Secure Virtual Architecture: Using LLVM to Provide Memory Safety to the Entire Software Stack

John Criswell, University of Illinois
Andrew Lenharth, University of Illinois
Dinakar Dhurjati, DoCoMo Communications Laboratories, USA
Vikram Adve, University of Illinois
What is Memory Safety?

Intuitively, the guarantees provided by a safe programming language (e.g., Java, C#)

- Array indexing stays within object bounds
- No uses of uninitialized variables
- All operations are type safe
- No uses of dangling pointers
- Control flow obeys program semantics
- Sound operational semantics
Benefits of Memory Safety for Commodity OS Code

- **Security**
  - Memory error vulnerabilities in OS kernel code are a reality\(^1\)

- **Novel Design Opportunities**
  - Safe kernel extensions (e.g. SPIN)
  - Single address space OSs (e.g. Singularity)

- **Develop New Solutions to Higher-Level Security Challenges**
  - Information flow policies
  - Encoding security policies in type system

---

1. Month of Kernel Bugs (http://projects.info-pull.com/mokb/)
Secure Virtual Architecture

- Compiler-based virtual machine underneath software stack
  - Uses analysis & transformation techniques from compilers
  - Supports commodity operating systems (e.g., Linux)
- Typed virtual instruction set enables sophisticated program analysis
- Provide safe execution environment for commodity OSs
Outline

- SVA Architecture
  - SVA Safety
  - Experimental Results
SVA System Architecture

- Applications
- OS Kernel
- Drivers
- OS Memory Allocator
- Safety Verifier
- Native Code Generator
- SVA-OS Run-time Library
- Memory Safety Run-time Library
- Safety Checking Compiler

- Hardware
- SVA ISA
- SVA Virtual Machine
- Native ISA
Software Flow

Compile-Time:

Kernel/Application Source

Safety Checking Compiler

Bytecode with Safe Types

Install/Load/Run-Time:

Safety Verifier

Bytecode + Run-Time Checks

Code Generator

Native Code

Hardware

TCB
Virtual Instruction Set

- **SVA-Core**
  - Subset of LLVM Instruction Set\(^1,2\)
  - Typed, Explicit Control Flow Graph, Explicit SSA form
  - Sophisticated compiler analysis and transformation

- **SVA-OS**
  - OS-neutral instructions support commodity OSs
  - Removes difficult to analyze assembly code
  - Encapsulates privileged operations
  - Like porting to a new hardware architecture

1. [CGO 2004]
2. [http://llvm.org]
Outline

- SVA Architecture
- SVA Safety
- Experimental Results
SVA Safety Guarantees

<table>
<thead>
<tr>
<th>Safe Language</th>
<th>Secure Virtual Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array indexing within bounds</td>
<td>Array indexing within bounds</td>
</tr>
<tr>
<td>No uses of uninitialized variables</td>
<td>No uses of uninitialized variables</td>
</tr>
<tr>
<td>Type safety for all objects</td>
<td>Type safety for subset of objects</td>
</tr>
<tr>
<td>No uses of dangling pointers</td>
<td>Dangling pointers are harmless</td>
</tr>
<tr>
<td>Control flow integrity</td>
<td>Control flow integrity</td>
</tr>
<tr>
<td>Sound operational semantics</td>
<td>Sound operational semantics</td>
</tr>
</tbody>
</table>

- Dangling pointers & non-type-safe objects do not compromise other guarantees
- Stronger than systems that do not provide any dangling pointer protection
Safety Checks & Transforms

Safety Checks
- Load/Store Checks
- Bounds Checks
- Illegal Free Checks
- Indirect Call Checks

Safety Transforms
- Stack to heap promotion
- Memory initialization

Object Bounds Tracking Methods
- “Fat” Pointers [SafeC, CCured, Cyclone,…]
- Programmer Annotations [SafeDrive,…]
- Object Lookups [Jones-Kelly, SAFECode,…]
Improved Object Lookups

- Alias analysis (DSA) groups objects into logical partitions
- Run-time records object allocations in partitions
- Run-time checks only consider objects in a single partition
- Reduces slowdown from 4x-11x to 10%-30% for nearly all standalone programs, daemons

1. Dhurjati et al. [ICSE 2006]
Type Safe (Homogeneous) Partitions

- Alias analysis performs type inference
- Type-homogeneous partitions reduce run-time checks:
  - No load/store checks
  - No indirect call checks
  - Harmless dangling pointers
- Type-unsafe partitions require all run-time checks
- Proved sound operational semantics [PLDI 2006]

1. Dhurjati et al. [TECS 2005, PLDI 2006]
Memory Allocator Requirements

- Memory for type-homogeneous partitions cannot be used by other partitions
- Objects must be aligned at a multiple of the object size
Outline

- SVA Architecture
- SVA Safety
  - *Experimental Results*
Prototype Implementation

- Ported Linux to SVA instruction set
  - Similar to porting to new hardware architecture
  - Compiled using LLVM
- Wrote SVA-OS as run-time library linked into kernel
- Provide safety guarantees to *entire* kernel except:
  - Memory management code
  - Architecture-dependent utility library
  - Architecture-independent utility library
Web Server Bandwidth

- Each measurement is median of 3 runs
- Memory safety overhead less than 70%

<table>
<thead>
<tr>
<th>File Size in KB</th>
<th>Percent Bandwidth Reduction Relative to Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58%</td>
</tr>
<tr>
<td>2</td>
<td>62%</td>
</tr>
<tr>
<td>4</td>
<td>64%</td>
</tr>
<tr>
<td>8</td>
<td>66%</td>
</tr>
<tr>
<td>16</td>
<td>68%</td>
</tr>
<tr>
<td>32</td>
<td>70%</td>
</tr>
<tr>
<td>64</td>
<td>72%</td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

Apache
Exploits

- Tried 5 memory exploits that work on Linux 2.4.22
- Uncaught exploit due to code not instrumented with checks

<table>
<thead>
<tr>
<th>BugTraq ID</th>
<th>Kernel Component</th>
<th>Caught?</th>
</tr>
</thead>
<tbody>
<tr>
<td>11956</td>
<td>Console Driver</td>
<td>Yes!</td>
</tr>
<tr>
<td>10179</td>
<td>TCP/IP</td>
<td>Yes!</td>
</tr>
<tr>
<td>11917</td>
<td>TCP/IP</td>
<td>Yes!</td>
</tr>
<tr>
<td>12911</td>
<td>Bluetooth Protocol</td>
<td>Yes!</td>
</tr>
<tr>
<td>13589</td>
<td>ELF/Support Library</td>
<td>No</td>
</tr>
</tbody>
</table>
Performance Improvements

- Source code changes
- Smarter run-time checks
  - Selective use of “fat” pointers
  - Pre-checking all accesses within monotonic loops
  - Removing redundant object lookups and run-time checks
  - Very fast indirect call checks
- Improve static analysis
  - Stronger type inference
  - More precise call graph
  - Restore context sensitivity
  - Static array bounds checking
Future Work

- Ensure safe use of:
  - SVA-OS instructions
  - MMU configuration
  - DMA operations

- Novel OS Designs
  - Recovery semantics for the virtual machine
  - Private application memory
  - Information flow
SAFECode Release

- Currently building memory debugger tool
  - Array bounds checks
  - Uninitialized pointer checks
  - Invalid control flow checks
  - Optional dangling pointer detection

1. Dhurjati et al. [DSN 2006]
Extras!

Questions?

See what we do at http://sva.cs.uiuc.edu