Software Fault Isolation

CS 6v81.002 Language-Based Security

January 23, 2008

The Problem: What is Fault Isolation?

- Need trusted and untrusted code to cooperate
  - Examples:
    - Micro-kernel design
    - Extensible applications
    - High I/O processes
- Fault Isolation:
  - Prevent faults from affecting other code
  - Could overwrite control or data

Traditional Solution

- Hardware-based Fault Isolation
  - Run modules in separate address spaces
  - Communicate via Remote Procedure Call
    - Switch to kernel mode
    - Copy arguments
    - Save/Restore registers
    - Switch address spaces
    - Return to user mode
  - Context switches are expensive

Software Fault Isolation

- Use Inline Reference Monitors to enforce "address spaces" in software
  - Fault domains
  - Reduce the context switch overhead
- Fault domains can not:
  - Modify each others' data
  - Execute each others' code (Except through RPC)
- Segment Matching vs. Sandboxing

Potential Problems

- Modifying control
  - Separate the Data and Control segments
- Jumping past IRMs
  - Use dedicated jump registers
- Shared resources
  - Make the OS keep track
  - Or, have a trusted fault domain manage resources
- Shared data
  - Map shared data into multiple fault domains

Limitations

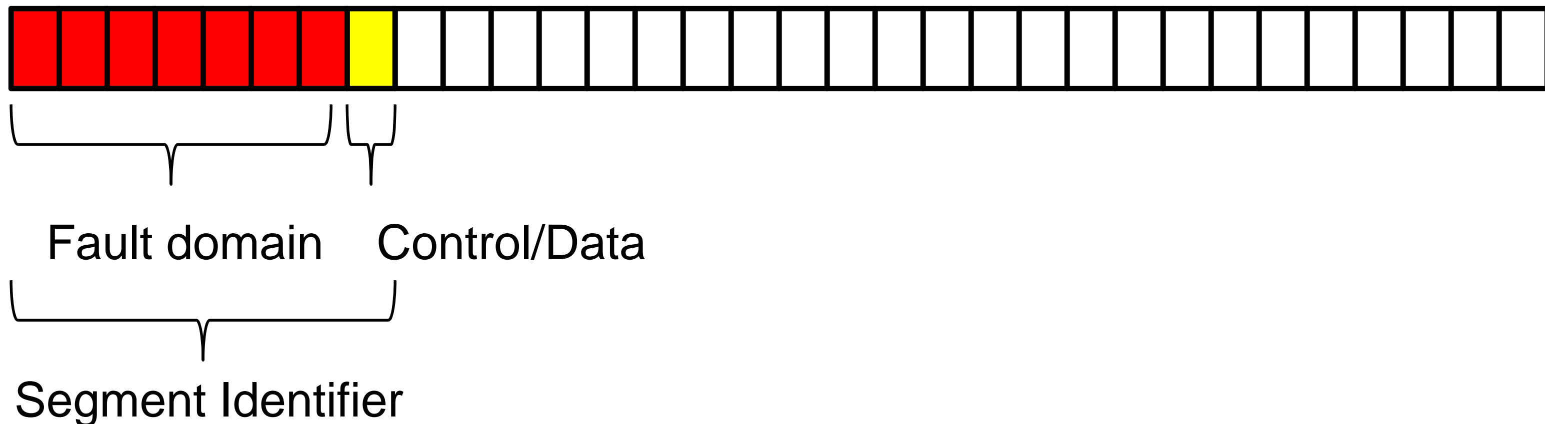
- Limited register systems
  - Limit control flow to statically verifiable jumps
- CISC systems
  - Complex instructions
    - Auto-looping computations, etc.
  - Easy with RISC load/store architecture

Implementation

- Two options:
  - Modify code at runtime
  - Compile encapsulated object code
    - Verify at runtime
- Wahbe, Lucco, Anderson, and Graham (*ACM SIGOPS OSR, 1993)* pick #2
  - Modifying code at runtime is too hard
    - Have to reduce number of registers
    - Identify code and data

IRM Details

- Divide up address space into segments
- Each fault domain has two segments
  - Segment addresses share an upper bit pattern
  - Control and data segments are adjacent



Where are IRMs needed?

- Statically analyze object code
  - Can verify:
    - PC-relative branches
    - Immediate addressing
  - Can’t verify
    - Jumps through registers (procedure returns, etc.)
    - Stores to register targets
- Insert IRMs before unsafe operations

IRM for Segment Matching

```
dedicated-reg ← target address
scratch-reg ← (dedicated-reg>>shift-reg)
compare scratch-reg and segment-reg
trap if not equal
store/jump uses dedicated reg
```

- Uses four dedicated registers
  - Shift register
  - Control segment identifier
  - Data segment identifier
  - Dedicated register

IRM for Sandboxing

```
dedicated-reg ← target-reg&and-mask-reg
dedicated-reg ← dedicated-reg|segment-reg
store/jump uses dedicated-reg
```

- Uses five dedicated registers
  - Segment identifier mask
  - Control segment identifier
  - Data segment identifier
  - Control address (once sandboxed)
  - Data address (once sandboxed)

Data Sharing

- Read-only sharing is trivial
- Read-write sharing
  - Lazy pointer swizzling
    - "Alias" shared region into each fault domain
      - Make sure lowest bits are shared
    - Works for sandboxed modules
  - Shared segment matching
    - Bitmap register for accessible segments

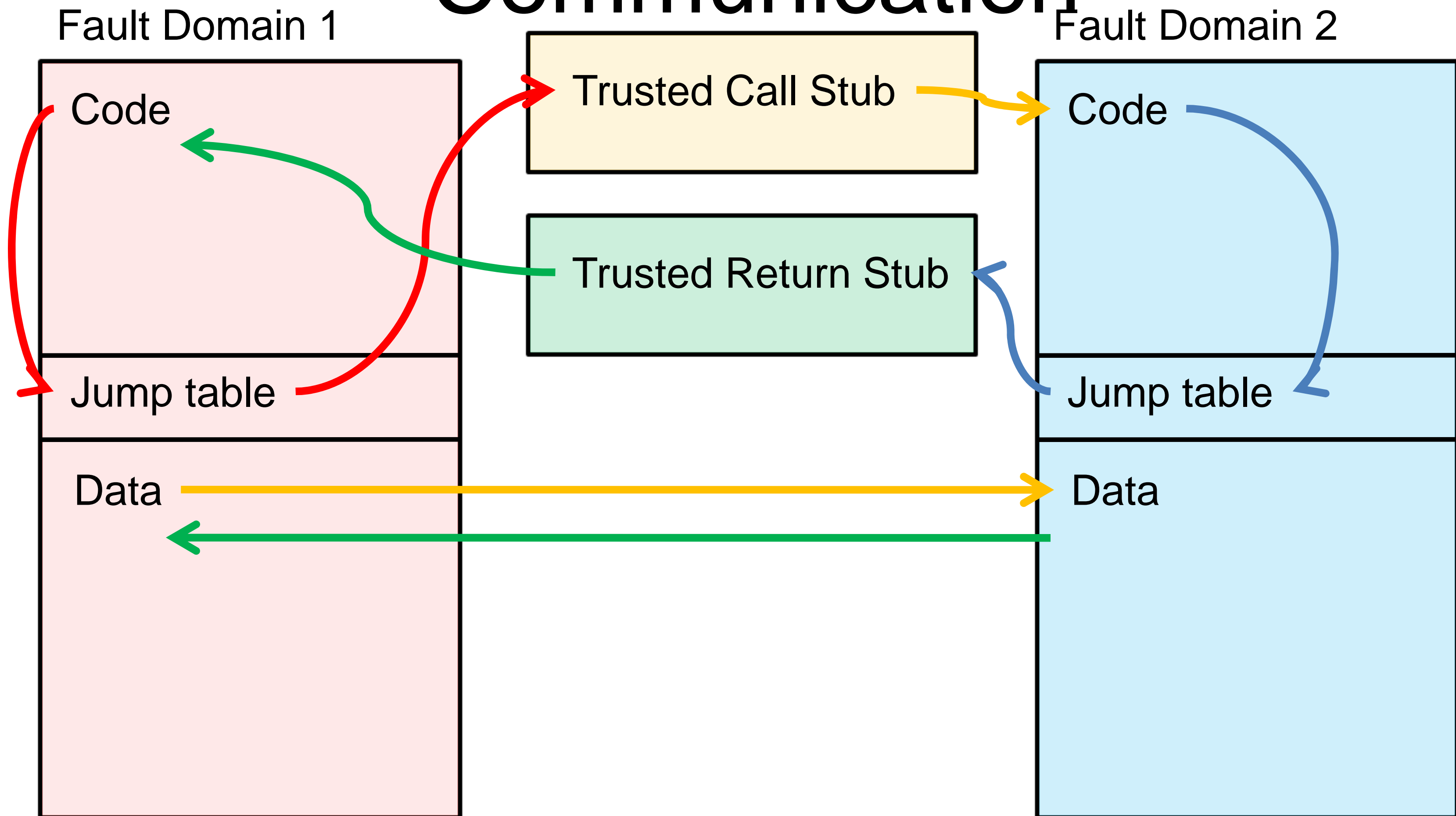
Verification

- Divide the program into *unsafe regions*
  - Starts with:
    - Modification of dedicated store/jump register
  - Ends with one of the following:
    - Next instruction is a store/jump to dedicated register
    - Next instruction is guaranteed not to be executed
    - No more instructions in the segment
- All stores and jumps use dedicated register
  - check dedicated registers are valid at region exit

Cross Fault Domain Communication

- Create trusted stubs to handle RPC
  - For each pair of fault domains
  - The stub:
    - Copies arguments between fault domains
    - Store/restore registers
    - Switches the execution stack
    - Validates dedicated registers

Cross Fault Domain Communication



Performance

- Encapsulation overhead
  - About 5%
  - Least noticeable for floating point and I/O
- Fault domain crossing
  - Order of magnitude faster than old solutions

When to use SFI

- Savings are dependent on:
  - t_d: Percent time spent in untrusted (slower) code
  - h: The overhead of encapsulated code
  - t_c: Percent time spend crossing fault domains
  - r: the ratio of SFI crossing time to best alternative

$$savings = (1 - r)t_c - ht_d$$

- Use when:
  - Not much time is spent in untrusted code
  - Fault domains are crossed frequently

Future Work

- CISC solution (next week)
- Limited register machines
- Runtime program-rewriting

References

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203-216, December 1993.

Discussion Questions

- Is it possible to use fewer registers?
- What are the relative merits of sandboxing and segment matching?
  - What do hardware systems enforce?
- How does this relate to the IRM applications we discussed last week? Can anything we talked about improve on this?
- Any other thoughts or questions?