Software-based Fault Isolation

Need for extensibility

- Applications can incorporate independently developed modules
  - Operating System
  - Add new file system
  - Database Management System
  - User-defined data type
  - Browser
  - Multimedia editor

Problem with extensions

- Security and Reliability
- Extensions may be
  - Malicious
  - Vulnerable
  - Faulty
- Solution:
  - Isolate from others

Isolation option 1

- Hardware-based isolation
  - Place each module in its own address space
  - Communicate via RPC

Isolation option 1

- Hardware-based isolation
  - Place each module in its own address space
  - Communicate via RPC
    - Switch to kernel mode
    - Copy arguments
    - Save/Restore registers
    - Switch address spaces
    - Return to user mode
**Isolation option 2**

- Software-based isolation
  - All modules in same virtual address
  - Protect them from each other
  - Provide efficient communication

---

**Efficient Software-based Fault Isolation**

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

**Goal**

- Protect the rest of an application from a buggy/malicious module on RISC architecture
- Separate untrusted code
  - Define a fault domain
  - Prevent the module from jumping or writing outside of it
  - While letting efficient communications

**Fault Domain**

- Load untrusted extension into its own fault domain
  - Code Segment
  - Data Segment
- Security Policy:
  - No code is executed outside of fault domain
  - No data changed outside of fault domain
  - Some protect load, too

**Unsafe Instructions**

- Jump or store instructions
  - Change Control flow
  - Change data
- Addressing issue
  - jmp 0x10001e0

jmp 0x10001e0
Add 0x4, %ebx
0x10001e0
Unsafe Instructions

- Jump or store instructions
  - Change Control flow
  - Change data
- Addressing issue
  - jmp 0x10001e0
  - mov %eax,0x11020028

Unsafe Instructions

- Jump or store instructions
  - Change Control flow
  - Change data
- Addressing issue
  - jmp 0x10001e0
  - jmp *%ecx
  - mov %eax,0x11020028
  - mov $0x1b80,(%ecx)

Segment ID

- Within a segment
  - Addresses share unique pattern of upper bits

Segment Matching

- Insert checking code before unsafe instruction
- Use dedicated registers

\[
\text{dedicated-reg} \leftarrow \text{target-address} \\
\text{scratch-reg} \leftarrow (\text{dedicated-reg} \gg \text{shift-reg}) \\
\text{if scratch-reg == segment-reg:} \\
\text{jmp/mov using dedicated-reg}
\]

Segment Matching

- Needs 4 dedicated registers
- Checking code must be atomic
- Exact location of fault can be detected
- Runtime overhead
  - 4 extra instructions

Address Sandboxing

- Ensure, do not check!
- Before each unsafe instruction
  - Set upper bit of target address to correct segment ID

\[
\text{dedicated-reg} \leftarrow \text{target-address} \& \text{and-mask} \\
\text{dedicated-reg} \leftarrow \text{dedicated-reg} \mid \text{segment-reg} \\
\text{jmp/mov using dedicated-reg}
\]
**Address Sandboxing**
- Prevents faults
- Needs 5 dedicated registers
- 2 extra instructions
  - less overhead compared to segment matching

---

**Optimizations**
- register-plus-offset mode
  - store value, offset(reg)
  - offset is in the range of -64K to +64K
  - mov %esi,0x8(%edx)

---

**Optimizations**
- Stack pointer
  - Just sandbox it when it is set (beginning of a function)
  - Ignore sandboxing for small changes (push, pop)
  - Works because of guard zones

---

**Cross Fault Domain Communication**
- Host to an untrusted module
- Untrusted module to host
- Call/Return Stub
  - For each pair of fault domains

---

**Cross Fault Domain Call**
- Trusted call/return stub
  - copy parameters
  - switch execution stack
  - maintain values of CPU registers
  - no traps or address space switching
    - faster
  - returns via jump table
    - jump targets are immediates
    - a legal address in target fault domain

---

**Implementation**
- Change the compiler
  - emit encapsulation code into distrusted code
- At the load time
  - check the integrity of encapsulation code
  - Verifier
Verifier

- Responsible for checking encapsulation instructions just before execution start
- Challenge:
  - indirect jump
- Hint:
  - every store/jump uses dedicated registers
  - Look for changes in dedicated registers
    - any change means beginning of a check region
    - verify the integrity of check region

What about CISC architectures?!
x86

Evaluating SFI for a CISC Architecture (PittSField)

Stephen McCamant, Greg Morrisett
USENIX 2005

CISC Architectures

- RISC Architecture
  - Fixed length instructions
  - More CPU registers
- Intel IA-32 (aka x86-32)
  - Variable length instructions
  - Less CPU registers
- Classical SFI is not applicable here

CISC Architectures

- Processor can jump to any byte
- Hard to make hidden instructions safe
- Solution: Instruction Alignment

```
push %esi
mov $0x56,%dh
adb $0xff,%al
inc %eax
or %al,%dl
```

Alignment

- Divide memory into 16-byte chunks
- No instruction is allowed to cross chunk boundary
- Target of jumps placed at the beginning of chunks
- Call instructions placed at the end of chunk
### Alignment
- Use NOP for padding
- No separation of an unsafe instruction

<table>
<thead>
<tr>
<th>0 1 2 3 4 5</th>
<th>6 7 8 9</th>
<th>a b c d e f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-byte nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f10</th>
<th>f20</th>
<th>f30</th>
<th>f40</th>
<th>f50</th>
<th>f60</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-byte nop</td>
<td></td>
<td>mov $0x0000</td>
<td>lea (%esi),</td>
<td>and $0x200000</td>
<td>shl %cl,exx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%ebx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Jumps
- Chunks are atomic
- Jump destinations are checked to be 16-byte aligned

<table>
<thead>
<tr>
<th>0 1 2 3 4 5</th>
<th>6 7 8 9</th>
<th>a b c d e f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-byte nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f10</th>
<th>f20</th>
<th>f30</th>
<th>f40</th>
<th>f50</th>
<th>f60</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-byte nop</td>
<td></td>
<td>mov $0x400</td>
<td>lea (%esi),</td>
<td>and $0x200000</td>
<td>shl %cl,exx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%ebx</td>
<td>%ebx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optimization: AND-only Sandboxing
- Choose code and data region addresses carefully
- Their ID just has one bit set
- Reduces sandboxing sequence to just one instruction

<table>
<thead>
<tr>
<th>Memory</th>
<th>SPI Code</th>
<th>SPI Data</th>
<th>Treated Code and Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x100000</td>
<td>0x000000</td>
<td></td>
</tr>
<tr>
<td>1x0</td>
<td>0x200000</td>
<td>0x200000</td>
<td></td>
</tr>
<tr>
<td>2x0</td>
<td>0x300000</td>
<td>0x300000</td>
<td></td>
</tr>
<tr>
<td>3x0</td>
<td>0x400000</td>
<td>0x400000</td>
<td></td>
</tr>
</tbody>
</table>

### Example

### Verification
- Statically check
  - No jump to outside of code region
  - No store to outside of data region
- Before each unsafe jump or store there should be a sandboxing AND
- The sandboxing AND should not be the last instruction in a chunk

### Performance overhead
- Implemented prototype
  - named PittSField
- Average module overhead: 21%
- But the overall execution can be improved because of faster communications
  - no trap, RPC, etc
Native-client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, et al.
IEEE S&P, 2009

Google Native Client

- Browser Plugin (Google Chrome)
  - Allows execution of untrusted C/C++ code in browser
- Browser?! Native Code?!
  - Yes! Browsers are new platform for applications
- Gives Browser plugins performance of native code
- Ships by default since Chrome 14

Sandboxing

- Inner Sandbox
  - Code sandboxing
    - Alignment and address sandboxing
    - Check branch target addresses
  - Data Sandboxing
    - Segmented addressing mode supported by x86_32
- Outer Sandbox
  - Controls system calls issued by native code
  - Whitelist

NaCl Architecture

- Inner Sandbox
  - On x86_32
    - Sandbox via segmented memory
    - Used to separate trusted from untrusted code/data
    - Modified when switching between trusted/untrusted
    - %cs code
    - %ds data
    - %gs thread local storage
    - %ss %es %fs all set to %ds
  - On x86_64
    - mov/branch alignment, guard pages
    - r15 keeps base address of an aligned 4GB range

Native Client Toolchain

- Modified GCC and GAS
  - To emit sandboxing instructions
- Final executable has ELF file structure (called NEXE)
  - Can be disassembled using standard tools
    - objdump -d
      
      naclcall %ebx and $0xffffffe0,%ebx
      call *%ebx

      nacljmp %ecx and $0xffffffe0,%ecx
      jmp *%ecx

      nacljump %ecx and $0xffffffe0,%ecx
      jmp *%ecx
Alignment

- Divide memory into 32-byte Bundles
- Target of jumps placed at the beginning of bundles
- No instruction is allowed to cross bundle boundary

Service Runtime

- No existing or new memory allocations may be marked as executable at runtime
  - This guarantees only validated code pages have executable permissions
- NaCl syscall
  - ~30 syscalls allowed
  - Basic operations such as open, close, read, write, ioctl, mmap, munmap, stat, _exit and a few others

Validator

- Disassembles all NEXE instructions
- Starts at trusted 32 byte aligned entry point
- Exits on any blacklisted instructions
  - Privileged instructions
  - Instructions that modify segment registers
  - ret
  - sysenter
CBI NaCl

or

Cross-Bundle Instruction Native Client

Types of Padding

- Indirect jump target
  - Will be placed at the next bundle start
- Call instruction
  - Will be placed at the end of the bundle
- Cross bundle instruction
  - Will be pushed to the start of next bundle

Padding vs Performance

- Change NaCl padding scheme
  - Pad removal
  - Greedy Algorithm

- Multipass Validator
  - We must guarantee sandboxing policy enforcement
  - Appropriate changes in validator

Types of Padding

- Indirect jump target
  - Will be placed at the next bundle start
- Call instruction
  - Will be placed at the end of the bundle
- Cross bundle instruction
  - Will be pushed to the start of next bundle
  - Conservative

Pad Removal
Pad Removal

Pad Removal

Pad Removal

Pad Removal

NaCl Validator

- One pass: from the start to the end of code
- Maintains two bitmaps: valid and target
- At each address checks the instruction
- If a valid instruction marks it in valid and advance by instruction size
- If indirect branch checks masking instruction presence
- If direct branch, the destination is marked in target
- At the end target and valid are compared together
Multipass Validator

- Challenge: Cross-Bundle Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>83 c8 01</td>
<td>or $0x1, %eax</td>
</tr>
<tr>
<td>88 41 28</td>
<td>mov %al, 0x28(%ecx)</td>
</tr>
<tr>
<td>8b 0e</td>
<td>mov (%esi), %ecx</td>
</tr>
<tr>
<td>8b 51 04</td>
<td>mov 0x4(%ecx), %edx</td>
</tr>
<tr>
<td>85 d2</td>
<td>test %edx, %edx</td>
</tr>
<tr>
<td>66 90</td>
<td>xchg %ax, %ax</td>
</tr>
<tr>
<td>0f 88 9a 01 00 00</td>
<td>js 1060600</td>
</tr>
<tr>
<td>8b 01</td>
<td>mov (%ecx), %eax</td>
</tr>
<tr>
<td>8d 3c 95 00 00 00 00</td>
<td>lea 0x0(,%edx,4), %edi</td>
</tr>
<tr>
<td>89 d6</td>
<td>mov %edx, %esi</td>
</tr>
<tr>
<td>83 ea 01</td>
<td>sub $0x1, %edx</td>
</tr>
<tr>
<td>83 e6 07</td>
<td>and $0x7, %esi</td>
</tr>
<tr>
<td>83 fa ff</td>
<td>cmp $0xffffffff, %edx</td>
</tr>
<tr>
<td>8d b6 00 00 00 00</td>
<td>lea 0x0(%esi), %esi</td>
</tr>
<tr>
<td>c7 04 38 84 06 03 11</td>
<td>movl $0x11030684, (%eax, %edi, 1)</td>
</tr>
<tr>
<td>8d 47 fc</td>
<td>lea -0x4(%edi), %eax</td>
</tr>
<tr>
<td>0f 84 70 01 00 00</td>
<td>je 1060600</td>
</tr>
</tbody>
</table>

Separate Compilation

- We process each source file separately
  - Decide about the paddings to be removed
  - Assemble them into object files (using modified GAS)
  - Then link them together

Relocations Problem

- movl $0x11015e2c, (%esp)
- jmp LABEL1

Thank you

Any Question?