Administrivia

- **Next week office hours moved to 3:30pm Monday**
  - Monday is Add/drop deadline
  - I'll also be around at 5pm Monday for non drop-related questions
- **Must vote on final exam date – May 3 or May 10**
  - Final will cover topics from all semester
  - Final will supersede midterm if you do better
- **Midterm evaluations:**
  - *Slow down!* – Next class 1 paper
  - Want example questions – I'll try to propose some in lecture with Q:

Uses of unsafe code

- **Extensible applications**
  - POSTGRESS database's extensible type system
  - Third party code for loading into Quark Express
  - "Executable content" from the web
- **Saving kernel/user crossings**
  - Packet filters (e.g., bpf for tcpdump)
  - Applications-specific virtual memory management
  - Active messages (application-specific msg. handlers)
- **Hardware VM provides protection**
  - Should we just run unsafe code in separate process?

Cross-address-space calls are expensive

- **System call overhead much higher than procedure**
  - Requires trapping into the kernel
  - Often requires draining the processor pipeline
- **Switching address spaces increasingly expensive**
  - On some architectures requires flushing the TLB
  - Increases cache pressure
  - Cache/TLB miss service times increasingly expensive compared to faster and faster cycle times
- **Kernel must copy arguments back and forth between address spaces**
  - Change page mappings, etc.

Running unsafe code also gives *control*

- **Example: Exokernel OS**
  - Goal: Let applications manage resources as much as possible
- **Don't hardcode TCP/IP or other protocols**
- **Instead, download packet filters into kernel**
  - Express which packets an application wants to see
  - By downloading filters, kernel can ensure no conflicts
  - Also ensures apps don't leak information on other's pkts
- **DPF (dynamic packet filter) created code on the fly**

Exokernel disk abstraction

- **How to multiplex disk with untrusted apps?**
  - Need metadata–i.e., for a file, what blocks to use
  - Don't want to hard-code metadata formats
- **Solution: UDFs (untrusted deterministic functions)**
  - Download metadata interpretation code
  - UDF takes metadata, outputs list of blocks
  - Kernel checks metadata updates by output of UDF
  - Downloading ensures that UDFs are deterministic

Challenges of untrusted code

- ***Fault domain*—logically separate portion of A.S.**
  - Each untrusted component runs in its own fault domain
- **Prevent FDs from trashing each other's memory**
- **Prevent FDs from jumping to arbitrary locations**
- **Prevent code from accessing operating system**
  - Otherwise, e.g., could execute arbitrary programs
- **Other possible goals:**
  - Prevent FDs from *reading* each other's memory
  - Prevent infinite loops
  - Bound physical memory utilization

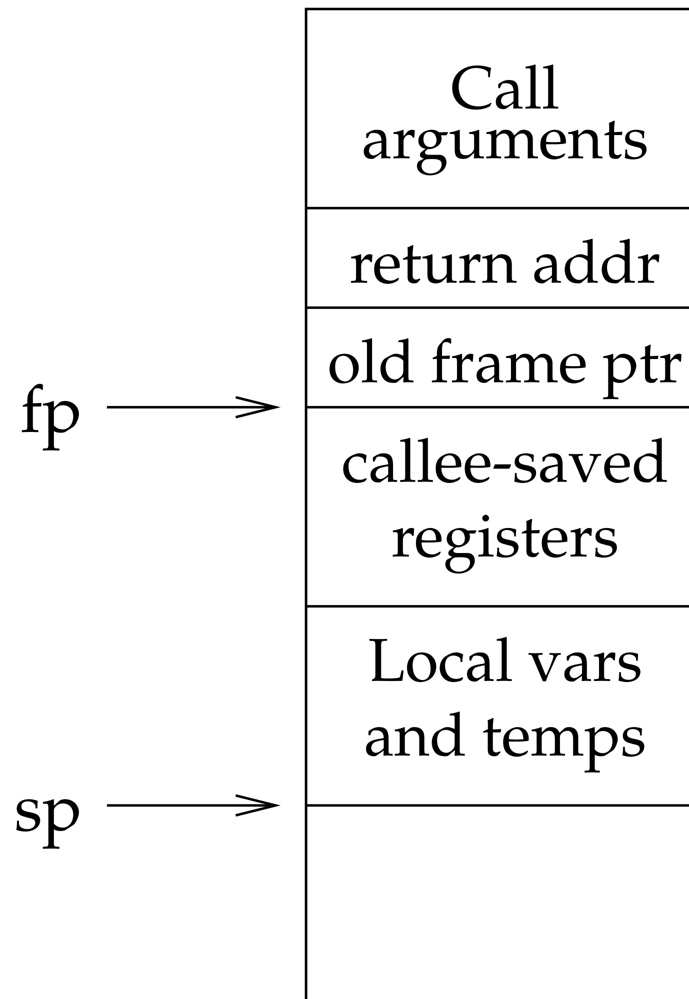
Software fault isolation

- **Goal: Make fault isolation cheap enough that developers can ignore performance impact**
- **General approach:**
  - Modify compiler to generate “safe” code
  - Verifier can check code is safe before loading/running it
- **Alternate approach: *binary patching***
  - Rewrite unsafe binaries to be safe
  - Doesn’t tie system to one compiler/language
  - Unfortunately, binary rewriting hard to do

Review: Typical RISC instruction sets

- **Have 31 general-purpose integer registers**
  - Instruction set treats all registers identically
  - Convention dictates certain uses (e.g., stack ptr, ...)
  - Across calls, some regs caller-saved, some callee-
  - All ALU operations occur on registers
- **Memory accessed w. load/store instructions only**
  - `LD rd, offset(rp)`   `ST rs, offset(rp)`
- **All instructions 32 bits (and must be aligned)**
  - Makes it easy to check each instruction in code

Review: calling conventions



SFI implementation

- **Divide virtual address space into segments**
  - All addresses in a segment share same prefix
  - Not all virtual addresses in segment need to be valid
- **Each fault domain has two segments**
  - Code segment and separate data segment
  - Q: Why not use one combined segment?
- **Go over code identifying *unsafe instructions***
  - Any store or jump that can't be statically verified
  - PC-relative branches OK, stores to static vars often OK
  - Insert checking code before instructions that are not OK

Segment matching

- **Use dedicated registers to hold addresses**
- **Always check segment ID of target address of store**

```
dedicated-reg <= target address
scratch-reg <= (dedicated-reg >> shift-reg)
compare scratch-reg segment-reg
trap if not equal
store value dedicated-reg
```

- **Adds 4 instructions to every store**
- **Q: Why use dedicated register for store address?**

Address sandboxing

- Segment matching good for debugging, but slow
- Instead of checking segment IDs, can just set them:

```
dedicated-reg <= target-reg & and-mask-reg
dedicated-reg <= dedicated-reg | segment-reg
store value dedicated-reg
```

- Now requires only 2 extra instructions per store
- Again, dedicated register prevents harm if code jumps to middle of store sequence

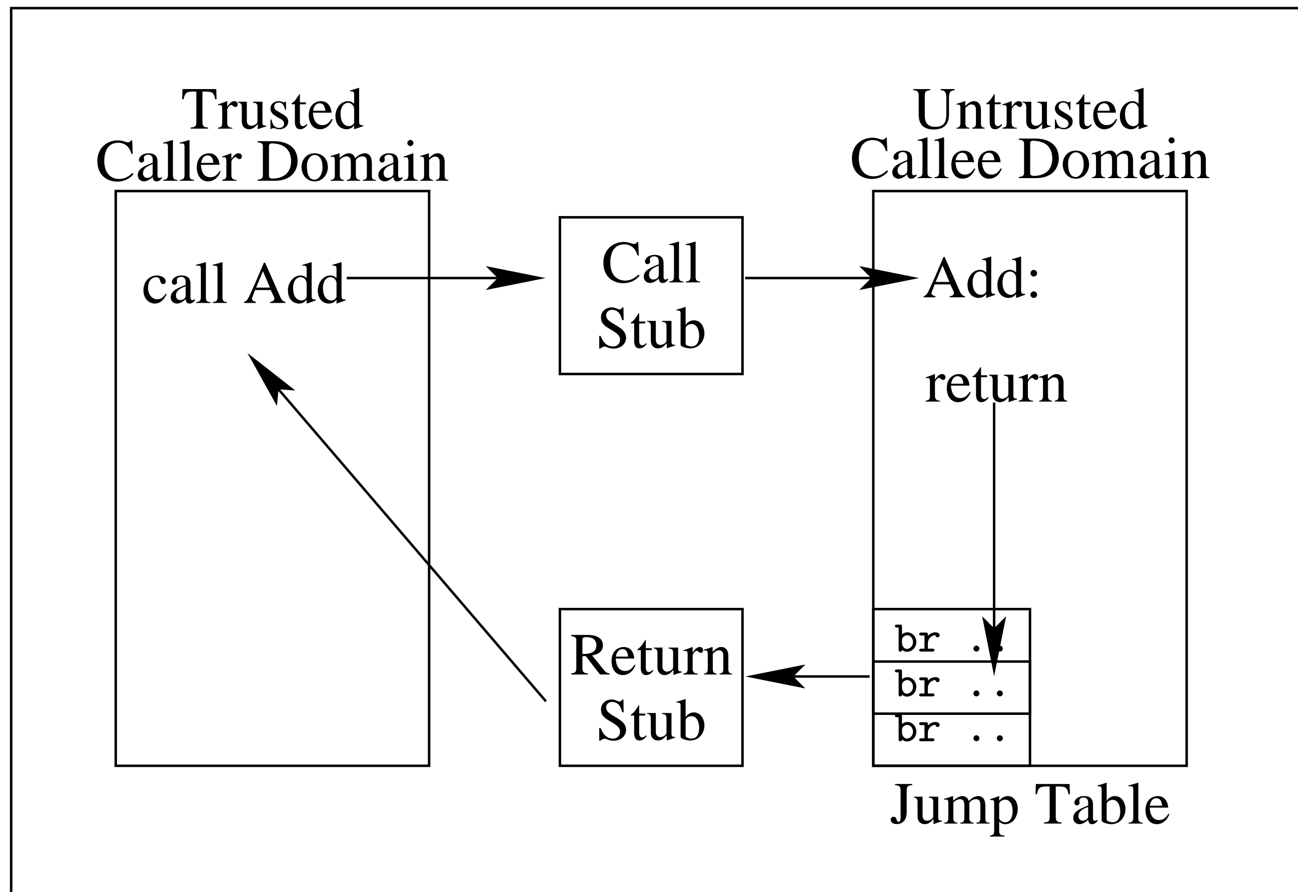
Optimizations

- **Traditional compiler optimizations**
  - E.g., might move sandboxing out of a loop
- **Guard zones at each end of data segment**
  - Load/store instructions tage address reg. & offset
  - Unmapped zones larger than maximum ld/st offset
  - Means only register need be sandboxed, not full addr
  - Sandbox the stack pointer only when it is set
  - Avoid sandboxing SP if adjusted by small amount and used before next control transfer

Cross-domain calls

- ***Jump table*** **contains allowed exit points from FD**
  - Each jump table entry is a control transfer instruction (address hard-coded into instruction, so no register use)
  - Explicitly enumerates allowed calls between each 2 FDs
  - Jump table trusted, and in read-only code segment
- **Jump table entries transfer control to *stubs***
  - Must save any caller-saved registers (can't trust target)
  - Copy arguments of call from caller's segment to target's

Fig 4



- **Q: Why not embed stubs directly in segment?**

Sharing memory accross domains

- **Read sharing is not a problem**
- **If we need write sharing, use VM hardware**
  - Just map the same page into multiple segments in same A.S.
- **Slight trickiness: pointer comparisons**
  - Don't compare aliased ptrs w. different segment IDs
  - Give shared region canonical address
  - Fix pointer for write access (automatic w. sandboxing)

Limitations of SFI

- **Performance**
  - Usually good, but slowdown bad for packet filters, . . .
- **Harder to implement on some architectures**
  - E.g., x86 has variable-length, unaligned instructions (would have to do more expensive checks on jumps)
  - x86 has fewer registers (can't dedicate 5 of them)
  - Most x86 instructions affect memory (more sandboxing)
- **Compiler and verifier tightly bound**
  - Once verifier deployed, might be hard to make further improvements in compiler

Proof carrying code

- **Goal: Safely run code in unsafe languages**
  - Download packet filter into kernel
  - Run photoshop plug-in with CPU-intensive inner loop
- **Ensure code safety even where SFI can't**
  - Might not want to be restricted to SFI's idioms
- **Idea: Accompany code with its proof of correctness**
  - Safe language compiler can output proof of correctness
  - Use hand-crafted proof for hand-written assembly
  - Doesn't matter how you generate the code, as long as it's provably safe, you can run it

Example

```
datatype T = Int of int | Pair of int * int

fun sum (l : T list) =
  let
    fun foldr f nil a = a
      | foldr f (h::t) a = foldr f t (f(a, h))
  in
    foldr (fn (acc, Int i) => acc + i
            | (acc, Pair (i, j)) => acc + i + j)
          l 0
  end
```

How data is layed out in memory

- See Fig. 4
- Integers just stored in memory
- Pairs stored as pointer to two elements
- Unions as pointer to pair $\langle\{1, 0\}$, pointer to value$\rangle$
- List as linked list of pairs w. value in first cell

Assembly code [Fig. 5]

```
sum:  INV rm |- r0 : T list
      MOV r1, 0
 L2:  INV rm |- r0 : T list ^ rm |- r1 : int
      BEQ r0, L14
      LD  r2, 0(r0)   % load head
      LD  r0, 4(r0)   % load tail
      ...
      [set r2 to Int, or sum of Pair elements]
      ...
L12:  ADD r1, r2, r1
      BR  L2
L14:  MOV r0, r1
      RET
```

Proving correctness

- **Need to show that if precondition holds on entry, postcondition will hold on return**
  - $Pre \equiv \mathbf{r_m} \vdash \mathbf{r_0} : \text{T list}$ (stated in line 0)
  - $Post \equiv \mathbf{r_m} \vdash \mathbf{r_0} : \text{int}$
- **Add loop invariant at line 2**
  - $\mathbf{r_m} \vdash \mathbf{r_0} : \text{T list} \land \mathbf{r_m} \vdash \mathbf{r_1} : int$
  - Will this be true when first crossed?
  - After loop interation?
  - Does this invariant imply the postcondition?

Note: You don't need to understand LF or pp. 7-14 of PCC paper

[stretch break]