Efficient Software-Based Fault Isolation


Presented by Nick Aschenbach
02/14/2011
Overview

Background

Techniques
  Segment Matching
  Sandboxing
  Jump Tables

Analysis of Performance
Motivation

Systems with expensive domain crossing mechanism encourage system designers to put untrusted code in the same domain.

What if extensive domain crossing was the norm instead of the exception?
Applications for Extensibility

Software Plug-ins
  POSTGRES database manager
  Quark Xpress

Operating Systems
  Micro-kernels
  SPIN

Microsoft OLE
foo() return

Single Address Space

Multiple Address Spaces
Multiple Address Space Issues

Transferring across address boundaries requires:
- Trap into kernel
- Copy arguments from caller to callee
- Saving and restoring registers
- Switching hardware address spaces
- Possible TLB flush
- Trap back to user level

These operations are duplicated on return!

Even Optimized RPC calls are relatively slow

Doesn’t work well for tightly coupled modules
Single Address Space Issues

Direct function calls are fast, however…

Faults in extension code
  Data corruption
  System reliability

Suggests need for segmentation fault isolation at the software level
Software Fault Isolation

Goals
- Efficient software encapsulation
- Efficient communication mechanisms

Authors use three approaches
- Segment matching
- Sandboxing
- Jump tables
Fault Domain Address Space

Division of application virtual address space into segments

Each segment shares a unique pattern of upper bits called a segment identifier

Fault domain comprised of code and data segment (constants, heap and stack)
Target Access Privilege Types

Jump targets
  Only within its code segment

Write targets
  Only within its data segment
Segment Matching

Target address verification
  Static
  Dynamic

Check that target has correct segment ID

If that fails trap to a system error routine
Segment Matching

Four dedicated registers required

One to hold addresses in code segment
One to hold addresses in data segment
One to hold shift amount (for bit shifting)
One to hold segment identifier

How does tying up extra registers affect the module execution time?
Segment Matching Example

Code executing with segment identifier 0101 (in the segment register), but wants to write to a target with segment identifier of 0110.

The segment identifier is the four most significant bits (MSB)

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

Dedicated reg = 0110 1011
Shift reg = 0000 0100
Segment reg = 0000 0101

Scratch-reg = (dedicated >> shift)
Scratch-reg = 0000 0110

Note 0000 0110 != 0000 0101

So we trap to a routine outside of the module’s domain. Also, we can pinpoint the offending instruction (for debugging)
Sandboxing

Optimization similar to segment matching

We lose the ability to pinpoint offending instruction

Works by masking and setting MSB
Sandboxing Example

Five dedicated registers required

One to hold segment mask
Two registers hold code and data segment identifiers
Two are used to hold sandboxed code and data addresses

dedicated-reg ← target-reg\&and-mask-reg
   Use dedicated register and-mask-reg
e to clear segment identifier bits.
dedicated-reg ← dedicated-reg\|segment-reg
   Use dedicated register segment-reg
to set segment identifier bits.
store instruction uses dedicated-reg
Sandboxing Example

Use simple bitwise AND and OR operators:

\[
\begin{align*}
\text{target} &= 0110 \ 1011 \\
\text{mask} &= 0000 \ 1111 \\
\text{segment ID} &= 1001 \ 0000
\end{align*}
\]

\[
\begin{align*}
\text{target} \land \text{mask} &= 0000 \ 1011 \\
\text{target} \lor \text{segment ID} &= 1001 \ 1011
\end{align*}
\]

What happens when we JMP back into our own code segment at an unintentional position?
Guard Zone Optimization

RISC instruction
register-plus-offset
mode:

store *value, offset (reg)*

Sandboxing would require three instructions, but saves one instruction by sandboxing *reg* instead of *reg+offset*
Requires guard zones at top and bottom of segment
Communication Outside Fault Domain

[Diagram showing a call from a Trusted Caller Domain to an Untrusted Callee Domain]
Communication Outside Fault Domain

![Diagram showing communication between trusted and untrusted domains, with a call from trusted to untrusted, and return blocked.]

- **Trusted Caller Domain**
- **Untrusted Callee Domain**

Call: Trusted to Untrusted

Return blocked: Untrusted to Trusted
Communication Outside Fault Domain

![Diagram showing communication between trusted and untrusted domains](image)

- Trusted Caller Domain
- Call Add
- Call Stub
- Return Stub
- Untrusted Callee Domain
- Add:
- Return
- Jump Table

Read-only Code segment
Results

Sandboxing overheads

Cross fault domain crossing times (RPC)

POSTGRES benchmark

Percentage of Execution Time Spent Crossing

Percentage of time spent in code crossing vs. number of fault domain crossings per ms
## Sandboxing Overheads

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

<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>
## Overhead w/ POSTGRES Queries

<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>
Execution Time Spent Crossing

100% Application code sandboxed

50% Application code sandboxed
Sandboxing Scalability

![Graph showing the percentage of execution time spent crossing boundaries for different systems.](image-url)
Conclusions

Sandboxing incurs 4% execution overhead

Cross-Fault Domain Communication > 10x faster than RPC mechanisms

Only untrusted modules need to be sandboxed

Techniques prevent store and jumping, but not reading (a potential security issue)

Possible to implement this at the OS level