Goals for Today

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

• Announcements, etc:
  • MP2 Due Dates — Wed or Fri (-10%), Demos Thur or Sat
  • Midterm debrief forthcoming, probably on Monday
  • Special Topics division of labor due today.

**Reminder:** Please put away devices at the start of class
• Creation of an isomorphism that maps a virtual guest system to a real host:
  – Maps guest state $S$ to host state $V(S)$
  – For any sequence of operations on the guest that changes guest state $S_1$ to $S_2$, there is a sequence of operations on the host that maps state $V(S_1)$ to $V(S_2)$
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)
System VMs

- Implement VMM (ISA emulation) on bare hardware
  - Efficient
  - Must wipe out current operating system to install
  - Must support drivers for VMM

- Implement VMM on top of a host OS (Hosted VM)
  - Less efficient
  - Easy to install on top of host OS
  - Leverages host OS drivers
• Emulate one ISA on top of another
  – Typically runs on top of host OS (e.g., install Windows compiled for IA-32 on top of MacOS running on PowerPC)
  – Note: this is different from a process virtual machine that emulates the Windows interface and user IA-32 instructions on top of MacOS running on PowerPC
• Problem: Emulate guest ISA on host ISA
• Problem: Emulate guest ISA on host ISA
• Solution: Basic Interpretation

```
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
• `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)
• 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
• Program is translated into chunks called “dynamic basic blocks”, each composed of straight machine code of the target architecture
  – Block starts immediately after a jump instruction in the source binary
  – Block ends when a jump occurs
• At the end of each block (i.e., at jumps), emulation manager is called to inspect jump destination and transfer control to the right block with help of map table (or create a new block and map table entry, if map miss)
Dynamic Binary Translation

Start with SPC

Look up SPC→TPC in map table

Hit in Table?

Yes

Branch to TPC and execute block

Get SPC of next block

No

Translate new block

Store new SPC→TPC entry in table
• Translation chaining
  – The counterpart of threading in interpreters
  – The first time a jump is taken to a new destination, go through the emulation manager as usual
  – Subsequently, rather than going through the emulation manager at that jump (i.e., once destination block is known), just go to the right place.
• What type of jumps can we do this with?
• Translation chaining
  – The counterpart of threading in interpreters
  – The first time a jump is taken to a new destination, go through the emulation manager as usual
  – Subsequently, rather than going through the emulation manager at that jump (i.e., once destination block is known), just go to the right place.
• What type of jumps can we do this with?
  • Fixed Destination Jumps Only!!!
Register Indirect Jumps?

- Jump destination depends on value in register.
- Must search map table for destination value (expensive operation)
- Solution?
  - Caching: add a series of if statements, comparing register content to common jump source program counter values from past execution (most common first).
  - If there is a match, jump to corresponding target program counter location.
  - Else, go to emulation manager.