Intro to Lecture

1) Lab 3 due.
2) Let's talk about some fundamentals (CS140).
  - Privilege separation (rings)
    - privilege instructions
    - exceptions: faults and traps
  - Interaction with rings and MMU via page table entries

Intro to Paper

Q: What is emulation? What is virtualization?

Paper uses definition from Popek and Goldberg's 1974 paper. A VMM must have:

  1) Fidelity. Software running in VMM runs identically to software on real machine. (Barring timing effects)

  2) Performance. The machine executes almost all of the instructions itself.

  3) Safety. VMM manages hardware resources. Isolation between VMs.

Emulation doesn't satisfy 2).

Q: How do we virtualize?

A: "Classically": trap-and-emulate. VMware: binary translate untrappable things.
   Future (present, past): Hardware

Trap-And-Emulate

Idea: Keep a shadow of physical CPU structures. Trap on priviledged instructions that modify/read these CPU structures and emulate/replay their effects.

Tracing:
  Q: What about memory mapped I/O devices?
  A: Use MMU to page-fault on all read/writes.

  Q: What about priviledged structures that are memory mapped?
  A: Use MMU to page-fault on all read, writes, or both.

What's the difference between hidden vs. true page faults?

A: Hidden: faults that occur BECAUSE of virtualization. Don't forward.
A: True: faults that occur due to guest OS setup. Forward.

x86: Trap-and-Emulate Not Possible?

Q: Why?
A: Don't have traps on some priviledged actions are done!
A: Machine reveals state (CPL, etc.)!

Examples:
  * popf silently ignores changes to interrupt flag (guest kernel should change)
    pushf reveals *real* interrupt flag
  * Can read %cs: no trap, CPL is in there.

What real x86 state do we have to hide (i.e. != virtual state)? (thanks, 6.828!)
  * 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

Q: So paper proposes one way to get around this: dynamic binary translation, and
   says in future (present), x86 is clasically virtualizable. But, how else can
   we get around this?
A: Emulate, but then this isn't virtualization.
A: Statically replace privledeged instructions with int 3: emulate in VMM.
  - won't this mess up the code?
  - nope: INT 3 is 1 byte!
  - what about code generation?......!

Dynamic Binary Translation
==========================

Idea: Translate the binary as it executes.
  As Dawson likes to say, super simple: fn :: x86 -> x86

Example: let's binary translate the following:
(example from paper in AT&T instead of Intel)

```
      isPrime:
PC ->   mov %edi, %ecx
        mov $2, %esi
        cmp %esi, %ecx
        jge prime
      nexti:
        mov %ecx, %eax
        ...
      prime:
        ...
        ret
      notPrime:
        ...
        ret
```

  Note: %edi contains the first argument, a (from calling convention)

So, PC is at isPrime. So, we binary translate.

  - Read/parse some number of instructions into IR objects grouped into one
    translation unit (TU).

    Q: How many?
    A: Up to 12 or until the first "terminating instruction" (control flow).

    Q: What is a terminating instruction?
    A. Any that does control flow. e.g.: jmp, ret, call, (mov %eax, %rip), etc.

    Q: Why this limit?
    A: Want to keep instructions being translated in static memory.

Translation pipeline (so far) is:

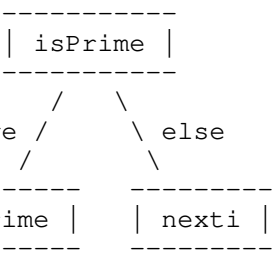
```
  ---------------
  | instruction |  -> (decoder) -> (wrap in IR) -> (add to TU)
  ---------------
```

So now we have a TU. For our example above, we have the following TU:

  isPrime:

```
    mov %edi, %ecx
    mov $2, %esi
    cmp %esi, %ecx
    jge prime
  nexti:
    ...
  prime:
    ...
```

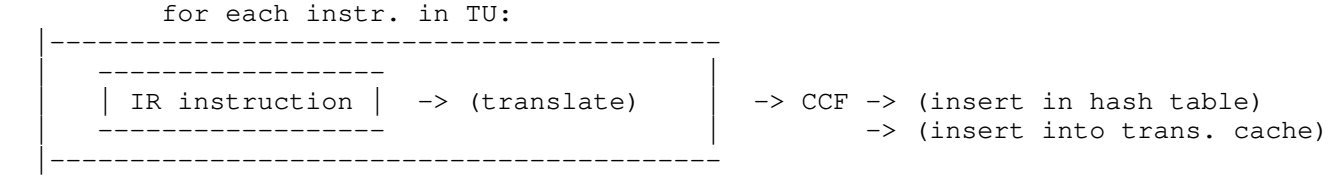
In compiler basic block notation, we have:



Now, we actually translate each instruction. Two types of translations:

1) IDENT: do nothing to the instructions
2) Non-IDENT: instuctions becomes something else, usually several instructions

Translation pipeline is:



Hash Table:
real PC of basic block -> address of CCF

First three instructions are IDENT. jge instructions is not because instructions are no longer in the same place as they used to be. So for a jump, we (may) have to figure out where to go dynamically. At the very least, we may have not yet translated the TU at that address. So, need to insert a call into the translator for each possible branch (have two, see diagrama above).

So the translated unit becomes:

```
  isPrime':
    mov %edi, %ecx
    mov $2, %esi
    cmp %esi, %ecx
    jge [takenAddr = prime]    - calls the translator with address `prime`
    jmp [fallThrAddr = nexti]  - calls the translator with address `nexti`
```

And we're done! So now we execute the code and keep going going until...?

Q: Until when?
A: Until something forces us to jump back into the translator! In the example above, the bottom two `jmp` instructions would do this.

Q: What exactly happens when we hit `jgq [takenAddr = prime]`?
A: Calls into the translator to translate the TU beginning at `prime`.
A: Probably something like:
   mov $prime, %gs:0xff890ec9b
   jmp $translator -> which then jumps to the CCF address for $prime

  Q: Do we have to do this jump to translate everytime?
  A: Afer the TU being jumped to has been translated, we can patch the old CCF
     to refer directly to the address of the TU's CCF. Then will jump from one
     CCF into another. This is "chaining" the CCFs.

  Q: What if we translate 12 instructions without a "terminating" instruction?
  A: Add a call into the translator at end with where to go next.

Q: What's this %gs?
A: Segment regsiter. Segmented memory: can partition memory; register selects
   which partition to refer to. VMM keeps its structures in segment pointed to
   be %gs. Needs to translate each reference to %gs in guest.

Q: With that, what's going on with the ret translation for notPrime?
```
  xor %eax, %eax
  pop %r11                          <-- stores returns address in %r11
  mov %rcx, %gs:0xff39eb8(%rip)     <-- stores %rcx in VMM memory
  movzx %r11b, $ecx                 <-- stores lower 8 bytes of ret addr in %ecx
                                    <-- this clobbers %rcx (ecx = lower 32 of rcx)
                                    <-- used by translator in some way
  jmp %gs(0xfc7dde0(8*%rcx))        <-- hash table lookup for real address
                                    <-- or jumps into translator to do lookup
                                    <-- and restore %rcx
```

Q: What other things do we have translate non-IDENT:
  * all control flow (TC cache is not in the same place as original bianry)
    * PC relative addressing, (in)direct control flow
  * priviledged instructions (cli, can be replaced with fast versions)
  * accesses to %gs: used by VMM for its data.

Q: Paper says we don't have to translate user-mode code. Why not?
A: ...

Adaptive Binary Translation

Doing the above translation for priveledged instructions is nice because we can translate typically expensive things (like cli) into cheap things (like a memory store). What about other traps?

We still have a bunch of traps from virtual memory stuff.

Idea: Make a non-IDENT translation for instructions that frequently cause these traps. In short: take something that would trap (say, a change of a page table) and translate that into the direct instructions necessary to update the VMM's page table metadata.

How?
  1) retranslate ccf to be more efficient
  2) add jmp at beginning of old ccf to new ccf
  3) that's it.

Q: What if instructions would not longer trap? (IE: address stopped corresponding
   to a page table or something...)
A: Revert the changes. Remove jmp from original ccf. Add jump to original ccf to
   old ccf for code pointing there.

Hardware Virtualization

Basically, Intel and AMD made it possible to trap on priveledged stuff.

Setup:
  * New CPL (ring): -1 (host mode).
  * VMM adds a VM entry for each virtual machine CPU

    -> Need VMCS (VMCB in paper) for each one. Stores information about the
       virtul CPU.
  * VMM does (vmlaunch (vmrun in paper), vmresume) to start VM
  * CPU handles almost all priviledged operations, virtualizes itself.
  * vmexit returns control to VMM on traps and other conditions
    * e.g, syscall, hypercall, (port) I/O, page faults, etc.
      * No EPT (MMU virtualization) in VT-x (VMX) version of paper.
    * programmable

Q: So, is the VMM doing anything anymore?
A: Yes!
  1. Programming the CPU to do VT-x.
  2. Still shadowing memory (no EPT - even with EPT need control)
  3. Implementing virtual devices (I/O for emulated devices).
  4. Handling I/O (MMIO).
  5. Managing multiples VMs.

Fork example: guest user-mode process calls fork(). What happens?

1) Well, this is a syscall. CPU handles this directly, invoking guest kernel.
  * CPU changes virtual state by itself.
2) Guest kernel needs to modify paging structures for new process.
  * This traps into VMM. Shadow page table updated, etc.
3) Guest switches address spaces (%cr3).
  * This traps into VMM. Virtual %cr3 updated.
4) Child runs. Page faults may occurs.
  * These trap into VVM. Shadow page tables updated or faults forwarded.

Q: What of child's sycalls?
A: Handled by CPU directly!

Q: What about popf now?
A: CPU handles it correctly, directly. Stored in VMCS.