Memory Safety for Low-Level Software/Hardware Interactions

Montreal or Bust!

John Criswell
Nicolas Geoffray
Vikram Adve
Memory Safety Future is Bright

- User-space memory safety is improving
  - Safe languages
  - SAFECODE, CCured, Baggy bounds checking, Softbound, etc
- Memory safety for operating systems exists!
  - Singularity (C#), SPIN (Modula-3)
  - Linux on Secure Virtual Architecture (C)
A New Enemy Arises: Software/Hardware Interactions

- What is a low-level software-hardware interaction?
  - Instruction that manipulates hardware resources
  - Below semantics of the programming language

- Perfectly type-safe code! But:
  - Can corrupt control-flow or data-flow

- Examples:
  - Processor State
  - I/O Objects
  - MMU mappings
Memory Safety: Processor State

- Operating systems explicitly manage Processor State
  - Processor states saved in memory buffers
- Type-safe stores can modify a saved processor state
  - Can subvert control/data-flow integrity
Memory Safety: Processor State

- Operating systems explicitly manage Processor State
  - Processor states saved in memory buffers
- Type-safe stores can modify a saved processor state
  - Can subvert control/data-flow integrity
Memory Safety: Processor State

- Operating systems explicitly manage Processor State
  - Processor states saved in memory buffers
- Type-safe stores can modify a saved processor state
  - Can subvert control/data-flow integrity
Memory Safety: I/O

- I/O device memory and RAM in same address space
- However, I/O memory *is* different
  - I/O memory incompatible with standard compiler analysis
  - I/O memory has side effects on hardware

- Intel E1000E Bug on Linux 2.6
  - Invalid write on I/O memory
  - Damaged Intel E1000E Network Cards
  - Potential DoS Attack
MMU can violate type safety
Memory Safety: MMU

- MMU can violate type safety

MMU can violate type safety
Memory Safety: MMU

• MMU can violate type safety

• MMU can make kernel pages accessible to user-space
  - BID9356, BID9686, BID18177 (www.securityfocus.com)
Memory Safety: MMU

- MMU can violate type safety

- MMU can make kernel pages accessible to user-space
  - BID9356, BID9686, BID18177 (www.securityfocus.com)
It’s Already Here!

- Intel E1000E Bug
- MMU exploits in Linux

Need solutions *before* these attacks become more sophisticated and commonplace!
SVA-OS: Memory Safety for Low-Level Software-Hardware Interactions

- First system to provide comprehensive memory safety for low-level software/hardware interactions
  - Linux 2.4.22 on Secure Virtual Architecture (SVA)

- Compiler analysis and runtime checks
  - Little overhead above and beyond traditional memory safety

- Effective at preventing software/hardware exploits
Outline

- Motivation
- High-level Solutions
- Design of SVA-OS
- Experimental Results
- Future Work and Conclusions
Foundations: What Do We Need?

- System that provides traditional memory safety
  - SVA-OS *will preserve* memory safety

- Examples
  - Type-safe languages, e.g. Singularity
  - Compiler techniques for commodity operating systems, e.g. Secure Virtual Architecture (SVA)
Solution: Processor State

- New instruction to save old state and restore new state
  - State saved in internal SVA-OS memory
  - State referenced by ID returned from VM
- Policy left to OS
  - Scheduling, context switching, signal delivery
Solution: Memory Mapped I/O

- New instruction to map I/O memory into address space
- New instructions to load/store I/O objects
- Add run-time checks to ensure that:
  - Regular load/stores access memory
  - I/O accesses access I/O memory

```c
// Store value v at memory address P1
store(v, *p1);

// Store value v at I/O address P2
iostore(v, *p2);
```
Solution: Memory Mapped I/O

- New instruction to map I/O memory into address space
- New instructions to load/store I/O objects
- Add run-time checks to ensure that:
  - Regular load/stores access memory
  - I/O accesses access I/O memory

```
store (v, *p2);
```

```
iostore (v, *p1);
```
Solution: Memory Mapped I/O

- New instruction to map I/O memory into address space
- New instructions to load/store I/O objects
- Add run-time checks to ensure that:
  - Regular load/stores access memory
  - I/O accesses access I/O memory

```
store (v, *p2);
STORE (v, *p1);
```

Diagram:
- Memory Pointer (P1)
- I/O Pointer (P2)
Solution: MMU

- Add run-time checks on MMU updates
  - Mapping kernel memory into user-space
  - Mapping data inconsistent with types
- Same mechanism as VMMs
  - Finer-grain checks
Solution: MMU

- Add run-time checks on MMU updates
  - Mapping kernel memory into user-space
  - Mapping data inconsistent with types
- Same mechanism as VMMs
  - Finer-grain checks
Solution: MMU

- Add run-time checks on MMU updates
  - Mapping kernel memory into user-space
  - Mapping data inconsistent with types
- Same mechanism as VMMs
  - Finer-grain checks
Solution: MMU

- Add run-time checks on MMU updates
  - Mapping kernel memory into user-space
  - Mapping data inconsistent with types
- Same mechanism as VMMs
  - Finer-grain checks
Solution: MMU

- Add run-time checks on MMU updates
  - Mapping kernel memory into user-space
  - Mapping data inconsistent with types
- Same mechanism as VMMs
  - Finer-grain checks
Outline

- Motivation
- High-level Solutions
- Design of SVA-OS
- Experimental Results
- Future Work and Conclusions
Secure Virtual Architecture

- Compiler-based virtual machine
  - Hosts a commodity OS (e.g., Linux)
  - Provides traditional memory safety guarantees (control-flow and data-flow integrity)

Criswell et al. [SOSP 2007]
From SVA to SVA-OS

- Extend the SVA software/hardware interface
  - New instructions control software/hardware interactions

- Enforce memory safety for low-level operations
  - Use static analysis when possible
  - Add run-time checks when necessary
Solution: Processor State

- Save old state and place new state in a single instruction
  - `sva_swap_integer`
- Return opaque handle
- Buffer saved in SVA-OS memory
  - Buffer released on `sva_swap_integer` call

---

**Process 1: ID 1**
- R1
- R2
- PC
- SP

**Process 3: ID 2**
- R1
- R2
- PC
- SP

**Process 8: ID 3**
- R1
- R2
- PC
- SP

---

**OS Task Structures**

**SVA-OS Memory**

**CPU**
Solution: Processor State

- Save old state and place new state in a single instruction
  - `sva_swap_integer`
- Return opaque handle
- Buffer saved in SVA-OS memory
  - Buffer released on `sva_swap_integer` call

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
</tr>
</tbody>
</table>

**OS Task Structures**

**SVA-OS Memory**

**CPU**
Solution: Memory Mapped I/O

- Operating system uses a pseudo-allocator
  - Map I/O objects into virtual address space

- New instructions for I/O reads and writes
  - `sva_io_readb`, `sva_io_writeb`

- Compiler marks I/O memory as type-unknown
  - Load/store check on each access
  - Load/store checks on memory objects that alias
Solution: MMU

- VMM-like interface to declare and update MMU mappings
  - sva_declare_l1_page, sva_declare_l2_page
  - sva_update_l1_mapping, sva_update_l2_mapping

- Runtime checks for typed memory
  - Pointer analysis in SVA segregates data by types
  - SVA-OS ensures this stays consistent

- Run-time checks for dividing memory
  - SVA-OS memory and kernel memory
  - Kernel memory and user-space memory
  - I/O memory and regular kernel memory
Less than **100 lines changes** from original SVA Linux port

- `switch_to` → `sva_swap_integer`
- `readb` → `sva_io_readb`
- `set_pte` → `sva_update_l1_mapping`
- `pte_alloc_one` → `sva_declare_l1_page`

**Compiler changes:**

- Allocation of I/O objects: `ioremap`
Outline

- Motivation
- High-level Solutions
- Design of SVA-OS
- Experimental Results
- Future Work and Conclusions
Does It Work?

- Tested two real world MMU exploits
  - BID9356, BID9686 on Linux 2.4
  - BID18177 exploit code not available

- Injected errors into our Linux 2.4 port
  - New system calls

- Studied the E1000E Intel Network bug
  - Paper study because only on Linux 2.6
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

Map count = 1

User

Kernel

Virtual Memory

Physical Memory
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

![Diagram showing virtual memory, physical memory, user, kernel, virtual memory, and physical memory with a map count of 2.]
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

Map count = 3

User
Kernel
Virtual Memory
Physical Memory
MMU Exploits on Linux 2.4

- BID9356
  - `fork, mmap`

Diagram:
- User
- Kernel
- Virtual Memory
- Physical Memory
- Map count = 4
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

Map count = 0
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

Diagram:
- User
- Kernel
- Virtual Memory
- Physical Memory
- Map count = 0
MMU Exploits on Linux 2.4

- BID9356
- fork, mmap

Map count = 0

Virtual Memory

Physical Memory
MMU Exploits on Linux 2.4

- BID9356
  - fork, mmap

- BID9686
  - Missing error check on mremap
  - MMU mappings not cleared
MMU Exploits on Linux 2.4

- **BID9356**
  - `fork, mmap`

- **BID9686**
  - Missing error check on `mremap`
  - MMU mappings not cleared

Both bugs were detected by SVA-OS, not SVA
Error Injection

- Modification of Processor State
- Double mapping of a type-safe memory object
- Modify metadata of SVA with incorrect bounds
Error Injection

- Modification of Processor State
  
  *SVA: control flow changed*

  *SVA-OS: Caught as an invalid integer to pointer cast*

- Double mapping of a type-safe memory object

- Modify metadata of SVA with incorrect bounds
Error Injection

- Modification of Processor State
  - **SVA:** control flow changed
  - **SVA-OS:** Caught as an invalid integer to pointer cast
- Double mapping of a type-safe memory object
  - **SVA:** Subsequent store succeeds
  - **SVA-OS:** Second mapping caught by MMU checks
- Modify metadata of SVA with incorrect bounds
Error Injection

- Modification of Processor State
  - SVA: control flow changed
  - SVA-OS: Caught as an invalid integer to pointer cast
- Double mapping of a type-safe memory object
  - SVA: Subsequent store succeeds
  - SVA-OS: Second mapping caught by MMU checks
- Modify metadata of SVA with incorrect bounds
  - SVA: Memory safety guarantees disabled
  - SVA-OS: Access to SVA memory caught by MMU checks
E1000E Bug on Linux 2.6

- cmpxchg on dangling pointer
  - Instruction thought it was code memory
  - Unpredictable behavior on I/O memory
  - Network card damaged

- With SVA-OS
  - No I/O memory mapped on code page
  - Load/Store checks on I/O memory
Web Server Bandwidth: thttpd

- Athlon 2100+, 1GB of RAM, 1Gb/s network
- Higher is better
- Micro-benchmark overheads in paper

Web Server Bandwidth Normalized to Native

File Size (KB)
# User-Application Benchmarks

- Negligible overhead on user-space applications

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>i386 (s)</th>
<th>SVA (s)</th>
<th>SVA-OS (s)</th>
<th>% Increase (i386 to SVA-OS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>18.7</td>
<td>18.3</td>
<td>18.0</td>
<td>0.0%</td>
</tr>
<tr>
<td>lame</td>
<td>133.3</td>
<td>132.0</td>
<td>126.0</td>
<td>-0.1%</td>
</tr>
<tr>
<td>perl</td>
<td>22.3</td>
<td>22.3</td>
<td>22.3</td>
<td>0.0%</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- High-level Solutions
- Design of SVA-OS
- Experimental Results
- Future Work and Conclusions
Future Work

- Improve Static Analysis
  - Reduce run-time checks

- Additional Security Properties
  - Information flow control

- Apply to other systems
  - Type-safe language OS, e.g. Singularity
  - JVMs, hypervisors
Contributions

- Identified memory-safety violations from low-level software/hardware operations
- *First system* to provide comprehensive safety guarantees for such operations
  - Leaves control under OS
  - Incurs little run-time overhead above SVA

Questions?

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