CS5460: Operating Systems

Lecture: Virtualization

Anton Burtsev
March, 2013
Traditional operating system
Virtual machines
A bit of history

- Virtual machines were popular in 60s-70s
  - Share resources of mainframe computers [Goldberg 1974]
  - Run multiple single-user operating systems
- Interest is lost by 80s-90s
  - Development of multi-user OS
  - Rapid drop in hardware cost
- Hardware support for virtualization is lost
What is the problem?

- Hardware is not designed to be multiplexed
- Loss of isolation
Virtual machine

Efficient duplicate of a real machine

- Compatibility
- Performance
- Isolation

Disk Driver

Block or File System Layer
Trap and emulate
What needs to be emulated?

- CPU and memory
  - Register state
  - Memory state
- Memory management unit
  - Page tables, segments
- Platform
  - Interrupt controller, timer, buses
- BIOS
- Peripheral devices
  - Disk, network interface, serial line
x86 is not virtualizable

- Some instructions (sensitive) read or update the state of virtual machine and don't trap (non-privileged)
  - 17 sensitive, non-privileged instructions [Robin et al 2000]
### x86 is not virtualizable (II)

<table>
<thead>
<tr>
<th>Group</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to interrupt flag</td>
<td>pushf, popf, iret</td>
</tr>
<tr>
<td>Visibility into segment descriptors</td>
<td>lar, verr, verw, lsl</td>
</tr>
<tr>
<td>Segment manipulation instructions</td>
<td>pop &lt;seg&gt;, push &lt;seg&gt;, mov &lt;seg&gt;</td>
</tr>
<tr>
<td>Read-only access to privileged state</td>
<td>sgdt, sldt, sidt, smsw</td>
</tr>
<tr>
<td>Interrupt and gate instructions</td>
<td>fcall, longjump, retfar, str, int &lt;n&gt;</td>
</tr>
</tbody>
</table>

#### Examples

- **`popf`** doesn't update interrupt flag (IF)
  - Impossible to detect when guest disables interrupts

- **`push %cs`** can read code segment selector (%cs) and learn its CPL
  - Guest gets confused
Solution space

- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
    - Intel VT-x, AMD SVM
Basic blocks of a virtual machine monitor: QEMU example
Interpreted execution:
BOCHS, JSLinux
What does it mean to run guest?

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086
- How many cycles per instruction?
Binary translation: VMWare
int isPrime(int a) {
    for (int i = 2; i < a; i++) {
        if (a % i == 0) return 0;
    }
    return 1;
}

isPrime:    mov    %ecx, %edi ; %ecx = %edi (a)
            mov    %esi, $2 ; i = 2
            cmp    %esi, %ecx ; is i >= a?
            jge    prime    ; jump if yes
nexti:      mov    %eax, %ecx ; set %eax = a
            cdq
            idiv   %esi      ; a % i
            test   %edx, %edx ; is remainder zero?
            jz     notPrime  ; jump if yes
            inc    %esi      ; i++
            cmp    %esi, %ecx ; is i >= a?
            jl     nexti     ; jump if no
prime:      mov    %eax, $1  ; return value in %eax
            ret
notPrime:   xor    %eax, %eax ; %eax = 0
            ret
isPrime:  mov %ecx, %edi ; %ecx = %edi (a)
        mov %esi, $2 ; i = 2
        cmp %esi, %ecx ; is i >= a?
        jge prime ; jump if yes
nexti:   mov %eax, %ecx ; set %eax = a
        cdq ; sign-extend
        idiv %esi ; a % i
        test %edx, %edx ; is remainder zero?
        jz notPrime ; jump if yes
        inc %esi ; i++
        cmp %esi, %ecx ; is i >= a?
        jl nexti ; jump if no
prime:   mov %eax, $1 ; return value in %eax
        ret
notPrime: xor %eax, %eax ; %eax = 0
        ret

isPrime’: mov %ecx, %edi ; IDENT
        mov %esi, $2
        cmp %esi, %ecx
        jge [takenAddr] ; JCC
        jmp [fallthrAddr]
isPrime':  *mov  %ecx, %edi    ; IDENT
  mov  %esi, $2
  cmp  %esi, %ecx
  jge  [takenAddr]    ; JCC
       ; fall-thru into next CCF
nexti':  *mov  %eax, %ecx    ; IDENT
  cdq
  idiv  %esi
  test  %edx, %edx
  jz   notPrime'    ; JCC
       ; fall-thru into next CCF
  *inc  %esi    ; IDENT
  cmp  %esi, %ecx
  jl    nexti'    ; JCC
  jmp  [fallthrAddr3]

notPrime':  *xor  %eax, %eax    ; IDENT
  pop  %r11    ; RET
  mov  %gs:0xff39eb8(%rip), %rcx    ; spill %rcx
  movzx  %ecx, %r11b
  jmp  %gs:0xfc7ddee0(8*%rcx)
Fig. 2. The VMWare Hosted Architecture. VMWare Workstation consists of the three shaded components. The figure is split vertically between host operating system context and VMM context, and horizontally between system-level and user-level execution. The steps labeled (i)–(v) correspond to the execution that follows an external interrupt that occurs while the CPU is executing in VMM context.
Address space during the world switch

Host OS Context

Host OS -> VMM Transition

VMM Context

VMM -> Host OS Transition

Cross page

Linear Address space
The world switch

- First, save the old processor state: general-purpose registers, privileged registers, and segment registers;
- Then, restore the new address space by assigning %cr3. All page table mappings immediately change, except the one of the cross page.
- Restore the global segment descriptor table register (%gdtr).
- With the %gdtr now pointing to the new descriptor table, restore %ds. From that point on, all data references to the cross page must use a different virtual address to access the same data structure. However, because %cs is unchanged, instruction addresses remain the same.
- Restore the other segment registers, %idtr, and the general-purpose registers.
- Finally, restore %cs and %eip through a longjump instruction.
Protecting the VMM

- **cpl=3**
  - userspace (direct execution)
  - `%cs, %ds, %gs`

- **cpl=1**
  - kernel code/data
  - `%ds` (sharable)
  - VMM
  - TC
  - `%cs, %gs`

- **cpl=0**
  - VMM
  - `%cs, %ds, %gs`

- **pte.us=1**
  - Linear Address Space
  - 0xffc00000

- **pte.us=0**
Translator continuations

Source Instructions

Translation Cache

TC Backmap

1: mov 12(%ebp), %eax
2: inc 8(%eax)
3: mov %eax, <gs>:bt.tmp_eax
4: mov 0(%ebx), %eax
5: push #t4
6: mov %eax, <gs>:vcpu.eip
7: mov <gs>:bt.tmp_eax, %eax
8: jmp fastDispatch

...
Interpreted execution revisited: Bochs
Instruction trace cache

- 50% of time in the main loop
  - Fetch, decode, dispatch
- Trace cache (Bochs v2.3.6)
  - Hardware idea (Pentium 4)
  - Trace of up to 16 instructions (32K entries)
- 20% speedup
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i) {
    Bit32u op2_32, op1_32, diff_32;

    op2_32 = BX_READ_32BIT_REG(i->nnn());

    if (i->modC0()) { // reg/reg format
        op1_32 = BX_READ_32BIT_REG(i->rm());
        diff_32 = op1_32 - op2_32;
        BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
    } else { // mem/reg format
        read_RMW_virtual_dword(i->seg(),
                               RMAddr(i), &op1_32);
        diff_32 = op1_32 - op2_32;
        Write_RMW_virtual_dword(diff_32);
    }
    SET.LAZY_FLAGS_SUB32(op1_32, op2_32, diff_32);
}

• 20 cycles penalty on Core 2 Duo
Improve branch prediction

- Split handlers to avoid conditional logic
  - Decide the handler at decode time (15% speedup)
Resolve memory references without misprediction

- Bochs v2.3.5 has 30 possible branch targets for the effective address computation
  - Effective Addr = (Base + Index*Scale + Displacement) mod(2^AddrSize)
  - e.g. Effective Addr = Base, Effective Addr = Displacement
  - 100% chance of misprediction

- Two techniques to improve prediction:
  - Reduce the number of targets: leave only 2 forms
  - Replicate indirect branch point

- 40% speedup
# Time to boot Windows

<table>
<thead>
<tr>
<th></th>
<th>1000 MHz Pentium III</th>
<th>2533 MHz Pentium 4</th>
<th>2666 MHz Core 2 Duo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bochs 2.3.5</td>
<td>882</td>
<td>595</td>
<td>180</td>
</tr>
<tr>
<td>Bochs 2.3.6</td>
<td>609</td>
<td>533</td>
<td>157</td>
</tr>
<tr>
<td>Bochs 2.3.7</td>
<td>457</td>
<td>236</td>
<td>81</td>
</tr>
</tbody>
</table>
## Cycle costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Bochs 2.3.5</th>
<th>Bochs 2.3.7</th>
<th>QEMU 0.9.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register move (MOV, MOVSX)</td>
<td>43</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Register arithmetic (ADD, SBB)</td>
<td>64</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>Floating point multiply</td>
<td>1054</td>
<td>351</td>
<td>27</td>
</tr>
<tr>
<td>Memory store of constant</td>
<td>99</td>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>Pairs of memory load and store operations</td>
<td>193</td>
<td>98</td>
<td>14</td>
</tr>
<tr>
<td>Non-atomic read-modify-write</td>
<td>112</td>
<td>75</td>
<td>10</td>
</tr>
<tr>
<td>Indirect call through guest EAX register</td>
<td>190</td>
<td>109</td>
<td>197</td>
</tr>
<tr>
<td>VirtualProtect system call</td>
<td>126952</td>
<td>63476</td>
<td>22593</td>
</tr>
<tr>
<td>Page fault and handler</td>
<td>888666</td>
<td>380857</td>
<td>156823</td>
</tr>
<tr>
<td>Best case peak guest execution rate in MIPS</td>
<td>62</td>
<td>177</td>
<td>444</td>
</tr>
</tbody>
</table>
References

• A Comparison of Software and Hardware Techniques for x86 Virtualization. Keith Adams, Ole Agesen, ASPLOS'06

• Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, Edward Y. Wang, ACM TCS'12.

• Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. Darek Mihocka, Stanislav Shwartsman.