

---

# Software-Based Memory Isolation

Greg Morrisett  
Harvard University

---

# Some References:

---

R.Wahbe *et al.* “Efficient Software-Based Fault Isolation.” In *Proceedings of the 14th ACM Symp. on Operating Systems Principles*, pages 203--216, December 1993.

C.Small. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems, *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, June 1997.

S.McCamant & G.Morrisett. Efficient, Verifiable Binary Sandboxing for a CISC Architecture.  
<http://people.csail.mit.edu/people/smcc/projects/pittsfield/usenix-sec/sfi.pdf>.

# Motivation

---

Many systems need to run 3rd party code.

- Kernels: drivers, modules, ...
- Servers: servlets, scripts, ...
- Although “trusted”, not “trustworthy”

The *right* way to do this is to fork:

- Keeps kernel/server state *isolated*.
- Can attenuate rights (e.g., change uid.)
- Can ensure availability by using only asynchronous communication.

# The Problem

---

Everyone uses shared memory and threads:

- A process fork costs a lot compared to a thread (usually thread pool)?
- Communication costs are greater:
  - Signals, pipes, etc: must go through kernel.
  - We end up copying data to and from the server.
  - We can use shared pages and copy-on-right *if* the data are relatively large and contiguous *and* there's no sensitive data on the pages.
  - In general, about 100x the cost of a proc. call.
- In some environments, it's not an option.
  - e.g., PDA or Phone without VM support.

# The Question:

---

From a security perspective, strong (process-based) isolation is better than shared memory and threads.

Can we avoid the overheads and limitations of the OS/Hardware-based enforcement mechanisms and still achieve isolation?

# Software Fault Isolation (SFI)

---

- Wahbe et al. (SOSP'93)
- Use a *software-based* reference monitor to isolate components into logical address spaces.
  - conceptually: check each read, write, & jump to make sure it is within the component's logical address space.
  - hope: communication as cheap as procedure call (minimize copying, costs of context switching, etc.)
  - worry: overheads of checking will swamp the benefits of communication.
- Note: doesn't deal with other policy issues
  - e.g., availability of CPU, attenuation of rights, etc.
  - Not ideal, but definitely an improvement over shared memory services.

# One Way to SFI

---

```
while (true) {
    if (pc >= codesz) exit(1);
    int inst = code[pc], rd = RD(inst), rs1 = RS1(inst),
        rs2 = RS2(inst), immed = IMMED(inst);
    switch (opcode(inst)) {
        case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
        case LD:  int addr = reg[rs1] + immed;
                 if (addr >= memsz) exit(1);
                 reg[rd] = mem[addr];
                 break;
        case JMP: pc = reg[rd]; continue;
        ...
    }
    pc++;
}
```

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
```

# Pros & Cons of Interpreter

---

## Pros:

- easy to implement (small TCB.)
- works with binaries (high-level language-independent.)
- easy to enforce other aspects of OS policy

## Cons:

- terrible execution overhead (x25? x70?)

but it's a start...

# Partial Evaluation (PE)

---

A technique for speeding up interpreters.

- At load time, we know what the code is.
- So *specialize* the interpreter to the code.
  - unroll the loop – one copy for each instruction
  - specialize the switch to the instruction
  - compile the resulting code
  - Voila!

# Example PE

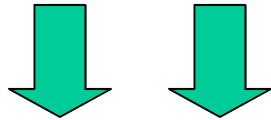
---

Original Binary:

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
...
```

Interpreter

```
while (true) {
    if (pc >= codesz) exit(1);
    int inst = code[pc];
    ...
}
```



Specialized interpreter:

```
reg[1] = reg[2] + reg[3];
addr = reg[3] + 12;
if (addr >= memsz) exit(1);
reg[4] = mem[addr];
pc = reg[4];
```

Resulting Compiled Code

```
0: add r1,r2,r3
1: addi r5,r3,12
2: subi r6,r5,memsz
3: jab _exit
4: ld r4,r5(0)
...
```



# Control Flow is an issue...

---

- Suppose the source code is:

```
0x00:  beq  r3, r4, 42
```

```
...
```

```
0x41:  ld   r2, r1(0)
```

```
0x42:  addi r2, r2, 1
```

- We'll insert code before the ld, so we have to adjust the branch target to the new address.

# Computed Jumps?

---

- In general, we may not know where we're jumping at translation time:

```
0x00:  ld r3, r1(10)
```

```
0x01:  jmp r3
```

```
...
```

```
0x41:  ld r2, r1(0)
```

```
0x42:  addi r2, r2, 1
```

- So how do we adjust the jump?

# Another Way to See This

---

```
while (true) {
    if (pc >= codesz) exit(1);
    int inst= code[pc], rd= RD(inst), rs1= RS1(inst),
        rs2 = RS2(inst), immed = IMMED(inst);
    switch (opcode(inst)) {
        case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
        case BEQ: if (reg[rs1] == reg[rs2]) pc += immed; break;
        ...
        case JMP: pc = reg[rd]; continue;
    }
    pc++; }
```

- Except for JMP, the pc's value only depends on statically known quantities.
- Need a mapping T from source instruction addresses to target instruction addresses.

# Computed Jumps

---

- But for computed jumps, we will *not* in general be able to apply  $T$  at translation time.
- Rather, we'll have to apply  $T$  at run-time if we hope to preserve the semantics of the original code.
- So `jmp r` in the worst case would have to become "`r := T(r)`"; `jmp r`

# Ways around this...

---

- Most computed jumps are actually procedure "returns".
  - The return address is computed dynamically as a pc-relative offset.
  - So no translation is necessary.
- We can restrict T to proper "entry-points" in the code (e.g., labels.)
  - We lose the "any possible" binary.
  - But this is good -- we've cut down the possible behaviors by restricting the possible control-flow.
- Or, we can do something very, very, very clever...

# Shades of Isolation

---

- RWJ isolation: if the program attempts to read, write, or jump to a bad address, halt.
- WJ isolation: if the program attempts to write or jump to a bad address, halt.
  - Sacrifices *confidentiality*.
  - Many more reads than writes/jumps, so faster.
- Sandboxing (Wahbe et al.): if the program attempts to write or jump to a bad address, change the address to a good one.
  - Don't halt -- let the program keep running.
  - Sacrifices *fail-stop* detection.

# How to Sandbox Writes

---

- Data for a domain placed in a contiguous segment.
  - Upper bits form a *segment id*.

**0x42XXXXXX**

- Inserts code to *mask* effective address

**Mem[EA] := v**

**EA := EA & 0x42FFFFFF**

**Mem[EA] := v**

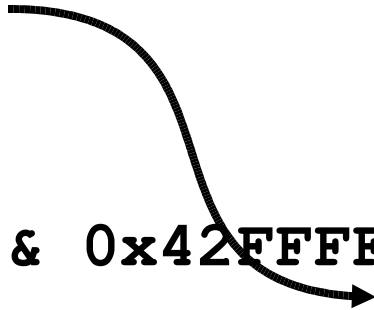
- No branch penalty
- But you don't detect a bad write.

# Sandboxing Jumps

---

- Again, mask destination address.
- But we also need to ensure that you don't jump into the middle of a masking operation.

```
0x000: z := 0xbad
0x004: r := 0x104
0x004: r := r & 0x00FFFFFF
0x008: jmp r
...
0x100: z := z & 0x42FFFFFF
0x104: Mem[z] := v
```



# The Trick

---

- Dedicate two registers,  $s$  and  $j$ .
- Invariants:
  - Register  $s$  always contains a valid data address, and  $j$  always contains a valid code address.
  - All writes use register  $s$  as the effective address, and all (computed) jumps use  $j$  as the effective address.

# Example

---

Original Code:

```
z := 0xbad
```

```
jmp r  
  0x00FFFFFFF
```

...

```
Mem[z] := v
```

Transformed Code:

```
z := 0xbad
```

```
j := r &
```

```
jmp j
```

...

```
s := z & 0x42FFFFFFF
```

```
Mem[s] := v
```

# Performance Results:

---

## Protection vs. Sandboxing:

- RWJ isolation:
  - required 5 dedicated registers, 4 instruction sequence
  - 20% overhead on 1993 RISC machines
- WJ sandboxing:
  - requires only 2 registers, 2 instruction sequence
  - 5% overhead

## Remote Procedure Call:

- 10x faster than a really good OS RPC

Sequoia DB benchmarks: 2-7% overhead for SFI compared to 18-40% overhead for OS.

# What about the x86?

---

Some serious problems:

- The most common computed jump is a RET.
  - RET pops an address from the stack & jumps to it.
  - Rewriting this to an explicit pop, mask, & jump destroys performance.
- Small # of registers:
  - Can't afford to dedicate 2 (much less 5).
- Variable length instructions:
  - We could jump into the “middle” of an instruction.
  - The “middle” may parse as a different instruction.

# A Solution (see McCamant paper)

---

- Force all computed to jumps to be to k-byte-aligned addresses (isolation:  $k=16$ , sandboxing:  $k=8$ ).
- Perform masking/etc. *within* k-byte chunk.
- No need for strong invariants on  $s$  &  $j$ : must assume any register should be masked on entry to a chunk, if it's to be used in a store or write.
- (Bunch of other low-level hacks to minimize overheads.)

# Some Thoughts

---

- Software-based isolation is a form of *code rewriting*.
  - Input: untrusted code
  - Output: code respecting invariants that imply our policy (memory isolation).
- But our policy is pretty weak.
  - For instance, we can't hand out memory/code in small pieces (need a contiguous chunk.)
  - Can we push this idea for stronger policies?
  - In particular, can we control down to the bit-level what can/can't be accessed?
  - And what about availability, confidentiality, etc.?

# More Thoughts

---

The invariants for the x86 are quite tricky.

- So tricky, Steven constructed a machine-checked proof using ACL2 that they were in fact invariants.

And the rewriter is a mess of perl code.

- I'm not sure that I trust the code is correct.
- More sophisticated policies, will demand more sophisticated rewriters.
- Even *less* likely that they will be correct.

# Coming Up:

---

Can we *statically* verify isolation of machine code?

Can we enforce stronger security policies than isolation?

How do we do this without getting a huge trusted computing base?