1. Overview

2. Virtualization

3. What We Did
   Proofs
   Model

4. Proof Statistics
1. Overview

2. Virtualization

3. What We Did
   Proofs
   Model

4. Proof Statistics
Motivations

- Verify (a very simple) hypervisor
Motivations

- Verify (a very simple) hypervisor
- Develop tools (and exemplar) for reasoning about X86 code
Motivations

- Verify (a very simple) hypervisor
- Develop tools (and exemplar) for reasoning about X86 code
- Develop tools (and exemplar) for reasoning about software
Motivations

- Verify (a very simple) hypervisor
- Develop tools (and exemplar) for reasoning about X86 code
- Develop tools (and exemplar) for reasoning about software
- Both System and User level
For our purposes, a hypervisor is a software system that allows multiple operating systems (called guests) to run concurrently on a host computer.
MinVisor is a minimal (Type I or bare metal) hypervisor:

- Being developed by the Systems Group at UT
MinVisor is a minimal (Type I or bare metal) hypervisor:

- Being developed by the Systems Group at UT
- Specific goal is to verifiably protect the physical machine from a malicious guest
1. Overview

2. Virtualization

3. What We Did
   Proofs
   Model

4. Proof Statistics
Popek and Goldberg, (1974) specified the conditions under which an ISA can support a hypervisor:

- execution of “normal” instructions must be roughly equivalent in both host and guest mode
Popek and Goldberg, (1974) specified the conditions under which an ISA can support a hypervisor:

- execution of “normal” instructions must be roughly equivalent in both host and guest mode
- There must be a method, such as an address translation system, to protect the host system and any other guest OS from the active guest OS.
Popek and Goldberg, (1974) specified the conditions under which an ISA can support a hypervisor:

- execution of "normal" instructions must be roughly equivalent in both host and guest mode

- There must be a method, such as an address translation system, to protect the host system and any other guest OS from the active guest OS.

- There must be a way to automatically signal the hypervisor when a guest OS attempts to execute a sensitive instruction. It must also be possible for the hypervisor to simulate the effect of the instruction.
Sensitive instructions include:

- Instructions that change the mode of the guest OS
Sensitive instructions include:

- Instructions that change the mode of the guest OS
- Instructions that change sensitive resources such as the clock or interrupt registers.
Sensitive instructions include:

- Instructions that change the mode of the guest OS
- Instructions that change sensitive resources such as the clock or interrupt registers.
- Instructions that change the storage protection system
Virtualization can be achieved using

- software — but this is inefficient and complex
- hardware — efficient, now supported in X86
**Basic Idea**

MinVisor

- setup

- handle intercept

VMRUN

- resume at next inst. after vmrun

Guest OS

- instruction flow inside guest

Hypervisor Verification Virtualization
Outline

1. Overview

2. Virtualization

3. What We Did
   - Proofs
   - Model

4. Proof Statistics
Popek and Goldberg, (1974) specified the conditions under which an ISA can support a hypervisor:

- execution of “normal” instructions must be roughly equivalent in both host and guest mode
- There must be a method such as an address translation system to protect the host and any other guest OS from the active guest OS.
- There must be a way to automatically signal the hypervisor when a guest OS attempts to execute a sensitive instructions. It must also be possible for the hypervisor to simulate the effect of the instruction.
Linear Address Translation to a 2-MByte Page

Inside the PDPTE and PDE, there are various status and control bits — in particular, a Page Present bit. If this bit is zero, a Page Fault will be signalled.
Inside the PDPTE and PDE, there are various status and control bits — in particular, a *Page Present* bit. If this bit is zero, a *Page Fault* will be signalled.
(defthm main-thm
  (implies
   (and (create_nested_pt-pre s0)
        (create_nested_pt-exists-next-exitpoint s0)
        (n32p addr))
    (b* ((pdpt (r32 (+ 4 (g :esp s0)) s0))
         (visor-start (R32 (+ 12 (G :ESP S0)) S0))
         (visor-size (R32 (+ 16 (G :ESP S0)) S0))
         (s1 (create_nested_pt-next-exitpoint s0))
         (s1 (s :cr3 pdpt s1))
         (s1 (set-paging t s1)))
    (equal (va-to-pa addr s1)
      (if (disjointp (list (list addr)
        (range visor-start 0 visor-size
          addr
          :PAGE-FAULT))))))
This proof was carried out on the Y86++ binary, compiled from the original C code using GCC, and assembled to binary using an augmented version of Hunt’s Y86 assembler.

The translation from X86 assembly to Y86++ assembly was done by hand.
Our Y86++ model is based on Hunt’s implementation of the Y86.
Our Y86++ model is based on Hunt’s implementation of the Y86.

The Y86:

- is designed for pedagogical purposes
- its ISA is similar to the X86’s (but a simplified subset)
Our Y86++ model is based on Hunt’s implementation of the Y86.

The Y86:
▶ is designed for pedagogical purposes
▶ its ISA is similar to the X86’s (but a simplified subset)

We augmented the Y86 to include
▶ new components in the machine state
▶ new instructions
▶ a guest/supervisor mode flag
▶ nested page tables

But we did not add any way to propagate or handle Page Faults.
Our Y86++ model is based on Hunt’s implementation of the Y86.

The Y86:
- is designed for pedagogical purposes
- its ISA is similar to the X86’s (but a simplified subset)

We augmented the Y86 to include
- new components in the machine state
- new instructions
- a guest/supervisor mode flag
- nested page tables

But we did not add any way to propagate or handle Page Faults.
Using a Lispy format for the Y86++:

X86                         Y86++
pushl %ebp                  (pushl :ebp)
Using a Lispy format for the Y86++:

X86
pushl %ebp
subl $48, %esp

Y86++
(pushl :ebp)
(irmovl 48 :imme1)
(subl :imme1 :esp)
Using a Lispy format for the Y86++:

X86                        Y86++
pushl %ebp                 (pushl :ebp)

subl $48, %esp             (irmovl 48 :imme1)
                          (subl :imme1 :esp)

addl $1, -16(%ebp)        (irmovl 1 :imme1)
                          (mrmovl -16 (:ebp) :valu1)
                          (addl :imme1 :valu1)
                          (rmmovl :valu1 -16 (:ebp))
Outline

1. Overview

2. Virtualization

3. What We Did
   - Proofs
   - Model

4. Proof Statistics
Some Proof Statistics

- approximately 125 lines of C code, including 30 lines of comments
- approximately 175 lines of X86 assembly
- approximately 225 lines of Y86++ assembly
Some Proof Statistics

- approximately 125 lines of C code, including 30 lines of comments
- approximately 175 lines of X86 assembly
- approximately 225 lines of Y86++ assembly

- approximately 200 defthm events
- approximately 100 defun events