Goals for Today

• **Learning Objective:**
  • Understand primitives for interpretation versus translation

• **Announcements, etc:**
  • Midterm debrief will have to happen after spring break; sorry!
  • MP2 extension: **now due on March 25th**
  • MP3 released March 27th
  • MP2.5 (Extra Credit) release on March 27th also

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

Professor Adam Bates
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??**
Emulation

• Problem: Emulate guest ISA on host ISA
• Solution: Basic Interpretation, switch on opcode

```
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

\[
\begin{align*}
\text{new} & \quad \text{inst} = \text{code (PC)} \\
\text{opcode} & \quad = \text{extract_opcode (inst)} \\
\text{routineCase} & \quad = \text{dispatch (opcode)} \\
\text{jump routineCase} \\
\text{...} \\
\text{routineCase} & \quad \text{call routine_address} \\
\text{jump new}
\end{align*}
\]
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...

– Also, must maintain two program counters. Why?
Note: Extracting Opcodes

**Example: MIPS Instruction Set**

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>LW</td>
<td>r1, 8(r2)</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td>ADD</td>
<td>r3, r3, r1</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td>SW</td>
<td>r3, 0(r4)</td>
<td></td>
</tr>
</tbody>
</table>
• Replace opcode with address of emulating routine

<table>
<thead>
<tr>
<th>Routine_address07</th>
<th>07</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routine_address08</th>
<th>08</th>
<th>03</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routine_address37</th>
<th>37</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
• 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 binary translation
  – How to tell whether something is code or data?
  – We encounter a jump instruction: Is word after the jump instruction 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

Edit: The original automata didn’t execute the current block unless there was a hit!
• 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.