Need for extensibility

- UNIX vnode interface
  - Add new file system
- Postgres database
  - User-defined data type
- Browser plugins
  - Incorporate plugins (possibly from untrusted sources)

Problem with extensions

- Security!
- Extensions may be
  - Malicious
  - Vulnerable
  - Faulty
- Solution:
  - Isolate from other codes

Isolation options

- Hardware-based isolation
  - Different virtual address space
  - Communicate via RPC

Isolation options (cont’d)

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

Goal

- Protect the rest of an application from a buggy/malicious module on RISC architecture
- Separate distrusted code
  - Define a fault domain
  - Prevent the module from jumping or writing outside of it
  - While letting efficient communications
- Security Policy:
  - No code is executed outside of fault domain
  - No data changed outside of fault domain
Fault Domain
- Load untrusted extension into its own fault domain
  - Code Segment
  - Data Segment

Unsafe Instruction
- Jump or store instructions
- Addressing issue
  - `jmp 10001e0`
  - `mov %eax,0x11020028`

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

Segment Matching
- Insert checking code before unsafe insn
  - check segment ID of target address
- Use dedicated registers

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

Unsafe Instruction
- Jump or store instructions
- Addressing issue
  - `jmp 10001e0`
  - `jmp *%ecx`
  - `mov %eax,0x11020028`
  - `mov $0x11018b80,%ecx`

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
- Prevents faults
- Needs 5 dedicated registers
- 2 extra instructions
  - less overhead compared to segment matching
**Process Resources**
- No direct syscall
- A trusted fault domain receives the syscall
- Determine if it is safe
- If so, make the syscall and return the result to distrusted code

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

**Cross Fault Domain Call**
- How to call from another 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

**Verifier**
- Divide program into unsafe regions
  - any modification to store dedicated register
    - start of store unsafe region
  - the store unsafe region ends when:
    - next instruction be a store (uses dedicated register)
    - next instruction be control flow change
    - next instruction is not guaranteed to be executed
    - no more instructions be in the code
  - at the end if dedicated register is not sandboxed correctly, reject the code
Performance Overhead

- 4.3% on average
- 21.8% when sandboxing read instructions as well

What about CISC architectures?!

x86

Evaluating SFI for a CISC Architecture (PittSFIeld)
Stephen McCamant, Greg Morrisett
USENIX 2005

CISC Architectures
- Processor can jump to any byte
- Hard to make hidden instructions safe

Solution
- 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 and its check

Jumps
- Chunks are atomic
- Jump destinations are checked to be 16-byte aligned
Optimization: AND-only Sandboxing

- Reduces sandboxing sequence to just one instruction
  - choose code and data region addresses carefully
  - Their ID just has one bit set

<table>
<thead>
<tr>
<th>Reserved</th>
<th>SFI Code</th>
<th>SFI Data</th>
<th>Trusted Code and Data</th>
</tr>
</thead>
</table>

Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>neg wdi</td>
<td>add $0x28, wesp</td>
<td>5-byte nop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7-byte nop</td>
<td>mov %esi, %edx</td>
<td>call 0xf59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>and $0xfffff0, %ebx</td>
<td>jmp *%ecx</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mov $0x488, %eax</td>
<td>sub %ecx, %eax</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>lea (%esi), %ebx</td>
<td>9-byte nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>and $0xffffffff, %ebx</td>
<td>mov %al, (%ebx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>shl %cl, %eax</td>
<td>test $8x7, %al</td>
<td>inc %cl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 native code in browser
- Browser?! Native Code?!?
  - Yes! browsers are new platform for applications
  - Gives Browser plugins performance of native code
  - Ships by default with Chrome 14
  - Very complex architecture
    - Focus on sandboxing technique

Sandboxing

- Inner Sandbox
  - Like PittSField
  - Alignment and address sandboxing
    - No cross boundary instructions
    - Jump target must be aligned
- Outer Sandbox
  - Controls system calls issued by native code
  - Whitelist

Inner Sandbox

- On x86-32 bit architecture
- Use segmented memory to guaranty data sandboxing
- Use 32-byte alignment to sandbox jumps
  - jump
  - call
  - return

  and $0xffffffff0, %ecx
  jmp *%ecx

and $0xffffffff0, %ecx
jmp *%ecx
Outer Sandbox

- Second layer of defense for native code
- Filters system calls
- On Linux uses ptrace
- Block any sys call not in whitelist
- For some, perform special argument checking
  - SYS_OPEN: can access to a whitelisted set of files
- Any violation from outer sandbox policy will terminate native code execution

Native Client Toolchain

- Modified GCC and GAS
  - To emit sandboxing instructions
- Final executable has .nexe extension
  - Compiled and linked as ELF file
- Can be disassembled using standard tools
  - objdump -d
    
    naclcall %ebx and $0xffffffe0,%ebx
    call *%ebx
    nacljmp %ecx and $0xffffffe0,%ecx
    jmp *%ecx

Performance Evaluation

- Imposes in average %5 overhead
- Sources of overhead
  - Inner sandbox
    - Alignment and padding
  - Outer sandbox
    - Syscall capturing and whitelisting

Recap

- Sandboxing
  - Execute untrusted code in a fault domain
- RISC
  - Simple instructions
  - Address Sandboxing
- CISC
  - Complex instructions
  - Address alignment
- Browser plugin
  - Benefit native performance in browser

Thank you
Questions?