Efficient Software-Based Fault Isolation

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

Software Extensibility

**Operating Systems**
- Kernel modules
- Device drivers
- Unix vNodes

**Application Software**
- PostreSQL
- OLE
- Quark Xpress, Office

**But:**
Flaws in extension modules could cause flaws in the entire system
- Crashes
- Data corruption
Hardware Isolation

is slow

• Traps, address space switches, TLB flushes...
• Performance doesn’t necessarily improve with integer performance

Software Isolation

• Load each untrusted module into its own fault domain
• Provide write protection so that untrusted code can’t corrupt data
• Limit execution so that untrusted code can’t hijack operating system resources or crash containing program
Implementation

• Fault domains are segments

  Segment ID  
  = Target Address
  Upper Address Bits

• Untrusted code gets code and data segments

• Write protection
  – Segment matching
  – Address sandboxing

Graphic stolen from Tony Bock

Segment Matching

store using target-address

Becomes:

dedicated-reg <= target-address
scratch-reg <= (dedicated-reg >> shift-reg)
compare scratch-reg segment-reg
trap if not equal
store using dedicated-reg
Address Sandboxing

store using target-address

Becomes:

\[ \text{dedicated-reg} \leftarrow \text{target-address} \& \text{mask-reg} \]
\[ \text{dedicated-reg} \leftarrow \text{dedicated-reg} \mid \text{segment-reg} \]
\[ \text{store using dedicated-reg} \]

Process Resources

• Need to protect file handles, other process resources.
  - Make operating system aware of fault domains
  - Require fault domains to access process resources through RPC
## Implementation

### Segment Matching
- Four dedicated registers
- Five extra instructions
- Trap indicates exact instruction that caused failure

### Address Sandboxing
- Five dedicated registers
- Two extra instructions
- No indication of failure

### Optimization

**Compiler customization** or **object patching**

## Data Sharing

- All data is readable from fault domains

- Pages mapped into multiple fault domains allow cross-fault-domain communication
Cross-Domain RPC

• Generate stubs for interfaces in trusted code.

• Stubs responsible for:
  – Copying arguments
  – Preserving machine state
  – Trapping failures and time-outs

• But no traps or address space switching

Performance

• Encapsulation overhead
• Cross-fault-domain RPC cost
• Effect on user programs
Performance

<table>
<thead>
<tr>
<th>Sequence 2000 Query</th>
<th>Untrusted Function Manager Overhead</th>
<th>Software-Enforced Fault Isolation Overhead</th>
<th>Number of 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>

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

• S&P 09’

• Google Inc
  – Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar

• Best paper award
Everyone uses the web browser

• Browser is the most important tool to get the information in modern society.
  – Restricted environment for safety purpose.
    • Interpreter-based sandbox
    • Slow
  – Native plug-ins for extra performance or functionality requirements.
    • Fast, versatile
    • Trust-based protection but not safe

Native code == unsafe?

• “No fundamental reason why native code should be unsafe”
  – Traditional difficulties:
    • The problem of deciding the outcome of arbitrary native code with executing it is undecidable.
    • Many unexpected side effects during code execution.
      – Exception, interrupt, racing condition, I/O.
  • But a safe and efficient isolated environment can be created for restricted native code.
Threat model

- Achieve comparable safety to accepted systems such as JavaScript.
  - Input: arbitrary code and data
    - support multi-threading, inter-module communication
  - Restrictions (Obligations):
    - No code page writing: No self-modification code, No JIT
    - No direct system call: No I/O
    - No hardware exception/interrupt: failsafe
    - No ambiguous indirect control flow transfer
    - Isolated direct memory access

Obey me or die

Binary code satisfies the obligations

Binary code does not satisfy the obligations

Native Client (NaCl)

Check by static analysis
Microkernel-based architecture

Untrusted native code runs in its own private address space created by X86 segment registers (%cs, %ds, %gs, %fs, %ss).

NaCl module and the browser runs in the same process.

All dangerous interfaces are forbidden or monitored by the sandbox (including the instructions modifying the segment registers).

Obligations for control flow transfer

C1 Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2 The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3 All indirect control transfers use a nacljmp pseudo-instruction (defined below).
C4 The binary is padded up to the nearest page with at least one hlt instruction (0x4).
C5 The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6 All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7 All direct control transfers target valid instructions.
Security properties under obligations

- A static code analysis will ensure:
  - **Data integrity**
    - All memory addresses are within the sandbox
    - Otherwise, a segmentation fault given (%cs, %ds,... are set)
  - **Reliable disassembly**
    - All possible jump targets are known (mandatory 32byte alignment for all jump instructions)
  - **No unsafe instructions**
    - Disassembler is reliable
  - **Control flow integrity**
    - Same reason for reliable disassembly

---

Load a NaCl module

**Memory address:**

1. Verify the module code according to the obligations.
2. Load control code block into memory (including system call trampolines, thread context data).
3. Load the module code and data into memory.
4. Set the segment registers to establish a private memory space (64KB afterwards, 64KB is the zero offset).
5. Transfer the control to the module code.
Applications, tools, and availability

- **Applications**
  - Allow developer to choose any language in the browser (not just JavaScript).
  - Allow simple, computationally intensive extensions for web applications
  - Binary-level sandbox without a trusted compiler
- **Tools:** GCC tool chain
  - on Ubuntu Linux, MacOS, Windows XP
- **Availability:** open source, part of Chrome

Easier than you imagine

- **Ported programs mentioned:**
  - SPEC CPU 2000 benchmarks
  - Some graphics computation demo
  - H.264 video decoder
  - Physics simulation system
  - FPS game *(Quake)*

All Rights Reserved. Copyright 2010 by CSRG-Yin Lab.
Insignificant performance overhead

Max space overhead is 57.5% code size increment for gcc in SPEC CPU 2000.

Mandatory alignment for jump targets impacts the instruction cache and increases the code size (more significant if compared to dynamic linked executables).