SOFTWARE-BASED FAULT ISOLATION

MICHAEL ROITZSCH
OVERVIEW

- Motivation
- The Idea
- Concepts
- Evaluation
- Ideas for Enhancements
- XFI: Enhancing SFI
- XFI: Evaluation
- NaCl Excursion
- Summary
This first part is based on the paper

Efficient Software-Based Fault Isolation

by

Robert Wahbe, Steven Lucco,
Thomas E. Anderson and Susan L. Graham

and appeared at the

Symposium on Operating System Principles in 1993
• Hardware-based isolation
• Applications with extensible interfaces (Plugins, Codecs, Query Code)
• HFI Slow (ten thousands of Cycles, @40 MHz DEC Station)
  → often not used (Postgres)
• RPC in one address space = dozens of cycles @40MHz DEC!
• But address spaces provide no fault isolation :(
MOTIVATION

- DECstation 5000/240 Mips R3400 @40MHz
- Integer performance comparable to 486DX4/100
- Source:
- Alfred Arnold
THE IDEA

- Do not use hardware isolation (paging) but map plugins and co into the address space, fast rpc (dozens of cycles)
- Price: stability
- Example: Quark Express Desktop Publishing
- Plugins crash main system (overwrite/corrupt state of main program), crashiness attributed to Quark → bad for reputation
THE IDEA (IMPROVED)

- Provide guarantees of HFI without the costs
- Use a custom compiler that enables the sandboxing of the software
- Verifier checks if the binary is correctly sandboxed
- Approach is especially beneficial for systems with high amounts of communication
CONCEPTS

- Segment Matching
- Address Sandboxing
- Resource Access
- Data Sharing
- Verification
- RPC
How to ensure stores only to “own” memory?

```
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
```
• Is there a simpler way? Do we need those guarantees?

\[
dedicated-reg \leftarrow target-reg \& and-mask-reg
\]
Use dedicated register \textit{and-mask-reg} to clear segment identifier bits

\[
dedicated-reg \leftarrow dedicated-reg \mid segment-reg
\]
Use dedicated register \textit{segment-reg} to set segment identifier bits

\textit{store instruction uses dedicated reg}
OPTIMIZATION

• Do we really need to verify all accesses? This is expensive!
• We lose some address arithmetic features
  \[ \text{reg} \leq \text{mem[EAX\pm\text{off}]} \]
• Solution: Guardzones!
• Shared resources in one address space
• Arbitration in Kernel
• ... or by restricting resource access to trusted arbitration code
- Reading is trivial
- R/W through shared (aliased) pages

```c
struct node {
    ...
    struct node *next;
};

node *a = 0x4000 0000
a->next = 0x4000 0010
a->next->next = null_ptr
```
How do segments communicate?

- One Stub-pair per caller/callee pair
- Stubs are trusted
- Signals for Fault Isolation
• Simplified by dedicated jump register
• Scan code segment (Fixed-width Opcodes :))
• Verify statical calls / accesses / jumps
• Check others for dedicated register
### EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th>DEC-ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fault Isolation Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP</td>
<td>1.4%</td>
</tr>
<tr>
<td>bps</td>
<td>FP</td>
<td>5.6%</td>
</tr>
<tr>
<td>cholesky</td>
<td>FP</td>
<td>0.0%</td>
</tr>
<tr>
<td>026.compress</td>
<td>INT</td>
<td>3.3%</td>
</tr>
<tr>
<td>056.ea</td>
<td>FP</td>
<td>-1.2%</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT</td>
<td>2.9%</td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT</td>
<td>12.4%</td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT</td>
<td>3.1%</td>
</tr>
<tr>
<td>022.li</td>
<td>INT</td>
<td>5.1%</td>
</tr>
<tr>
<td>locus</td>
<td>INT</td>
<td>8.7%</td>
</tr>
<tr>
<td>mp3d</td>
<td>FP</td>
<td>10.7%</td>
</tr>
<tr>
<td>psgrind</td>
<td>INT</td>
<td>10.4%</td>
</tr>
<tr>
<td>qcd</td>
<td>FP</td>
<td>0.5%</td>
</tr>
<tr>
<td>072.sc</td>
<td>INT</td>
<td>5.6%</td>
</tr>
<tr>
<td>tracker</td>
<td>INT</td>
<td>-0.8%</td>
</tr>
<tr>
<td>water</td>
<td>FP</td>
<td>0.7%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td><strong>4.3%</strong></td>
</tr>
</tbody>
</table>
## EVALUATION

<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>1.11 (\mu s)</td>
<td>1.81 (\mu s)</td>
<td>2.83 (\mu s)</td>
<td>0.10 (\mu s)</td>
<td>204.72 (\mu s)</td>
</tr>
<tr>
<td>DEC-ALPHA</td>
<td>0.75 (\mu s)</td>
<td>1.35 (\mu s)</td>
<td>1.80 (\mu s)</td>
<td>0.06 (\mu s)</td>
<td>227.88 (\mu s)</td>
</tr>
</tbody>
</table>

Table 2: Cross-fault-domain crossing times.

<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.
IDEAS FOR ENHANCEMENTS

- Compiler support is a problem
- Can we do without the compiler? Binary rewriting!
- Can this be applied to CISC architectures (verification is more difficult!)
- is the limitation to one specific segment feasible?
- Can we allow more fine-granular access?
This second part is based on the paper
XFI: Software Guards for System Address Spaces
by
Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu and George C. Necula
and appeared at the
Symposium on Operating System Design and Implementation in 2006
Challanges Addressed by XFI

- Make SFI work in systems software (drivers)
- Remove the single segment limit
- Prevent attacks through ROP techniques
- Protect individual system instructions
GENERAL CONCEPT

The diagram illustrates the general concept of software-based fault isolation. It breaks down the host-system space into different components:

- **Host-system executable**: Contains code and data.
- **XFI module**: Represents the execution environment with code, data, read-only data, and read/write data.
- **Host-system stacks**: Indicates areas for stack management.
- **Host-system heap**: Shows the memory area for dynamic allocation.

Support routines and entry points are also depicted, indicating how different components interact within the host-system space.
ENFORCED PROPERTIES

- P1: Memory-access constraints
- P2: Interface Restrictions
- P3: Scoped Stack Integrity
- P4: Simplified Instruction Semantics
- P5: System-environment integrity
- P6: Control-flow integrity
- P7: Program-data integrity
EAX := 0x12345677  # Identifier - 1
EAX := EAX + 1
if Mem[EBX - 4] ≠ EAX, goto CFIERR

call EBX

...  

0x12345678  # Target identifier  
L: push EBP  # Callee code
MEMORY RANGE GUARDS

# mrguard(EAX, L, H) ::=  
if EAX < A + L, goto S  
if B - H < EAX, goto S  
M: Mem[EAX] := 42       # Two writes  
Mem[EAX - L] := 7        # both allowed  
...  
S: push EAX             # Arguments for  
push L, H               # slower guard  
call SlowpathGuard      
jump M                  # Allow writes
ACCESS CONTROL

- Slowpath guards enable byte granularity permissions
- System level
- For individual instructions the “unsafe region” can be specified
- This enables the protection of outside code
REQUIRED RUNTIME SUPPORT

- Slowpath permission tables
- Allocation-Stack manager
- Software Call Gates (changed stack model!)
- Exception Handling Support (Windows SEH)
• Guards may be implemented in Hardware
• Can be simulated as NOP guards
• Required Instructions:
  - `cfilabel` instruction
  - variants of jump/call/return instructions
  - `mrguard` instruction
• Implemented in Alpha architecture simulator
• Do not use dedicated registers (esp. on x86)
• Verification is more difficult
• Needs verification state
• must check all paths
<table>
<thead>
<tr>
<th>Code</th>
<th>Verification State</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrguard(EAX, 0, 8)</td>
<td></td>
</tr>
<tr>
<td>EDX := Mem[EAX]</td>
<td></td>
</tr>
<tr>
<td>Mem[EAX+4] := EDX</td>
<td></td>
</tr>
<tr>
<td>EAX := Mem[ASP-4]</td>
<td></td>
</tr>
<tr>
<td><strong>POP</strong> ASP</td>
<td></td>
</tr>
<tr>
<td><strong>RET</strong></td>
<td># SSP := SSP+4; jump Mem[SSP-4]</td>
</tr>
</tbody>
</table>
Code | Verification State
--- | ---
{origSSP = SSP+8, valid[SSP,SSP+8] } |
{retaddr=Mem[SSP+4]} |
{origASP=Mem[SSP],valid[ASP-32,ASP]}

```
mrguard(EAX,0,8)

EDX := Mem[EAX]
Mem[EAX+4] := EDX
EAX := Mem[ASP-4]
POP ASP

RET # SSP := SSP+4; jump Mem[SSP-4]
```
### Code Verification State

<table>
<thead>
<tr>
<th>Code</th>
<th>Verification State</th>
</tr>
</thead>
<tbody>
<tr>
<td>{origSSP = SSP+8, valid[SSP,SSP+8)}</td>
<td></td>
</tr>
<tr>
<td>{retaddr=Mem[SSP+4]}</td>
<td></td>
</tr>
<tr>
<td>{origASP=Mem[SSP],valid[ASP-32,ASP)}</td>
<td></td>
</tr>
<tr>
<td>mrguard(EAX,0,8)</td>
<td></td>
</tr>
<tr>
<td>{valid[EAX-0,EAX+8)}</td>
<td></td>
</tr>
<tr>
<td>EDX := Mem[EAX]</td>
<td></td>
</tr>
<tr>
<td>Mem[EAX+4] := EDX</td>
<td></td>
</tr>
<tr>
<td>EAX := Mem[ASP-4]</td>
<td></td>
</tr>
<tr>
<td><strong>POP</strong> ASP</td>
<td></td>
</tr>
<tr>
<td><strong>RET</strong></td>
<td># SSP := SSP+4; jump Mem[SSP-4]</td>
</tr>
</tbody>
</table>
### Code Verification State

<table>
<thead>
<tr>
<th>Code</th>
<th>Verification State</th>
</tr>
</thead>
</table>
| {origSSP = SSP+8, valid[SSP,SSP+8]}  
{retaddr=Mem[SSP+4]}  
{origASP=Mem[SSP],valid[ASP-32,ASP]} | mrguard(EAX,0,8) |
| {valid[EAX-0,EAX+8]} | EDX := Mem[EAX]  
Mem[EAX+4] := EDX  
EAX := Mem[ASP-4] |
| POP ASP  
# ASP := Mem[SSP]; SSP := SSP+4 | {origASP=ASP, valid[SSP,SSP+4]}  
{origSSP=SSP+4, retaddr=Mem[SSP]} |
| RET  
# SSP := SSP+4; jump Mem[SSP-4] |
**Table 1:** Code-size increase and slowdown of SFI benchmarks, with XFI. The % rows show slowdown.

<table>
<thead>
<tr>
<th></th>
<th>NOP</th>
<th>fastpath</th>
<th>slowpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotlist Δ sz %</td>
<td>2.1x (2.6x)</td>
<td>2.5x (4.1x)</td>
<td>3.9x (8.3x)</td>
</tr>
<tr>
<td>ld Δ sz %</td>
<td>1.2x (1.3x)</td>
<td>1.5x (1.8x)</td>
<td>1.7x (2.3x)</td>
</tr>
<tr>
<td>MD5 Δ sz %</td>
<td>1.1x (1.1x)</td>
<td>1.2x (1.3x)</td>
<td>1.3x (1.5x)</td>
</tr>
</tbody>
</table>

**Table 2:** Code-size increase and slowdown for different kernel buffer sizes for a WDF benchmark, with XFI. The unprotected driver is 11KB of x86 machine code; its transactions per second (shown in thousands) form the baseline for the slowdown percentages.

<table>
<thead>
<tr>
<th>Kt/s</th>
<th>NOP</th>
<th>fastpath</th>
<th>slowpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>193</td>
<td>5.0% (4.8%)</td>
<td>6.8% (6.1%)</td>
</tr>
<tr>
<td>512</td>
<td>151</td>
<td>4.7% (3.9%)</td>
<td>5.3% (4.7%)</td>
</tr>
<tr>
<td>4K</td>
<td>71</td>
<td>1.7% (1.7%)</td>
<td>2.7% (2.9%)</td>
</tr>
<tr>
<td>64K</td>
<td>5</td>
<td>1.2% (1.9%)</td>
<td>1.4% (0.4%)</td>
</tr>
</tbody>
</table>

**Table 4:** Slowdown of Mediabench kernels, with XFI.

<table>
<thead>
<tr>
<th></th>
<th>NOP</th>
<th>fastpath</th>
<th>slowpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_encode</td>
<td>0%  (4%)</td>
<td>2%  (49%)</td>
<td>13% (149%)</td>
</tr>
<tr>
<td>adpcm_decode</td>
<td>−3% (2%)</td>
<td>3%  (12%)</td>
<td>36% (112%)</td>
</tr>
<tr>
<td>gsm_decode</td>
<td>3%  (1%)</td>
<td>79% (97%)</td>
<td>125% (230%)</td>
</tr>
<tr>
<td>epic_decode</td>
<td>3%  (9%)</td>
<td>7%  (19%)</td>
<td>119% (220%)</td>
</tr>
</tbody>
</table>
CONCLUSION

- SFI is also feasible on x86
- Verification becomes more complex
- Mechanisms also enable other restrictions
- SFI is feasible for kernel code
- Separate allocation- and scoped stacks reduce exploitability
This third part is based on the paper

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

by

Bennet Yee, David Sehr, Gregory Dardyk,
J. Bradley Chen, Robert Muth, Tavis Ormandy,
Shiki Okasaka, Neha Narula and Nicholas Fullager

and appeared at the

Symposium on Security and Privacy in 2009
APPLICATIONS

- Implemented as “Pepper” in Chrome
- Used for
  - Flash
  - PDF Viewer
  - Quake
  - Doom
  - Lara Croft and the Guardian of Light
- Uses LLVM IR bytecode for platform independence
• x86-32 segmentation for memory isolation
• SFI for arm / amd64
• CFI through restriction on indirect jumps
  - must be to 32 byte aligned basic blocks
  - no opcodes are allowed to cross this boundary
• Requires Recompilation
PERFORMANCE RESULTS

![Performance Results Graph]

- align32
- nacl32

The graph compares the performance slowdown of different benchmarks when using align32 and nacl32. The x-axis represents various benchmark programs, and the y-axis shows the slowdown compared to a static baseline. The bars indicate the percentage change in performance for each benchmark under the two conditions.
<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>aligned</th>
<th>NaCl</th>
<th>increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>657</td>
<td>759</td>
<td>766</td>
<td>16.7%</td>
</tr>
<tr>
<td>art</td>
<td>469</td>
<td>485</td>
<td>485</td>
<td>3.3%</td>
</tr>
<tr>
<td>bzip2</td>
<td>492</td>
<td>525</td>
<td>526</td>
<td>7.0%</td>
</tr>
<tr>
<td>crafty</td>
<td>756</td>
<td>885</td>
<td>885</td>
<td>17.5%</td>
</tr>
<tr>
<td>eon</td>
<td>1820</td>
<td>2016</td>
<td>2017</td>
<td>10.8%</td>
</tr>
<tr>
<td>equake</td>
<td>465</td>
<td>475</td>
<td>475</td>
<td>2.3%</td>
</tr>
<tr>
<td>gap</td>
<td>1298</td>
<td>1836</td>
<td>1882</td>
<td>45.1%</td>
</tr>
<tr>
<td>gcc</td>
<td>2316</td>
<td>3644</td>
<td>3646</td>
<td>57.5%</td>
</tr>
<tr>
<td>gzip</td>
<td>492</td>
<td>537</td>
<td>537</td>
<td>9.2%</td>
</tr>
<tr>
<td>mcf</td>
<td>439</td>
<td>452</td>
<td>451</td>
<td>2.8%</td>
</tr>
<tr>
<td>mesa</td>
<td>1337</td>
<td>1758</td>
<td>1769</td>
<td>32.3%</td>
</tr>
<tr>
<td>parser</td>
<td>641</td>
<td>804</td>
<td>802</td>
<td>25.2%</td>
</tr>
<tr>
<td>perlbmk</td>
<td>1167</td>
<td>1752</td>
<td>1753</td>
<td>50.2%</td>
</tr>
<tr>
<td>twolf</td>
<td>773</td>
<td>937</td>
<td>936</td>
<td>21.2%</td>
</tr>
<tr>
<td>vortex</td>
<td>1019</td>
<td>1364</td>
<td>1351</td>
<td>32.6%</td>
</tr>
<tr>
<td>vpr</td>
<td>668</td>
<td>780</td>
<td>780</td>
<td>16.8%</td>
</tr>
</tbody>
</table>

Table 5: Code size for SPEC2000, in kilobytes.
GAMING PERFORMANCE

<table>
<thead>
<tr>
<th>Run #</th>
<th>Native Client</th>
<th>Linux Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>143.2</td>
<td>142.9</td>
</tr>
<tr>
<td>2</td>
<td>143.6</td>
<td>143.4</td>
</tr>
<tr>
<td>3</td>
<td>144.2</td>
<td>143.5</td>
</tr>
<tr>
<td>Average</td>
<td>143.7</td>
<td>143.3</td>
</tr>
</tbody>
</table>

Table 8: Quake performance comparison. Numbers are in frames per second.

The Catch: Software rendering ;)

TU Dresden
Software-Based Fault Isolation
41
SUMMARY

- Software-based Fault Isolation
- ... can be done on CISC and RISC architectures
- ... enables finer grained access controls and enforcement
- ... at a (quite high) performance price
- ... but is mitigated for applications with high communication frequency by reduced context switching overhead
- ... can be used to enforce other policies than only memory protection
- ... is actively used by Google Chrome