Goals for Today

- **Learning Objective:**
  - Define a taxonomy for virtualization architectures

- **Announcements, etc:**
  - “Informal Early Feedback” at end of class today
  - Midterm debrief forthcoming on Friday
  - MP2 extension: now due on March 23rd

**Reminder:** Please put away devices at the start of class
CS 423
Operating System Design:
Virtual Machines

Professor Adam Bates
Spring 2017
What’s a virtual machine?

• Virtual machine is an entity that emulates a guest interface on top of a host machine
  – Language view:
    • Virtual machine = Entity that emulates an API (e.g., JAVA) on top of another
    • Virtualizing software = compiler/interpreter
  – Process view:
    • Machine = Entity that emulates an ABI on top of another
    • Virtualizing software = runtime
  – Operating system view:
    • Machine = Entity that emulates an ISA
    • Virtualizing software = virtual machine monitor (VMM)

Different views == who are we trying to fool??
Purpose of a VM

• Emulation
  – Create the illusion of having one type of machine on top of another

• Replication (/ Multiplexing)
  – Create the illusion of multiple independent smaller guest machines on top of one host machine (e.g., for security/isolation, or scalability/sharing)

• Optimization
  – Optimize a generic guest interface for one type of host
Types of VMs

• Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.
  – Process/language virtual machines (emulate ABI/API)
  – System virtual machines (emulate ISA)
Taxonomy

• Language VMs
  – Emulate same API as host (e.g., application profiling?)
  – Emulate different API than host (e.g., Java API)

• Process VMs
  – Emulate same ABI as host (e.g., multiprogramming)
  – Emulate different ABI than host (e.g., Java VM, MAME)

• System VMs
  – Emulate same ISA as host (e.g., KVM, VBox, Xen)
  – Emulate different ISA than host (e.g., MULTICS simulator)
• Emulation: General technique for performing any kind of virtualization (API/ABI/ISA)

• Not to be confused with Emulator in the colloquial sense (e.g., Video Game Emulator), which often refers to ABI emulation.
• Problem: Emulate guest ISA on host ISA
Writing an Emulator

• Problem: Emulate guest ISA on host ISA

• Create a simulator data structure to represent:
  – Guest memory
    • Guest stack
    • Guest heap
  – Guest registers

• Inspect each binary instruction (machine instruction or system call)
  – Update the data structures to reflect the effect of the instruction
• Problem: Emulate guest ISA on host ISA
• Solution: Basic Interpretation, switch on opcode

```plaintext
inst = code (PC)
opcode = extract_opcode (inst)
switch (opcode) {
    case opcode1 : call emulate_opcode1 ()
    case opcode2 : call emulate_opcode2 ()
    ...
}
```
Emulation

• Problem: Emulate guest ISA on host ISA
• Solution: Basic Interpretation

new
inst = code (PC)
opcode = extract_opcode (inst)
routineCase = dispatch (opcode)
jump routineCase
...

routineCase
... call routine_address
jump new
Threaded Interpretation…

[ body of emulate_opcode1 ]
inst = code (PC)
opcode = extract_opcode (inst)
routine_address = dispatch (opcode)
jump routine_address

[ body of emulate_opcode2]
inst = code (PC)
opcode = extract_opcode (inst)
routine_address = dispatch (opcode)
jump routine_address
Note: Extracting Opcodes

- **extract_opcode (inst)**
  - Opcode may have options
  - Instruction must extract and combine several bit ranges in the machine word
  - Operands must also be extracted from other bit ranges

- **Pre-decoding**
  - Pre-extract the opcodes and operands for all instructions in program.
  - Put them on byte boundaries (intermediate code)
  - Must maintain two program counters. Why?
lwz   r1, 8(r2)
add   r3, r3, r1
stw   r3, 0(r4)

<table>
<thead>
<tr>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
</tr>
<tr>
<td>03</td>
</tr>
<tr>
<td>00</td>
</tr>
</tbody>
</table>

Note: Extracting Opcodes
Direct Threaded Impl.

- Replace opcode with address of emulating routine

<table>
<thead>
<tr>
<th>Routine_address07</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routine_address08</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routine_address37</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>00</td>
</tr>
</tbody>
</table>
Binary Translation

• Emulation:
  – Guest code is traversed and instruction classes are mapped to routines that emulate them on the target architecture.

• Binary translation:
  – The entire program is translated into a binary of another architecture.
  – Each binary source instruction is emulated by some binary target instructions.
• Can we really just read the source binary and translate it statically one instruction at a time to a target binary?
  – What are some difficulties?
Challenges

- Code discovery and dynamic translation
  - How to tell whether something is code or data?
  - Consider a jump instruction: Is the part that follows it code or data?

- Code location problem
  - How to map source program counter to target program counter?
  - Can we do this without having a table as long as the program for instruction-by-instruction mapping?
Things to Notice

• You only need source-to-target program counter mapping for locations that are *targets of jumps*. Hence, only map those locations.

• You always know that something is an instruction (not data) in the source binary if the source program counter eventually ends up pointing to it.

• The problem is: You do not know targets of jumps (and what the program counter will end up pointing to) at static analysis time!
  – Why?
Solution

• Incremental Pre-decoding and Translation
  – As you execute a source binary block, translate it into a target binary block (this way you know you are translating valid instructions)
  – Whenever you jump:
    • If you jump to a new location: start a new target binary block, record the mapping between source program counter and target program counter in map table.
    • If you jump to a location already in the map table, get the target program counter from the table
  – Jumps must go through an emulation manager. Blocks are translated (the first time only) then executed directly thereafter
Informal Early Feedback