Locking Down Insecure Indirection with Hardware-Based Control-Data Isolation

William Arthur, Sahil Madeka, Reetuparna Das, Todd Austin
Goal of this work

MAKE SOFTWARE MORE SECURE

Reducing the software attack surface by **subtracting** the root cause leading to many software exploits today

Accomplished by **locking down** insecure indirection
Every control transfer in executing application comes from the programmer:

- Every PC address encoded in instructions, OR
- Is derived from secure hardware structures

Executing application *always* adheres to the *programmer-defined* control-flow graph

**Stopping** control-flow **attacks** which derail the CFG
Achieved by hardware-software co-design

**Software:**
Eliminate all indirect control-flow instructions – via Control-Data Isolation (CDI) [1]

**Hardware:**
Memoization of secure control transitions in secure hardware – via Indirect Edge Cache

Outline

Software (in)security – Control-Flow Attack

Hardware-Based Control-Data Isolation

Measure performance and security

Conclusions
Control-Flow Attacks violate, at runtime, the CFG of an application by corrupting the PC with user-injected data.

- Buffer Overflow
- Heap Spray
- Return-to-libc
- Stack Smash
- Code Gadget

return ??????

local variables, return

attack
code

Stack Smash
Outline

- Software (in)security
- Hardware-Based Control-Data Isolation
- Measure performance and security
- Conclusions
int bar() {
    // function code
    return;
}

int bar() {
    // function code
    if ([%esp] == _ret1)
        jump _ret1;
    else if ([%esp] == _ret2)
        jump _ret1;
    else
        call _abort;
}

White-list of valid CFG edges

“Sled” of conditional branches and direct jumps
Software-only CDI (CGO ’15) retains higher than desired runtime overheads for some applications – 31% for gcc

Key insight: **Caching** previously executed sled edges obviates subsequent re-executions of the sled

Addition of hardware edge cache
Hardware-Based CDI Algorithm

- Indirect Instruction (*jmp, *call, ret)
- Check Edge Cache for <source,target> pair
- Hit?
- Miss?
- Fall-through to **sled**, retain <source>
- Taken branch from sled
- Cache <source,target> of taken branch

Execute Instructions
Hardware-Based CDI Algorithm

1. Indirect Instruction (jmp, call, ret)
2. Check Edge Cache for <source,target> pair
   - Hit?
   - Miss?
3. Fall-through to sled, retain <source>
4. Taken branch from sled
5. Cache <source,target> of taken branch
6. Execute Instructions
Hardware-Based CDI Algorithm

Indirect Instruction
(*jmp, *call, ret)

Check Edge Cache for <source,target> pair

Execute Instructions

Hit?

Miss?

Cache <source,target> of taken branch

Fall-through to sled, retain <source>

Taken branch from sled
Hardware-Based CDI Algorithm

- Execute Instructions
- Indirect Instruction (jmp, call, ret)
- Check Edge Cache for <source,target> pair
- Hit?
- Miss?
- Cache <source,target> of taken branch
- Fall-through to sled, retain <source>
- Taken branch from sled
Hardware-Based CDI Algorithm

1. **Execute Instructions**
   - Hit?
     - Check Edge Cache for \(<\text{source},\text{target}\>\) pair
     - Miss?
       - Indirect Instruction (jmp, call, ret)
2. **Cache \(<\text{source},\text{target}\>\) of taken branch**
3. **Fall-through to \textbf{sled}, retain \(<\text{source}\>\)**
4. **Taken branch from sled**
Hardware-Based CDI Algorithm

1. **Execute Instructions**
2. **Check Edge Cache for \(\langle\text{source, target}\rangle\) pair**
3. **Cache \(\langle\text{source, target}\rangle\) of taken branch**
4. **Fall-through to sled, retain \(\langle\text{source}\rangle\)**
5. **Indirect Instruction (jmp, call, ret)**

- Hit?
- Miss?

**Steps:**
- Taken branch from sled
- Hit?
- Miss?
Hardware-Based CDI Algorithm

- Place <source,target> in the Edge Cache
- Execute Instructions
  - Indirect Instruction (jmp, call, ret)
  - Check Edge Cache for <source,target> pair
    - Hit?
    - Miss?
      - Fall-through to sled, retain <source>
      - Taken branch from sled
Hardware-Based CDI Algorithm

1. **Indirect Instruction** (jmp, call, ret)
2. **Check Edge Cache for <source,target> pair**
3. **Hit?**
4. **Execute Instructions**
5. **Fall-through to sled, retain <source>**
6. **Cache <source,target> of taken branch**
7. **Miss?**
8. **Taken branch from sled**
Hardware-Based CDI Algorithm

- **Indirect Instruction** (*jmp, *call, ret)
  - Check Edge Cache for <source,target> pair

- **Execute Instructions**

- **Cache** <source,target> of taken branch

- **Hit?**

- **Fall-through to sled, retain <source>**

- **Miss?**

- **Taken branch from sled**
Hardware-Based CDI Algorithm

1. **Execute Instructions**
2. **Hit?**
   - **Cache** `<source,target>` of taken branch
3. **Miss?**
   - **Check Edge Cache for** `<source,target>` pair
4. **Indirect Instruction** (jmp, call, ret)
5. **Fall-through to sled**, retain `<source>`
6. **Taken branch from sled**
New hardware structure – **edge cache**

Memoization of most recent indirect edges
Edge Cache

Commit

Fetch

Edge Cache

Squash, execute sled

<src,target>

Yes

Retire

No
Edge Cache(2)

PC → BTB → Fetch → Edge Cache

GHR → Target Index

Commit → <src,target> → Yes → Retire

No → Squash, execute sled
Challenges

Edge Cache

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Target Address</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

128 + 2 bits per entry!
1k entries = 16 kB
Region Table

<table>
<thead>
<tr>
<th>Source Addr. Offset</th>
<th>Target Addr. Offset</th>
<th>G</th>
<th>Region Pointer(S)</th>
<th>G</th>
<th>Region Pointer(T)</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

offset, 18 bits

index, 5 bits
### Region Table

**Source Addr. Offset** | **Target Addr. Offset** | **G** | **Region Pointer(S)** | **G** | **Region Pointer(T)** | **U** | **V**
---|---|---|---|---|---|---

**Region Table**

<table>
<thead>
<tr>
<th>Region Address</th>
<th><strong>G</strong></th>
<th><strong>U</strong></th>
<th><strong>V</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Region Table

<table>
<thead>
<tr>
<th>Source Addr. Offset</th>
<th>Target Addr. Offset</th>
<th>G</th>
<th>Region Pointer(S)</th>
<th>G</th>
<th>Region Pointer(T)</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Edge Cache

50 bits per entry!
1k entries
6.75 kB total

full address
Outline

- Software (in)security
- Hardware-Based Control-Data Isolation
- Measure performance and security
- Conclusions
**Experimental Setup**

- **gem5** architectural simulator
  - Detailed **O3 cpu** model, configured similar to *Intel Haswell* processor, **x86-64**
- SPECINT 2000 & 2006
- 1,024-entry edge cache
  - 4-way set associative
- 32-entry region table
Benchmark Applications

Branch prediction – 6% speedup 400.perlbench vs BTB
**Average Indirect target Reduction – AIR [2]**

Measure of the reduction in the software attack surface

99.999%+ reduction in indirect target set

Average of **tens of targets** per indirect

**Previous works:** average of **tens of thousands** of targets per indirect instruction

Conclusions

- Locking down insecure indirection can eliminate contemporary control-flow attacks
- Hardware-based control-data isolation efficiently realizes this capability

\(<\textbf{Minimal runtime overhead} = 0.5\%\)
Questions?