Efficient Software-Based Fault Isolation

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

Presenter:
Christopher Head
# Approaches to Isolation

<table>
<thead>
<tr>
<th>Address Spaces</th>
<th>Software Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Control read/write sharing by <code>shm</code>, <code>mmap</code>, <code>chmod</code></td>
<td>● Control read/write sharing by userspace data structures</td>
</tr>
<tr>
<td>● No procedure calls</td>
<td>● Intra-process procedure calls</td>
</tr>
<tr>
<td>● Heavy-weight IPC</td>
<td>● Trivial IPC</td>
</tr>
<tr>
<td>● Zero overhead for pure execution</td>
<td>● Small but nonzero overhead for pure execution</td>
</tr>
</tbody>
</table>
Address Classification

Data or Code Address

- High
- Low

Variable, offset into isolated module

Fixed for this module
Different in all other modules
Module cannot change this!

**Code**: direct transfers statically validated; indirect transfers must go through dedicated register whose top bits are forced

**Data**: direct loads/stores statically validated; indirect loads/stores must go through dedicated register; guard regions

**Stack**: stack pointer is considered dedicated; sets checked rather than usages
Inter-Domain Control Transfers

- Untrusted modules can only transfer control to addresses in a *jump table* which contains secured entry points published by other modules.

- Jumps between untrusted modules go through trusted stubs which copy call parameters from caller data segment to callee data segment.
System Calls

- Untrusted modules cannot make system calls
- Untrusted modules access resources by calling into trusted modules to *request* access
- Trusted module checks whether access is permitted and arbitrates access to resources
Shared Memory

- Untrusted modules can share memory with each other:
  - Publisher and subscriber make inter-domain call into trusted module
  - Trusted module issues mmap call to alias physical RAM into application virtual address space
Implementations

- Compiler Modification
  - Modify compiler to output machine code with recognizable instruction sequences
  - Loader proves module obeys rules
  - Compiler optimization possible

- Binary Patching
  - Loader patches in trusted instruction sequences
  - Portable to any language or closed-source modules
  - Difficult to deal with dedicated registers
Performance

- Basic sandboxing: 4.3% overhead
  - Writes, control transfers restricted
  - Memory reads unrestricted
- Full sandboxing: 21.8% / 17.6% overhead
  - Memory reads also restricted
- Performance loss from register restriction: 0.4%
- Growth of instruction stream size: 10.5%
Conclusion

- System was developed and applied to a real-life example (PostgreSQL extensions)
- Small overhead occurs in performance, but overhead is much lower than overhead of using multiple processes for isolation and small enough to be acceptable especially for typical everyday applications
Questions

• Does this require two versions of GCC to be maintained?
  • Yes
  • Not very hard; already done for developing for multiple platforms or embedded systems

• Eating 4 out of 32 registers doesn't sound very nice!
  • Performance analysis says 0.4% overhead
  • 32 is a lot of registers
Questions

• Has this been done on X86?
  • Similar: Google Native Client
  • Easier:
    – X86 allows immediates encoded in instructions
    – No dedicated regs needed
  • Harder:
    – X86 instructions are variable length.
    – How can static analysis prove anything if you can jump into the middle of an instruction?
Questions

• Binary patching is mainstream now; VMWare does it for the entire operating system. Which approach (patching-vs-compiler) makes more sense from a performance POV?

• VMWare is slow unless you have VT, in which case it doesn't use binary patching anyway

• Compiler can never lose:
  – Apply the binary patch to its own output to break even.
  – Anything better is a win.
Questions

- Is it secure?
  - Yes:
    - Module is proven to only transfer control inside itself or to a jump table entry, and to write only to its own data memory.
    - All potentially-dangerous operations guarded by sequences of instructions carefully written to eliminate any danger even if only a suffix of the instructions are executed (e.g. by a malicious jump).
Questions

- Can't we solve this at a higher level with a well-defined RPC API?
  - No: defining an API doesn't guarantee untrusted modules will obey it.
- Do we need to modify the OS? How large is a segment? Will we run out of segments?
  - Not hardware segments
  - Defined entirely in userspace at any power-of-two size
Questions

- Parameters passed might be wrong. If callee doesn't sanity-check, could corrupt whole system?
  - Yes.
  - Trusted modules must treat published entry points like kernel system calls: \textit{everything must be checked}
Questions

- If two modules running in parallel wish to communicate by only passing pointers to objects back and forth, can they?
  - Shared memory
- In memory-bound program, will cache issues cause overhead?
  - Data layout is identical to native
  - Code size grows by ~10.5%
Questions

• What happens if there is a hardware fault?
  • OS sees fault
  • Application receives signal
  • Put signal handler in trusted module

• What is the purpose of the stubs?
  • Untrusted modules can't write to each other's data segments
  • Trusted stubs copy arguments between segments