6.828 2011 Lecture 19: Virtual Machines

Read: A comparison of software and hardware techniques for x86 virtualizaton, Keith Adams and Ole Agesen, ASPLOS 2006.

what's a virtual machine?
  simulation of a computer
  running as an application on a host computer
  accurate
  isolated
  fast

why use a VM?
  one computer, multiple operating systems (OSX and Windows)
  manage big machines (allocate CPUs/memory at o/s granularity)
  kernel development environment (like qemu)
  better fault isolation: contain break-ins

how accurate do we need?
  handle weird quirks of operating system kernels
  reproduce bugs exactly
  handle malicious software
    cannot let guest break out of virtual machine!
  usual goal:
    impossible for guest to distinguish VM from real computer
    impossible for guest to escape its VM
  some VMs compromise, require guest kernel modifications

VMs are an old idea
  1960s: IBM used VMs to share big machines
  1990s: VMWare re-popularized VMs, for x86 hardware

terminology
  [diagram: h/w, VMM, VMs..]
  VMM ("host")
  guest: kernel, user programs
  VMM might run in a host O/S, e.g. OSX
    or VMM might be stand-alone

VMM responsibilities
  divide memory among guests
  time-share CPU among guests
  simulate per-guest virtual disk, network
    really e.g. slice of real disk

why not simulation?
  VMM interpret each guest instruction
  maintain virtual machine state for each guest
    eflags, %cr3, &c
  much too slow!

idea: execute guest instructions on real CPU when possible
  works fine for most instructions

e.g. add %eax, %ebx
how to prevent guest from executing privileged instructions?
could then wreck the VMM, other guests, &c

idea: run each guest kernel at CPL=3
ordinary instructions work fine
privileged instructions will (usually) trap to the VMM
VMM can apply the privileged operation to *virtual* state
not to the real hardware
"trap-and-emulate"

Trap-and-emulate example -- CLI / STI
VMM maintains virtual IF for guest
VMM controls hardware IF
Probably leaves interrupts enabled when guest runs
Even if a guest uses CLI to disable them
VMM looks at virtual IF to decide when to interrupt guest
When guest executes CLI or STI:
Protection violation, since guest at CPL=3
Hardware traps to VMM
VMM looks at *virtual* CPL
If 0, changes *virtual* IF
If not 0, emulates a protection trap to guest kernel
VMM must cause guest to see only virtual IF
and completely hide/protect real IF

trap-and-emulate is hard on an x86
not all privileged instructions trap at CPL=3
popf silently ignores changes to interrupt flag
pushf reveals *real* interrupt flag
all those traps can be slow
VMM must see PTE writes, which don't use privileged instructions

what real x86 state do we have to hide (i.e. != virtual state)?
CPL (low bits of CS) since it is 3, guest expecting 0
gdt descriptors (DPL 3, not 0)
gdtr (pointing to shadow gdt)
idt descriptors (traps go to VMM, not guest kernel)
idtr
pagetable (doesn't map to expected physical addresses)
%cr3 (points to shadow pagetable)
IF in EFLAGS
%cr0 &c

how can VMM give guest kernel illusion of dedicated physical memory?
guest wants to start at PA=0, use all "installed" DRAM
VMM must support many guests, they can't all really use PA=0
VMM must protect one guest's memory from other guests
idea:
claim DRAM size is smaller than real DRAM
ensure paging is enabled
maintain a "shadow" copy of guest's page table
shadow maps VAs to different PAs than guest
real %cr3 refers to shadow page table

virtual %cr3 refers to guest's page table
example:
VMM allocates a guest phys mem 0x1000000 to 0x2000000
VMM gets trap if guest changes %cr3 (since guest kernel at CPL=3)
VMM copies guest's pagetable to "shadow" pagetable
VMM adds 0x1000000 to each PA in shadow table
VMM checks that each PA is < 0x2000000

Why can't VMM just modify the guest's page-table in-place?

also shadow the GDT, IDT
real IDT refers to VMM's trap entry points
VMM can forward to guest kernel if needed
VMM may also fake interrupts from virtual disk
real GDT allows execution of guest kernel by CPL=3

note we rely on h/w trapping to VMM if guest writes %cr3, gdtr, &c
do we also need a trap if guest *read*s?

do all instructions that read/write sensitive state cause traps at CPL=3?
push %cs will show CPL=3, not 0
sgdt reveals real GDTR
pushf pushes real IF
suppose guest turned IF off
VMM will leave real IF on, just postpone interrupts to guest
popf ignores IF if CPL=3, no trap
so VMM won't know if guest kernel wants interrupts
IRET: no ring change so won't restore restore SS/ESP

how can we cope with non-trapping instructions that reveal real state?
modify guest code, change them to INT 3, which traps
keep track of original instruction, emulate in VMM
INT 3 is one byte, so doesn't change code size/layout
this is a simplified version of the paper's Binary Translation

how does rewriter know where instruction boundaries are?
or whether bytes are code or data?
can VMM look at symbol table for function entry points?

idea: scan only as executed, since execution reveals instr boundaries
original start of kernel (making up these instructions):

```
entry:
  pushl %ebp
  ...
  popf
  ...
  jnz x
  ...
  jxx y
x:
  ...
  jxx z
```

when VMM first loads guest kernel, rewrite from entry to first jump
replace bad instrs (popf) with int3

replace jump with int3
then start the guest kernel
on int3 trap to VMM
look where the jump could go (now we know the boundaries)
for each branch, xlate until first jump again
replace int3 w/ original branch
re-start
keep track of what we've rewritten, so we don't do it again

indirect calls/jumps?
same, but can't replace int3 with the original jump
since we're not sure address will be the same next time
so must take a trap every time

ret (function return)?
== indirect jump via ptr on stack
can't assume that ret PC on stack is from a call
so must take a trap every time. slow!

what if guest reads or writes its own code?
can't let guest see int3
must re-rewrite any code the guest modifies
can we use page protections to trap and emulate reads/writes?
no: can't set up PTE for X but no R
perhaps make CS != DS
put rewritten code in CS
put original code in DS
write-protect original code pages
on write trap
emulate write
re-rewrite if already rewritten
tricky: must find first instruction boundary in overwritten code

do we need to rewrite guest user-level code?
technically yes: SGDT, IF
but probably not in practice
user code only does INT, which traps to VMM

how to handle pagetable?
remember VMM keeps shadow pagetable w/ different PAs in PTEs
scan the whole pagetable on every %cr3 load?
to create the shadow page table

what if guest writes %cr3 often, during context switches?
idea: lazy population of shadow page table
start w/ empty shadow page table (just VMM mappings)
so guest will generate many page faults after it loads %cr3
VMM page fault handler just copies needed PTE to shadow pagetable
restarts guest, no guest-visible page fault

what if guest frequently switches among a set of page tables?
as it context-switches among running processes
probably doesn't modify them, so re-scan (or lazy faults) wasted
idea: VMM could cache multiple shadow page tables

cache indexed by address of guest pagetable
start with pre-populated page table on guest %cr3 write
would make context switch much faster

what if guest kernel writes a PTE?
store instruction is not privileged, no trap
does VMM need to know about that write?
yes, if VMM is caching multiple page tables
idea: VMM can write-protect guest's PTE pages
trap on PTE write, emulate, also in shadow pagetable

this is the three-way tradeoff the paper talks about
trace costs / hidden page faults / context switch cost
reducing one requires more of the others
and all three are expensive

how to guard guest kernel against writes by guest programs?
both are at CPL=3
delete kernel PTEs on IRET, re-install on INT?

how to handle devices?
trap INB and OUTB
DMA addresses are physical, VMM must translate and check
rarely makes sense for guest to use real device
want to share w/ other guests
each guest gets a part of the disk
each guest looks like a distinct Internet host
each guest gets an X window
VMM might mimic some standard ethernet or disk controller
regardless of actual h/w on host computer
or guest might run special drivers that jump to VMM

Today's paper

Two big issues:
How to cope with instructions that reveal privileged state?
e.g. pushf, looking at low bits of %cs
How to avoid expensive traps?

VMware's answer: binary translation (BT)
Replace offending instructions with code that does the right thing
Code must have access to VMM's virtual state for that guest

Example uses of BT
CLI/STI/pushf/popf -- read/write virtual IF
Detect memory stores that modify PTEs
Write-protect pages, trap the first time, and rewrite
New sequence modifies shadow pagetable as well as real one

How to hide VMM state from guest code?
Since unprivileged BT code now reads/writes VMM state
Put VMM state in very high memory
Use segment limits to prevent guest from using last few pages
But set up %gs to allow BT code to get at those pages

BT challenges
  Hard to find instruction boundaries, instructions vs data
  Translated code is a different size
    Thus code pointers are different
    Program expects to see original fn ptrs, return PCs on stack
    Translated code must map before use
    Thus every RET needs to look up in VMM state

Intel/AMD hardware support for virtual machines
  has made it much easier to implement a VMM w/ reasonable performance
  h/w itself directly maintains per-guest virtual state
    CS (w/ CPL), EFLAGS, idtr, &c
  h/w knows it is in "guest mode"
    instructions directly modify virtual state
    avoids lots of traps to VMM
  h/w basically adds a new priv level
    VMM mode, CPL=0, ..., CPL=3
    guest-mode CPL=0 is not fully privileged
  no traps to VMM on system calls
    h/w handles CPL transition
  what about memory, pagetables?
    h/w supports *two* page tables
    guest page table
    VMM's page table
    guest memory refs go through double lookup
      each phys addr in guest pagetable translated through VMM's pagetable
    thus guest can directly modify its page table w/o VMM having to shadow it
      no need for VMM to write-protect guest pagetables
      no need for VMM to track %cr3 changes
    and VMM can ensure guest uses only its own memory
      only map guest's memory in VMM page table