Efficient Software-Based Fault Isolation
The Cost of Protection-Domains

Micro-kernel performance
- L4-Linux is not as fast as monolithic Linux
- but is it acceptable given the benefits?

What were the benefits?
- Did we get fine-grain protection domains?
- L4 had the Linux server in a single domain
- What performance would L4 achieve with fine-grain protection?
Alternate Approaches

Protection using address space boundaries is inherently expensive
- even with hardware support
- execution within a domain is fast, but across domain boundaries is slow

Language-level protection might help
- binary rewriting (this paper)
- type-safe languages (SPIN)
Software Fault Isolation

Basic idea: create multiple protection domains within a single address space

- catch attempts to access memory outside a domain
- but don’t do this using hardware faults
- instead, insert code that checks accesses
  - we could do this dynamically or statically
  - how can we communicate across domain boundaries?
Enforcing Protection Domains

Protection domains are aligned and sized as a power of two bytes

- all memory addresses in the same domain share the same high-order bits
- those bits are checked for all memory references (loads, stores, jump targets)
- if the bits don’t match, the access is invalid
But That’s Too Expensive

Reads can’t hurt anything, so only check stores and jump targets?
- is that ok?

If that is still too expensive, maybe just set the high order bits without checking
- force the access to fall within the domain
- this is called sandboxing
- it may be hard to detect bugs
How Is It Implemented?

Modify GCC to generate sandboxed binaries
  - done at RTL level
  - RTL is an architecture independent assembly language
  - involves reserving up to 5 registers
  - adding sandboxing instructions

Its done at compile-time but still needs runtime checks!
Segment Matching

dedicated-reg ← target address
Move target address into dedicated register.

scratch-reg ← (dedicated-reg >> shift-reg)
Right-shift address to get segment identifier.

scratch-reg is not a dedicated register.
shift-reg is a dedicated register.

compare scratch-reg and segment-reg
segment-reg is a dedicated register.

trap if not equal
Trap if store address is outside of segment.

store instruction uses dedicated-reg

Figure 1: Assembly pseudo code for segment matching.
Sandboxing

dedicated-reg $\leftarrow$ target-reg$\&$and-mask-reg
  Use dedicated register and-mask-reg
to clear segment identifier bits.
dedicated-reg $\leftarrow$ dedicated-reg$\mid$segment-reg
  Use dedicated register segment-reg
to set segment identifier bits.
store instruction uses dedicated-reg

Figure 2: Assembly pseudo code to sandbox address in target-reg.
Costs

Segment matching
- 4x instruction overhead
- 4 dedicated registers

Sandboxing:
- 3x instruction overhead
- 5 dedicated registers

How can this possibly be a win?
Performance Implications

Impact of dedicated registers depends on architecture
- with 32 registers it’s not noticeable
- with 8 registers it is

The extra instructions make normal execution much slower!

Can we win back this performance through lower IPC costs?
Cross Domain RPC

Single thread runs in caller and callee
No address space switch
Control flow uses a jump table to cross domain boundaries
- readable, but not be writeable!
Cross Domain RPC

Call stub:
1. save trusted registers
2. set up dedicated registers
3. switch to untrusted stack
4. copy arguments to stack
5. Jump to called procedure

Return stub:
1. restore trusted registers
2. switch to trusted stack
3. copy return values back to trusted stack

Figure 4: Major components of a cross-fault-domain RPC.
## Null Procedure Call Cost

<table>
<thead>
<tr>
<th>Platform</th>
<th>Caller Save Registers</th>
<th>Save Integer Registers</th>
<th>Save Integer+Float Registers</th>
<th>C Procedure Call</th>
<th>Pipes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC-MIPS</td>
<td>1.11µs</td>
<td>1.81µs</td>
<td>2.83µs</td>
<td>0.10µs</td>
<td>204.72µs</td>
</tr>
<tr>
<td>DEC-ALPHA</td>
<td>0.75µs</td>
<td>1.35µs</td>
<td>1.80µs</td>
<td>0.06µs</td>
<td>227.88µs</td>
</tr>
</tbody>
</table>

Table 2: Cross-fault-domain crossing times.
## Execution Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th>DEC-ALPHA</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault Isolation Overhead</td>
<td>Protection Overhead</td>
<td>Reserved Register Overhead</td>
<td>Instruction Count Overhead</td>
<td>Fault Isolation Overhead (predicted)</td>
<td>Fault Isolation Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP 1.4% 33.4%</td>
<td>-0.3% 19.4%</td>
<td>0.2% 8.1%</td>
<td>35.5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bps</td>
<td>FP 5.6% 15.5%</td>
<td>-0.1% 8.9%</td>
<td>5.7% 4.7%</td>
<td>20.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cholesky</td>
<td>FP 0.6% 22.7%</td>
<td>0.5% 6.5%</td>
<td>-1.5% 0.0%</td>
<td>9.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026.compress</td>
<td>INT 3.3% 13.3%</td>
<td>0.0% 10.9%</td>
<td>4.4% -4.3%</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>056.ear</td>
<td>FP -1.2% 19.1%</td>
<td>0.2% 12.4%</td>
<td>2.2% 3.7%</td>
<td>18.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT 2.9% 34.4%</td>
<td>1.0% 2.7%</td>
<td>2.2% 2.3%</td>
<td>17.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT 12.4% 27.0%</td>
<td>-1.6% 11.8%</td>
<td>10.5% 13.3%</td>
<td>33.6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT 3.1% 18.7%</td>
<td>-9.4% 17.0%</td>
<td>8.9% NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>022.li</td>
<td>INT 5.1% 23.4%</td>
<td>0.3% 14.9%</td>
<td>11.4% 5.4%</td>
<td>16.2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>locus</td>
<td>INT 8.7% 30.4%</td>
<td>4.3% 10.3%</td>
<td>8.6% 4.3%</td>
<td>8.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mp3d</td>
<td>FP 10.7% 10.7%</td>
<td>0.0% 13.3%</td>
<td>8.7% 0.0%</td>
<td>6.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>psgrind</td>
<td>INT 10.4% 19.5%</td>
<td>1.3% 12.1%</td>
<td>9.9% 8.0%</td>
<td>36.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qcd</td>
<td>FP 0.5% 27.0%</td>
<td>2.0% 8.8%</td>
<td>1.2% -0.8%</td>
<td>12.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>072.sc</td>
<td>INT -0.8% 10.5%</td>
<td>0.4% 3.9%</td>
<td>2.1% 10.9%</td>
<td>19.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tracker</td>
<td>INT 5.6% 11.2%</td>
<td>7.0% 8.0%</td>
<td>3.8% NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>water</td>
<td>FP 0.7% 7.4%</td>
<td>0.3% 6.7%</td>
<td>1.5% 4.3%</td>
<td>12.3%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4.3% 21.8% 0.4% 10.5% 5.0%</td>
<td>4.3% 17.6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When Does It Win?

100% of time in untrusted code

50% of time in untrusted code

\[(1-r)t_c = ht_d\]

r: y-axis
tc: x-axis
h: overhead when in untrusted
td: percent of time in untrusted
Closing Thoughts

Is it really language independent?
How portable is it across CPU architectures?