Building a Hypervisor on a Formally Verifiable Protection Layer

Michael McCoyd Robert Bellarmine Krug, Deepak Goel, Mike Dahlin, William Young
Dept of Computer Science
University of Texas at Austin
Austin, USA
{mccoyd, rkrug, deepak, dahlin, byoung}@cs.utexas.edu

Abstract—Virtualization promises significant benefits in security, efficiency, dependability, and cost. Achieving these benefits depends upon the reliability of the underlying hypervisor. Hypervisors provide complete control of the virtualized resources (protection), a reasonably accurate view of these resources (fidelity), and performance. To facilitate formal verification of protection, we present an architecture, aligned with the hardware virtualization barrier, that separates hypervisor protection from the other goals. The hypervisor is constructed on a minimal trusted computing base or “minvisor” whose main responsibility is protection. Each real guest is paired with an untrusted fidelity guest that builds on the protection layer to provide a fully virtualized environment. This allows verification of protection without considering much of the functionality of a traditional hypervisor. We have coded such a protection layer, developed a simple hypervisor on it, and begun formally verifying its protection properties at the machine code level. The current paper is a progress report.

Keywords—hypervisor; hardware virtualization; formal verification;

I. INTRODUCTION

Many modern computer resources are built on top of hypervisors for security, efficiency, dependability, and cost. We argue for securing hypervisor based resources by building the hypervisor on a mechanically verified protection layer. We present a small protection layer, MinVisor, on which we have built a simple hypervisor, and the partial verification of this protection layer with respect to an ACL2 [1], [2] model of the AMD64 instruction set architecture (ISA).

Many computing service providers, such as Amazon and Rackspace, use commodity hypervisors such as Xen [3] to provide their clients the convenience of their own virtual machine and operating system, as part of the security for their web services [4], [5]. Though smaller than a full-blown operating system, such as the 10 million line Linux kernel [6], Xen, at 82,000 lines, is still a very large system with a history of security vulnerabilities (e.g.,[7]). Our hope is to provide virtualization with a much more constrained and highly assured software platform.

Popek and Goldberg [8] define virtual machine monitors (hypervisors) as having the essential characteristics of protection, fidelity, and speed. Our initial focus is on providing verified protection, with the hope of adding fidelity and speed in the future. To make verification manageable, we reduce the trusted computing base (TCB) to just protection and some minimal devices, and make maximum use of hardware virtualization by aligning our TCB with the hardware guest/host barrier. This splits the hypervisor into the small protection layer, MinVisor, and an untrusted fidelity guest, the Fidelity, paired with each real guest, the Guest, see Figure 1. Each Fidelity uses the paravirtualized devices of the protection layer to provide a fully virtualized machine to its Guest. Guests are isolated from each other, except that a Fidelity can see the RAM of its Guest. Protection, here, is defined as the ability of MinVisor to restore a resource (cpu, memory, network card, etc) to a predefined state. See also Sec. VI-B.

This approach seems viable for a simple networked virtual machine, though our proof of concept provides only one device and one pair of guests, and there is a performance cost to our multiple guest/host transitions. We do not envision using this approach where a wide range of devices is required, but rather for high assurance where a more limited device set will suffice. Our hope is that our approach is scalable to realistic systems, but that remains to be seen. An informal proof skeleton [9] suggests that MinVisor protects all programmer-visible machine resources, based on its C andasm source code and an enumeration of the AMD64 resources.

Our threat model is that we are defending the machine, including attached devices, and other Guests from a malicious or compromised guest. We assume the MinVisor protection layer gets delivered to RAM and starts executing intact on an uncompromised machine.

Formal mechanical verification provides a systematic approach to ensuring that MinVisor protects the programmer-visible resources. Using an ACL2 model of the AMD64 host ISA, we have verified that the machine code of the protection layer, in combination with the AMD-V hardware, protects a subset of resources during guest execution [10]. Protection of this subset is necessary to ensure that MinVisor’s code and data are not changed during guest execution. This is essential for MinVisor to run as intended on intercept of guest execution, and a first step toward verifying that MinVisor fully protects machine resources.

The remaining sections of this paper present: (2) related work; (3) some background on AMD-V hardware virtu-
alization, mechanical verification, and modeling the x86; (4) our approach to hypervisor construction; (5) details of our protection layer MinVisor; (6) details of our machine code verification effort in ACL2 with a model of the x86 ISA; and finally (7) conclusions.

II. RELATED WORK

Hypervisors were first introduced in the 1960s as a way to multiplex scarce and expensive computing resources. The advent of inexpensive hardware and multitasking operating systems eroded their value in the 1980s and 1990s. Over the last few years, however, hypervisors have regained popularity as a versatile technology for enhancing security and reliability. One effect is that processor manufacturers are removing obstacles that made it difficult to virtualize some system resources on earlier processor designs without significant emulation performance penalties. Both Intel and AMD have developed hardware virtualization extensions to the x86 architecture. [11], [12]

The Denali isolation kernel [13] was an effort to host a large number of untrusted internet services at a time when hardware had poor virtualization support. Prioritizing speed and scalability, it provided only simplified devices and relied on a library linked into the guest process to turn these into the expected legacy devices. We mirror its desire to host untrusted code, but prioritize verification of resource protection, use the hardware virtualization barrier for much of our protection, rely on an untrusted guest to provide the hardware that a legacy operating system expects, and accept a cost in speed.

Xenon [14], [15] is a high assurance hypervisor based on re-engineering Xen. The designers have specified a formal security policy based on the notion of independence [16]. Though related to our work, the Xenon hypervisor is much larger (around 70,000 lines of code) and the effort is focused on gaining assurance through a policy-to-code development methodology.

Several research efforts have demonstrated that it is possible to construct small, robust and useful hypervisors. The SecVisor project [17], an ancestor of our work, implemented two hypervisors (1739 and 1112 lines of code, respectively) supporting Linux kernel version 2.6.20. These systems provide strong integrity guarantees. However, a subsequent formal analysis [18] using the Metro model checker found two significant and exploitable design flaws. SecVisor hosted a modified Linux, while MinVisor aims to support hosting an unmodified legacy guest, though currently a serial port is the only device.

Microsoft has developed and used VCC [19], a verifier for Concurrent C, to verify components of a simple hypervisor. This is part of the Verisoft XT effort to verify a small research hypervisor [20], [21], part of a larger software system called Hyper-V. This project performs the verification at the level of the C source code and relies on the fidelity of the mapping from C to the machine code level, while ours operates on final machine code.

The Fiasco project developed a microkernel running on x86 PCs and intended to be compatible with the L4 microkernel. The related VFiasco effort attempted to prove security properties from the Fiasco source code including a small hypervisor developed in C++. A “semantics compiler” translates the C++ code into logical formulas that are analyzed using PVS. They also use a formal but incomplete x86 model. [22]

One of the most ambitious projects aimed at verifying system code is the seL4 project [23], targeting components of the L4 microkernel, particularly the virtual memory manager. With others they later created and verified seL4 [24], a third-generation L4 family microkernel. Their proof is done using the Isabelle/HOL theorem prover and spans three layers across which a refinement proof is performed—an abstract specification, a functional prototype in a subset of Haskell, and an implementation in a subset of C. Our proof is done on the machine code and does not rely on such cross model links.

III. BACKGROUND

A. Hardware virtualization support

Machines with hardware assisted virtualization divide execution into guest or host mode, entered by a host instruction and hardware intercepts. The switch between guest and host is called world shift. Various conditions can be set to trigger the world shift to host mode. On the AMD platform, a virtual machine control block (VMCB) holds information about a guest’s virtualization. The VMCB can restrict use of certain instructions or control registers and access to memory, I/O ports, and model specific registers (MSR). The VMCB contains saved guest register state and information about the last intercept. If a guest attempts actions that have been restricted, an intercept is triggered, and the host can respond to the guest’s attempted action. On each world shift,
AMD-V hardware uses a host save area and the VMCB to save and restore machine registers associated with the host and guest.

AMD-V protects memory with a nested page table that is used after any guest address translation and before access to a physical memory address. A nested page table is created for each guest. A guest’s nested page table could be set to map guest addresses onto fixed regions of the machine address space dedicated to that guest.

AMD-V protects the I/O address space with a bitmap that is part of the Boyer-Moore family of provers. It is both a mechanical verification and software. Example industrial users include Motorola to certify several microcode programs for the Motorola CAP digital signal processor; AMD, IBM, and Centaur Technology to verify microcode for floating point operations (bugs that had escaped all previous tests were found by all three); and Rockwell Collins to prove information flow properties about its AAMP7 processor [25].

As a simple example of formalizing a (hardware) system, let us consider a ripple-carry adder. We will encode natural numbers as bit-vectors, least significant bit first. For example, 5 could be encoded as (T F T F)

Finally (using a function which converts bit-vectors to the corresponding natural numbers, bv→to→nat , not presented here) we can state our theorem (note that we must seed the initial carry value appropriately):

Theorem adder-is-correct

\[ \text{bv→to→nat ( adder ( a, b, F ) ) = } \text{bv→to→nat ( a ) + bv→to→nat ( b )} \]

Note that this correctness theorem is easily provable in ACL2.

C. Model of the x86

For this project, rather than modeling hardware as in the example above, we have developed a series of ever more detailed and accurate models of the x86 based machines at the programmer’s or ISA (Instruction Set Architecture) level. This model consists of two parts—a specification of the physical machine (memory and the programmer-visible registers, including the instruction pointer), and a x86 function that describes the state transitions of this physical machine. The x86 function operates by reading the binary code as pointed to by the instruction pointer, decoding this instruction stream, and updating the registers and memory as specified in the AMD and Intel manuals. Additionally, there is a run-x86 function that steps some number of times.

A schema for a theorem about a piece of software, code, would then look like:

Theorem code-is-correct

\[ \text{code→precondition ( x86 ) & code→to→exit ( n, x86 ) } \implies \text{code→postcondition ( run→x86 ( n, x86 ) )} \]

where code→precondition specifies that the x86’s state is such that code will run correctly. Typical preconditions specify that code is loaded into memory, that the instruction pointer points to the beginning of code, that code’s memory image, call stack, and other memory regions operated on by code are all disjoint, as well as any preconditions specific to code’s correct operation. We state the disjointness properties by, for instance, stating that the call stack consists of the memory region pointed to by the stack pointer plus or minus some specific bounds. Similarly, the code’s memory image lies within the memory region pointed to by the
instruction pointer plus or minus some specific bounds. It then remains as a proof obligation whenever we read some code to show that no stack operations wrote beyond those bounds—otherwise we could have overwritten the code image. Further, assuming that code is called by some higher level procedure, we must show that the return address was not overwritten by code. code—runs—to—exit specifies that code will complete its run in n steps.

code—postcondition typically specifies the parts of the x86’s state that do not change (the frame condition), as well as how the state changes for those parts of concern. Note that without the frame condition, we would not be able to reason about the effects of composing multiple software runs. Note further that we do not need to specify code’s effect on the entire state, only on those portions that we care about.

One caveat should be mentioned here. An x86 program’s verification is only meaningful to the extent the x86 model is accurate. Clearly, one cannot prove the model’s accuracy, given that Intel’s and AMD’s specifications consists of many thousands of pages of often ambiguous English language documents [26], [27]. However, our current model is executable and capable of running over three million instructions per second. We can thus compare the results of extended software runs with those on real hardware, increasing our confidence in our model’s accuracy.

IV. Approach

A. Building on a protection layer

In trying to formally verify a hypervisor, we must limit our system to what can be formally verified. As we are motivated by concern for security, we have focused on complete protection, while accepting limited devices and possible compromise on fidelity and speed. There are already full featured and fast hypervisors that may be secure. Our goal is to examine how to provide a hypervisor with formally verified protection, a few devices, and reasonable speed. Further motivating the choice of protection as our base layer is that protection is an absolute, whereas fidelity, speed, and available devices take on a range of acceptable values. We can not pick the attacks that will be used against us; we can pick the devices our machine uses.

To simplify our verification, we align our trusted computing base with the guest/host barrier provided by the virtualization hardware. The virtualization hardware, in providing support for an authentic virtual machine, already limits the effects of guest execution. By reusing that effort, we reduce the complexity of our verification, as it can focus on the guest/host transition, and not also the user/system or a barrier we construct alone elsewhere.

Since we are verifying only protection, we chose to isolate that into a separate layer to reduce the interaction with other hypervisor code and to provide a smaller code base for verification. Yet protection itself is fairly simple; providing it with some useful fidelity and devices is the hard part. Thus we have built minimal paravirtualized device access into our protection layer as well as created an architecture we feel can allow an untrusted guest to provide fidelity and speed. In addition to making verification manageable, our choice of a limited device set drastically reduces the attack surface relative to hypervisors providing a broad set of devices. Our design also provides an untrusted area, the Fidelity, for part of the device interaction, reserving the trusted core for very simplified, and presumably safer, access. While our goal is security properties based on verification, from a design standpoint this architecture should mitigate the risk of a device exploit such as affected Xen (e.g., [7]).

Our intent is to advance MinVisor’s capability and verification until we have network, timer and interrupts to run a Linux or Windows server with local or network storage. The first stage of this is to verify MinVisor’s current protections using our ACL2 methodology. Then we will add a paravirtualized timer and network card to MinVisor, and have our provided Fidelity create a fully virtualized timer and a network card for the Guest. This should allow the Guest to run Linux with a networked file system. Then the protection of MinVisor will be reverified with those devices.

Our goal is to mechanically verify that MinVisor provides protection. Fidelity and speed will only be demonstrated to the level of “We ran small experiments and the output was what you would expect from an x86.” We defer verification of fidelity, and speed, to a later project phase. There are limits to the fidelity possible with this approach. For example, a trusted platform module provided to the Guest via the Fidelity would expose its details to the Fidelity.

MinVisor and our Fidelity run a simple Guest that demonstrates external, serial port, communication and computation. This is an initial verification of the feasibility of the architecture of building a hypervisor from a protection layer and an untrusted Fidelity guest.

B. Formal verification as a solution

The standard approach to attaining assurance in the correctness of software is to test it. Testing is an inductive approach to predicting software behavior. Programs are tested on a selection of potential inputs; software modules that behave adequately for these inputs are deemed suitably reliable. Modules that misbehave on test cases are “patched” until adequate behavior is achieved. This, however, can only ensure that software does what we want for those inputs tested.

Formal verification, however, can ensure that software is truly correct by analyzing mathematical models of digital systems. Using rigorous, mathematically-based techniques that model programs and computing systems as mathematical entities, it is possible to establish that program models meet their specifications for all potential inputs.

Some tour de force software verification projects (e.g., the CLI stack [28], seL4 kernel [24]) have shown that software
verification is possible and effective at gaining very high assurance. However, current formal methods have proven too labor intensive for most realistic systems, which are either too complex or too dynamic for current verification technology. A potential solution is to limit mathematical analysis to the most critical components of a software system or to architect the system such that its safety / security properties hinge upon only a small, stable and tractable core. This, for example, is the underlying motivation of the seL4 kernel verification project, which nevertheless took many person-years of effort to complete.

The current project adapts a similar motivation. We envision a hypervisor consisting of a minvisor—a small protection and device layer — and a fidelity guest—an arbitrarily complex untrusted guest that allows the accurate emulation of the provided virtual machine. Guarantees of safety and security are provided by the minvisor, not by the hypervisor as a whole. If the protection layer is reliable, we gain the security guarantees regardless of the reliability of the rest of the hypervisor. Because the protection layer itself is relatively small—a few thousand lines of code, compared to millions for the guest OS—it is amenable to rigorous formal verification.

C. Machine code verification

With others, we are creating an ACL2 [29] machine model of the AMD64 to allow us to reason about machine code. To provide a high level of assurance, we are targeting a machine code level verification of our protection layer. This allows us to reason about the software at a single level, that at which it actually runs, and to avoid having to specify the effects of running a mixture of machine code and high-level code on the underlying machine model. Verifying at the machine code level also avoids issues of needing to trust the tools used to compile the hypervisor [30].

V. MINVISOR

MinVisor is a proof of concept of our approach of building a hypervisor on top of a verifiable protection layer. To simplify verification, MinVisor places its protection at the hardware guest/host barrier. A more detailed account of its design is available in [9], including an informal proof skeleton based on the source code. The proof skeleton suggests that MinVisor provides the protection characteristics of a hypervisor. Further, we have built a hypervisor for a simple x86-based machine that runs on top of MinVisor.

MinVisor is a protection layer, running on bare hardware, that helps an untrusted guest, the Fidelity, create a hypervisor for a single guest, the Guest, see figure 1. MinVisor, a 32-bit binary, is loaded by the BIOS and runs on bare AMD-V [31] hardware. Currently, it runs one pair of untrusted 32-bit guests, the Fidelity and the Guest. For simplicity we have restricted MinVisor to 32-bits. As MinVisor must protect or clear machine resources from the effects of the guests, we must restrict the guests to 32-bits as well. MinVisor protects itself and the machine resources from all guests, and provides paravirtualized device access to the Fidelity. MinVisor does not provide the normal hypervisor isolation between guests. Instead, it allows the Fidelity to see the RAM, intercept, and register information of its paired Guest, so that the Fidelity can create a hypervisor for that Guest. It is intended that MinVisor’s protection will provide isolation between pairs of guests. The protection and isolation between pairs of guests depends 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. This code size is well within the range feasible for formal mechanical verification. MinVisor provides a paravirtualized serial port; our Fidelity provides a transparently virtualized serial port to the Guest.

Guest access to devices is mediated by the Fidelity, as shown in Figure 2. When the Guest tries to access the serial port, the hardware intercepts the IN or OUT instruction to the I/O address space and passes control to MinVisor. In all cases of an intercept from the Guest, MinVisor passes the information about the intercept to the Fidelity without examining the information. The Fidelity does whatever processing it wants to create the result of the device access, including any hypercalls to access the actual hardware device. If MinVisor receives a device access hypercall from the Fidelity, MinVisor verifies the access is safe and sends data from the Fidelity, or puts received data in the Fidelity. Once the Fidelity has finished constructing a response, it signals MinVisor to use that information to resume the Guest. MinVisor adds the fixed security restrictions to new state that has been created for the Guest and resumes Guest execution, without examining that new state.

MinVisor’s limited interaction with the guests makes it very difficult to compromise the host. Once the guests have been started, MinVisors code has only four decision points: whether the Fidelity is finished with its intercept processing, whether it is making a device hypercall, the direction of a device hypercall, and if that device hypercall is safe. It is expected that this type of processing can create simple network access, which a Fidelity can use to create a transparently virtualized network card for the Guest, at some cost to performance due to the extra guest/host transitions.

A. Design

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 to MinVisor. MinVisor uses the hardware host mode for protection and alternates between running the pair of guests as needed.

B. Construction

The BIOS loads MinVisor over the networked pre-execution environment (PXE). PXE was chosen for ease of development; a trusted platform start point (TXT or SKINIT) would be preferable as they allow a late launch of the TCB and would allow it to be smaller. Here, we assume that the machine is not compromised before loading MinVisor. MinVisor uses PXE to download two 32KiB images, for the Fidelity and the Guest, staging them after it in low memory.

MinVisor makes use of the AMD-V hardware virtualization to protect the memory, I/O ports, and model specific registers, and to restore registers to host values on intercept. To help the Fidelity create a hypervisor, MinVisor copies information from the VMCB about the current state and intercept of the Guest into the memory of the Fidelity. In this way the Fidelity has all the information and paravirtualized devices it needs to create a response to the intercept of the Guest. MinVisor blindly routes information between the Guest and the Fidelity, thus allowing the Fidelity to provide transparently virtualized devices while MinVisor need only provide paravirtualized devices and a communication channel.

MinVisor sets up the VMCB so that the hardware protects several components of the machine state during guest execution. These crucially include MinVisor’s memory, the I/O address space, and the model specific registers (MSR). Without protecting MinVisor’s memory during guest execution, MinVisor could not easily resume its intended execution on intercept. MinVisor reserves separate RAM areas for MinVisor and each guest, Guests are prevented from directly accessing the memory addresses of other guests, MinVisor, the machine BIOS, or device buses. Protecting the I/O address space allows MinVisor to control access to devices. Protecting the MSRs prevents the guest from entering 64-bit mode, a simplification in our work, and from changing various machine characteristics the MSR space can control. Of the machine resources that can change during guest execution, the AMD-V hardware restores most registers to the host state on intercept. The few registers not restored by the hardware on intercept are restored to saved values by MinVisor. After intercept and MinVisor register restore, this limits the effects of guest execution to guest memory and a few areas of the VMCB. If desired, these can be easily cleared by the host, and the minimal paravirtualized devices returned to a default state returning the machine to a completely host defined state.

C. Results

Experiments on the bare silicon of an AMD Opteron 1300 show that MinVisor and a Fidelity can provide an x86 based virtual machine with a transparently virtualized serial port. The virtual 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 reads. The virtual machine ignored model specific register (MSR) write attempts. Thus the Fidelity design satisfied basic execution needs but was far from complete. A protected mode Guest polled the serial port and echoed any input. The external serial port input was echoed and logs show the expected execution. This suggests that with expanded devices MinVisor could serve as a basis for constructing a useful x86 or AMD64 hypervisor. Speed for full virtualization is an issue, a Guest echo program will miss
characters if flooding at 14.4K bytes/s with no buffering in the serial port on a 2.1 GHz AMD Opteron 1300.

VI. VERIFICATION

A. Early work

We have completed verification of several pieces of code using an earlier and simpler machine model we call the Y86++. This processor model is based upon Bryant, et. al., Y86 [32], but enhanced to include paging and other features necessary for verifying MinVisor. The primary simplifications relative to the x86 were: 1) a much simpler assembler/binary language, 2) the exclusive use of 16-bit operands, and 3) no exception handling—any exception caused the machine to enter a halted state.

Using this model we verified an earlier but more complex version of MinVisor’s code for setting up the nested page tables. We showed that after this code is run, the page table translation is as desired — “good” memory addresses (those not being protected by MinVisor) yield the identity map, while “bad” memory addresses (those being protected by MinVisor) yield a page fault.

Note that repeating this earlier work with the current, simpler, code and the current, more complex and realistic, model of the x86, should be a rather simple affair. First, the added complexity of the x86 assembly/binary will add no complexity to the proof, since the binary code is assumed to be loaded into memory, and its only purpose is to direct the step−x86 function as to which operation (e.g., ADD or SHIFTL) should be simulated next. Second, since the current code for setting up the page tables uses 32-bit operands, the (binary) code is actually simpler than with 16-bit operands. Finally, since we assume as a precondition, that the state is such that no intercepts (nested page faults) will be thrown, their presence or absence in the machine model does not affect the proof complexity (although it will affect the complexity of the precondition).

B. Protection

Our current model of an x86 machine’s state includes the memory and the programmer visible registers plus a few “extra” registers such as a pointer to the VMCB and the host’s save area necessary to model MinVisor. Our model also includes some one-bit flags to indicate whether the machine is currently running as a host or guest. The step−x86 function includes all the opcodes necessary to model MinVisor’s code for setting up its protections. This partial set of opcodes is also used to model guest execution.

In proving that MinVisor protects programmer-visible resources, we first wish to prove that the protections MinVisor sets up are sufficient to protect a subset of the resources during execution of a potentially malicious guest. This protection is necessary for MinVisor to run as intended after intercept. Given the resources present in our current model, this subset is the model specific registers (MSR) and MinVisor’s memory. What we mean by protecting the resources is that those resources are not changed by any possible execution of a guest (including the Fidelity). Note that this subset includes 1) the memory occupied by the nested page tables, MinVisor’s code image, the VMCB, and the host save area; as well as 2) model specific registers such as the EFER which prevents the guest from entering 64-bit mode and changing 64-bit resources.

The set of programmer-visible resources that we must prove are protected during guest execution will have to be enhanced as we add resources, such as the BIOS and devices, to our model. Adding the I/O address space to our model will be a crucial addition to modeling protections during guest execution. But we believe that the basic theorems and proof outline described below will still hold.

C. Theorems

There are three theorems we need to prove:

Theorem minvisor-setup-is-correct

minvisor−setup−precondition ( x86 ) &
minvisor−setup−runs−to−exit ( n, x86 )
⇒
protections−in−place ( run−x86 ( n, x86 ) )

Theorem guest-run-does-nothing-bad

guest−run−precondition ( x86 ) &
protections−in−place ( x86 ) &
run−stays−in−guest ( n, x86 )
⇒
msr−r−protected ( x86, run−x86 ( n, x86 ) )

Theorem minvisor-run-does-nothing-bad

minvisor−run−precondition ( x86 ) &
protections−in−place ( x86 )
⇒
protections−in−place ( run−x86 ( n, x86 ) )

Simply put, these three theorems state that 1) MinVisor’s setup code sets up the protections correctly, 2) the MSRs and MinVisor RAM are protected from all runs of a guest, and 3) no run of MinVisor disturbs these protections. We examine these theorems and their proofs more carefully below.

D. Proof outline

The first of these theorems, minvisor-setup-is-correct is probably the simplest. The precondition minvisor−setup−precondition states that: both the setup (protection) and runtime code are loaded into memory;
that the instruction pointer points to the start of this code; that pointers to the memory regions to be occupied by the nested page-tables, the VMCB, and the host save area point to sufficiently sized memory regions; and these memory regions, the call stack, and the code are all disjoint. The precondition also states that the machine is in a “rational” state, such as that it is running as a host in 32-bit protected mode. The postcondition protections—in—place states that MinVisor’s run-time code is still in place and that the VMCB and nested page-tables are set up appropriately. To prove this theorem is just a straight-forward analysis of (the compilation of) less than 200 lines of C code.

The second theorem, guest-run-does-nothing-bad, is more interesting. The precondition guest—run—precondition is very weak. It only states that the machine is running in guest mode. The third hypothesis, run—stays—in—guest ( n, x86 ), states that stepping the state x86 n times or fewer leaves the state in guest mode. That is, there is no intercept that causes a world switch to host mode during a run of n steps. Thus, the conclusion msrmram—protected( x86, run—x86 ( n, x86 ) ) relates 1) the initial state of a guest x86 to 2) the final state run—x86 ( n, x86 ) of any execution that does not include an intercept caught by MinVisor. This relation, msrmram—protected, states that the MSRs and those memory regions that are being protected by MinVisor remain unchanged. In particular, we have

**Theorem useful-lemma**

\[
\text{protections—in—place ( x86 ) } \land \\
\text{msrmram—protected( x86, run—x86 ( n, x86 ) )} \\
\implies \\
\text{protections—in—place ( run—x86 ( n, x86 ) )}
\]

This proof proceeds by induction on n. The base case, for n = 0, is trivial, since run—x86 ( n, x86 ) returns the entire state unchanged. We now need to show that if guest-run-does-nothing-bad holds for some n it also holds for n+1. There are two subcases. In the first, run—stays—in—guest ( n+1, x86 ) no longer holds and the theorem is trivially true. In the second, we need to prove that after taking one more step, those memory regions and registers that are being protected by MinVisor are still unchanged.

This second case also holds because of the way the step—x86 function is written. To see this, recall that the step—x86 function describes the state transitions of the x86, updating the registers and memory as specified in the AMD and Intel manuals. Now, the step function is mostly a wrapper around a large conditional branch statement, each branch of which is responsible for calling a particular execute—xxx, such as execute—ADD or execute—SHIFTL. Each of these execute—xxx functions then checks whether the operation it is about to perform will raise an exception (intercept). In particular, it checks to see if the machine is in guest mode and, if so, if the operation is permitted by the VMCB. If the operation is not permitted, it raises an exception. Similarly, for memory reads or writes, the operation checks if an intercept would occur by walking the nested page tables. These intercepts are then caught by the step—x86 function and a world shift to host mode occurs. Note here that we are assuming that run—stays—in—guest ( n+1, x86 ) and so such “bad” operations can be ignored by this theorem.

Thus, while the first theorem was largely about code this second theorem is instead largely about the step—x86 function and a couple of static data structures; and it is by the coding of the step—x86 function—given the assumption the VMCB and nested page-tables were set up correctly—that guest-run-does-nothing-bad holds. Our proof reflects the fact that MinVisor’s correct operation is based on AMD-V’s hardware virtualization support. Note also that because this theorem is not specific to any particular code, it applies to all guests including the fidelity guest.

The third theorem, minvisor-run-does-nothing-bad, combines aspects of both the previous ones. The precondition minvisor—run—precondition states that the machine is running as a host in 32-bit protected mode, that MinVisor’s run-time code is loaded into memory, and that the instruction pointer is pointing somewhere inside that code. Note that n is “free” in the conclusion; that is, n is not mentioned in any of the hypotheses and so is free to take on any arbitrary value. The postcondition protections—in—place then says that after running any arbitrary number of steps, the protections are still in place. Note that some of these n steps may be due to the execution of a guest.

The proof of minvisor-run-does-nothing-bad proceeds as follows. First, no part of MinVisor’s run-time changes the protections. This follows from a simple code analysis of MinVisor’s run-time code. Second, by guest-run-does-nothing-bad and useful-lemma we see that no guest execution changes the protections. Finally, the world changes induced by the host and the interception of a guest do not change the protections. Pasting this all together, we get the desired conclusion.

This theorem could be strengthened if we weakened msrmram—protected slightly to exclude those parts of the VMCB used for saving guest state data upon world shift from guest to host, and those parts of the memory actively used by MinVisor’s run-time.

**VII. Conclusion**

The construction of MinVisor, our earlier verification of its nested page tables, and the creation of an ACL2 model of AMD-V nested paging suggest that mechanical machine code verification of hypervisor protection is feasible. We have shown that an untrusted guest running on a very small protection layer can create a hypervisor for a simple x86 based machine with one device. An informal source code
proof skeleton suggests that this protection layer does protect itself and programmer-visible resources. Our approach to the mechanical machine code verification of correct nested page table creation and its protection of MinVisor’s memory has been presented.

We believe a staged development of a protection layer, sample Fidelity guest, and verification of the protection layer is a promising approach to achieving a hypervisor with verified protection. Once this is achieved, verification can be extended to ensuring that the virtual machine provided by a specific Fidelity provides a fairly accurate model of the x86 or AMD64 ISA. Crucially, this need not require change or reverification of the protection layer.

ACKNOWLEDGMENTS

We thank our colleagues at the University of Texas, particularly Sandip Ray, Warren Hunt, and J Moore for their work on the Y86, Matt Kaufmann for help with ACL2, and Ryan Johnson for help testing. We’d particularly like to thank Sandip Ray for helpful discussions throughout the project. The comments from the anonymous reviewers and our shepherd were also very helpful. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0917162, and by Raytheon under contract 200901296.

REFERENCES


