Efficient Software-Based Fault Isolation

Robert Whabe, Steven Lucco, Thomas Anderson and Susan Graham

Presented by Seth Terashima
The problem

- Putting different modules in the same address space is not secure
- Using different address spaces and RPCs is not efficient

We've tried making RPCs fast, but what about making shared address spaces safe?
Place modules in *fault domains*

Code in an *untrusted* fault domain cannot:

- Write to memory outside the fault domain
- Jump to code outside the fault domain
- Divide address space into segments
- A fault domain contains two segments: one for code (instructions), and one for data
- Addresses with the same $n$-bit prefix are in the same segment. This prefix is the segment ID.
- Add assembly code before sandboxed JUMP and STORE instructions to enforce boundaries
**Problem:** What if the program jumps directly to a JUMP or STORE, bypassing safety code?
Solution: Use dedicated registers for all JUMP and STORE instructions

- These registers cannot be used for any other purpose
- They always point to addresses in the appropriate segment
store value, address-reg

- Uses the following dedicated registers:
  - dedicated-reg – Used for all store ops
  - shift-reg – Used to isolate the segment identifier (most significant $n$ bits of address)
  - scratch-reg – Stores intermediate value
  - Segment-reg – ID of current segment

```
dedicated-reg <= address-reg
scratch-reg <= (dedicated_reg >> shift-reg)
compare scratch-reg, segment-reg
trap if not equal
store value, dedicated-reg
```
Sacrifices error detection for speed

Uses the following dedicated registers:
- dedicated-reg – Used for all store ops
- mask-reg – zeroes non-segment bits
- segment-reg – Sets MSBs to segment id

\[
dedicated\-reg \leq target\-reg \& mask\-reg
\]
\[
dedicated\-reg \leq dedicated\-reg \| segment\-reg
\]
\[
store \text{ value, dedicated}\-reg
\]
Optimizations

- Some architectures have a store-with-offset
  - Put unallocated *guard zones* at either end of a segment
  - Guard zones have size equal to max offset
  - We can now sandbox the register directly, rather than performing offset arithmetic first
Problem: System resources are allocated on a per-address space basis, and we don't want to modify kernels to be fault-domain aware.
**Solution:** System calls are rewritten to go through an Arbitor in the current address space. The Arbitor associates system resources with fault domains.

![Diagram showing the process resources and the role of the Arbitor](image)
Shared memory

- Read-only shared memory automatic
- Shared read/write memory handled through lazy pointer swizzling
- LOADs from shared memory sandboxed
Inter-domain communication

Trusted domain

Call foo()

Untrusted domain

Procedure foo() {
  return value
}

Jump table

Jump table instructions use a direct address, not a register. They can't be modified by an untrusted domain, since they are in the code segment.
Inter-domain communication

- Since stubs are in a trusted domain, they can directly copy data across domains
- Stubs save/restore register values as necessary
- Possible to optimize process by ignoring registers whose values can safely be discarded
Implementation & Verification

- Two ways to implement:
  - Compile sandboxed object code, verify on load
  - Compile code normally, modify on load

- Verification:
  - Break code up into *unsafe store (jump) regions* that begin when a dedicated store (jump) register is modified
  - Segments end on control transfers, or when the dedicated register specifies a store address
  - Verify dedicated register is valid at end of region
Performance

- How much overhead incurred by sandboxing?
- How fast are cross-fault-domain RPCs?
- What is the overall performance impact on an end-user application?
# Performance Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th></th>
<th>DEC-ALPHA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault</td>
<td>Protection</td>
<td>Instruction</td>
<td>Fault</td>
</tr>
<tr>
<td></td>
<td>Isolation Overhead</td>
<td>Overhead</td>
<td>Count Overhead</td>
<td>Isolation Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP</td>
<td>1.4%</td>
<td>33.4%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>bps</td>
<td>FP</td>
<td>5.6%</td>
<td>15.5%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>cholesky</td>
<td>FP</td>
<td>0.0%</td>
<td>22.7%</td>
<td>0.5%</td>
</tr>
<tr>
<td>026.compress</td>
<td>INT</td>
<td>3.3%</td>
<td>13.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>056.ear</td>
<td>FP</td>
<td>-1.2%</td>
<td>19.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT</td>
<td>2.9%</td>
<td>34.4%</td>
<td>1.0%</td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT</td>
<td>12.4%</td>
<td>27.0%</td>
<td>-1.6%</td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT</td>
<td>3.1%</td>
<td>18.7%</td>
<td>-9.4%</td>
</tr>
<tr>
<td>022.l1</td>
<td>INT</td>
<td>5.1%</td>
<td>23.4%</td>
<td>0.3%</td>
</tr>
<tr>
<td>locus</td>
<td>INT</td>
<td>8.7%</td>
<td>30.4%</td>
<td>4.3%</td>
</tr>
<tr>
<td>mp3d</td>
<td>FP</td>
<td>10.7%</td>
<td>10.7%</td>
<td>13.3%</td>
</tr>
<tr>
<td>psgrind</td>
<td>INT</td>
<td>10.4%</td>
<td>19.5%</td>
<td>1.3%</td>
</tr>
<tr>
<td>qcd</td>
<td>FP</td>
<td>0.5%</td>
<td>27.0%</td>
<td>2.0%</td>
</tr>
<tr>
<td>072.sc</td>
<td>INT</td>
<td>5.6%</td>
<td>11.2%</td>
<td>7.0%</td>
</tr>
<tr>
<td>tracker</td>
<td>INT</td>
<td>-0.8%</td>
<td>10.5%</td>
<td>0.4%</td>
</tr>
<tr>
<td>water</td>
<td>FP</td>
<td>0.7%</td>
<td>7.4%</td>
<td>0.3%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.3%</strong></td>
<td><strong>21.8%</strong></td>
<td><strong>0.4%</strong></td>
<td><strong>10.5%</strong></td>
</tr>
</tbody>
</table>
Performance analysis

- Low or negative overhead values result from cache idiosyncrasies or reduced number of pipeline structural hazards
- Model that accounts for interlocks avoided accurately predicts overhead
- Overhead would be further reduced on I/O intensive benchmarks
How fast are RPCs?

- Roughly two orders of magnitude better than pipes.
- Different hardware makes direct comparison difficult, but roughly 5 times faster than L3 for a NULL RPC.
  - Both papers published December 1993.
When is the overhead worth it?

Horizontal axis: $c =$ Fraction of time spent making RPCs
Vertical axis: $r =$ Cross-domain RPC time / RPC time for alternative method
Shaded region: Sandboxes are better

\[ \text{Savings} = (1 - r) c - ht \]

Where $h = 0.043$ is the overhead cost incurred by sandboxing
Benchmarking Sequoia 2000

- Sequoia 2000 is a database allowing custom datatypes
- Queries involving user-defined types require dynamically loading user code
  - Unsafe! Good candidate for sandboxing

<table>
<thead>
<tr>
<th>Sequoia 2000 Query</th>
<th>Untrusted Function Manager Overhead</th>
<th>Software-Enforced Fault Isolation Overhead</th>
<th>Number Cross-Domain Calls</th>
<th>DEC-MIPS-PIPE Overhead (predicted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 6</td>
<td>1.4%</td>
<td>1.7%</td>
<td>60989</td>
<td>18.6%</td>
</tr>
<tr>
<td>Query 7</td>
<td>5.0%</td>
<td>1.8%</td>
<td>121986</td>
<td>38.6%</td>
</tr>
<tr>
<td>Query 8</td>
<td>9.0%</td>
<td>2.7%</td>
<td>121978</td>
<td>31.2%</td>
</tr>
<tr>
<td>Query 10</td>
<td>9.6%</td>
<td>5.7%</td>
<td>1427024</td>
<td>31.9%</td>
</tr>
</tbody>
</table>

Table 3: Fault isolation overhead for POSTGRES running Sequoia 2000 benchmark.
Conclusion

- **Benefits:**
  - Secure modular code
  - Very fast communication between modules (no kernel involvement)!

- **Costs:**
  - All sandboxed code runs slower
    - Additional instructions
    - Fewer registers available
  - No method for automatically writing stubs developed
Thank you!