
<th>Protected</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID function 8000_0001h.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Secure Virtual Machine was not enabled (EFER.SVME=0).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>The instruction is only recognized in protected mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>CPL was not zero.</td>
</tr>
<tr>
<td>General protection, #GP</td>
<td></td>
<td></td>
<td>X</td>
<td>rAX referenced a physical address above the maximum supported physical address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>The address in rAX was not aligned on a 4Kbyte boundary.</td>
</tr>
</tbody>
</table>
Bibliography


