HDFI: Hardware-Assisted Data-flow Isolation

Presented by Ben Schreiber

Chengyu Song¹, Hyungon Moon², Monjur Alam¹, Insu Yun¹, Byoungyoung Lee¹, Taesoo Kim¹, Wenke Lee¹, Yunheung Paek²

¹Georgia Institute of Technology
²Seoul National University
A simple stack overflow

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

```
main:
1  add    sp,sp,-32
2  sd     ra,24(sp)
3  ld     a1,8(a1) ; argv[1]
4  mv     a0,sp    ; char buff[16]
5  call   strcpy    ; strcpy(buff, argv[1])
6  li     a0,0
7  ld     ra,24(sp)
8  add    sp,sp,32
9  jr      ra       ; return
```
Defense mechanisms

```c
int main(int argc, const char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

```
1   main:
2       add    sp,sp,-32
3       sd     ra,24(sp)
4       ld     a1,8(a1) ; argv[1]
5       mv     a0,sp ; char buff[16]
6       call   strcpy ; strcpy(buf, argv[1])
7       li     a0,0
8       ld     ra,24(sp)
9       add    sp,sp,32
10      jr      ra ; return
```
Prior Limitations

• Software: lacks good isolation mechanisms in 64-bit world
  • SFI and virtual address space: **secure** but **expensive**
  • Address randomization: **efficient** but **insecure**

• Hardware: lacks **flexibility**
  • Context saving/restoring (setjmp/longjmp), deep recursion, kernel stack, etc.
  • Other data: code pointers, non-control data

• Data shadowing: adds **overheads**
  • Breaks data locality, needs additional step to look up or reserved register(s)
  • Occupies additional memory
Hardware-assisted data-flow isolation

- **Secure** and **efficient**
  - Low performance overhead and strong security guarantees

- **Flexible**
  - Capable of supporting different security model/mechanisms

- **Fine-grained**
  - No more data-shadowing

- **Practical**
  - Minimized hardware changes
Data-flow Integrity [OSDI’06]

Runtime data-flow should not deviate from static data-flow graph

```
1  main:
2    add    sp, sp, -32
3    sd     ra, 24(sp)
4    ld     a1, 8(a1); argv[1]
5    mv     a0, sp; char buff[16]
6    call   strcpy; strcpy(buff, argv[1])
7    li     a0, 0
8    ld     ra, 24(sp)
9    add    sp, sp, 32
10   jr      ra; return
```
ISA extension

• Tagged memory
  • Machine word granularity
  • Fixed tag size → currently only 1 bit (sensitive or not)

• Three new *atomic* instructions to enable DFI-style checks
  • *sdset1, ldchk0, ldchk1*

• New semantic of old instructions (backward compatible)
  • *sd*: *sdset0*
  • *ld*: does not check tags
Hardware extension

• Cache extension
  • Extra bits in the cache line for storing the tag (reusing existing cache coherence interconnect)

• Memory Tagger
  • Emulating tagged memory without physically extending the main memory
  • Tags are stored in separate table in protected main memory
Optimizations

• Memory Tagger introduces additional performance overhead
  • Naive implementation: 2x memory accesses, 1 for data, 1 for tag

• Three optimization techniques
  • Tag cache
  • Tag valid bits (TVB)
  • Meta tag table (MTT)
Tag Cache

• A 64 byte cache line needs only 8 tag bits

• The memory interface is a fixed 64 bytes
  • Many lines of tags (enough for 4 KB block) can be fetched at the same time

• Tag cache located within DFITagger block
  • 64 byte entries corresponding to 4 KB of memory
Tag Valid Bits (TVB)

• Observation: a majority of memory accesses do not check tags
• If the processor does not ask for tags, do not lookup/supply tags
  • TVB indicates whether tags are present with data in cache
• Loads will ask DFITagger for tags if they are needed and not present
• Writes always set TVB
Meta Tag Table

- Observation: most memory is tagged with 0
- Meta Tag Table is an in-memory data structure where each bit indicates whether a group of entries in the Tag Table are all zero
- Meta Tag Directory is a register that functions similarly on the Meta Tag Table
- These structures can eliminate some memory reads
Return address protection

• Policy: return address should always have tag 1

• Benefits: secure and supports context saving/restoring, deep recursion, modified return address, kernel stack

```
1  main:
2    add    sp,sp,-32
3    *sdset1 ra,24(sp)
4    ld    a1,8(a1)  ; argv[1]
5    mv    a0,sp     ; char buff[16]
6    call  strcpy    ; strcpy(buff, argv[1])
7    li    a0,0
8    *ldchk1 ra,24(sp)
9    add    sp,sp,32
10   jr     ra       ; return
```
Standard Library Protections

• Heap Metadata
  • Protect pointers within ptmalloc

• Global Offset Table
  • GOT is a data structure used for dynamic linking
  • Add tags to table at same time as ASLR is applied

• Exit handler
  • Currently pointer is (weakly) encrypted
  • Use tag on pointer to ensure security
# Various applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Security Policy (invariants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow Stack</td>
<td>return address and register spills should have tag 1 (push / pop)</td>
</tr>
<tr>
<td>\texttt{vptr} Protection</td>
<td>\texttt{vptr} should have tag 1 (constructor / virtual function call)</td>
</tr>
<tr>
<td>Code Pointer Separation</td>
<td>code pointer should have tag 1 (CPI [OSDI’14])</td>
</tr>
<tr>
<td>C Library Enhancement</td>
<td>important data/pointers should have tag 1 (manual modification)</td>
</tr>
<tr>
<td>Kernel Protection</td>
<td>sensitive kernel data should have tag 1 (Kenali [NDSS’16])</td>
</tr>
<tr>
<td>Heartbleed Prevention</td>
<td>crypto keys should have tag 1</td>
</tr>
<tr>
<td></td>
<td>output buffer should have tag 0</td>
</tr>
</tbody>
</table>
Implementations

• Hardware
  • RISC-V RocketCore generator: 2198 LoC
  • Instantiated on Xilinx Zynq ZC706 FPGA board

• Software (RISC-V toolchain)
  • Assembler gas: 16 LoC
  • Kernel modifications: 60 LoC
  • Security applications: 170 LoC
Effectiveness of optimizations

- Memory bandwidth and latency

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Tag Cache</th>
<th>+TVB</th>
<th>+MTT</th>
<th>+TVB+MTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 hit</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>L1 miss</td>
<td>14.47%</td>
<td>5.26%</td>
<td>14.47%</td>
<td>5.26%</td>
</tr>
<tr>
<td>Copy</td>
<td>13.14%</td>
<td>4.44%</td>
<td>11.84%</td>
<td>4.26%</td>
</tr>
<tr>
<td>Scale</td>
<td>10.62%</td>
<td>4.79%</td>
<td>9.45%</td>
<td>4.67%</td>
</tr>
<tr>
<td>Add</td>
<td>4.37%</td>
<td>1.26%</td>
<td>4.13%</td>
<td>1.2%</td>
</tr>
<tr>
<td>Triad</td>
<td>9.66%</td>
<td>1.96%</td>
<td>8.8%</td>
<td>1.83%</td>
</tr>
</tbody>
</table>

- SPEC CINT2000

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Tag Cache</th>
<th>+TVB</th>
<th>+MTT</th>
<th>+TVB+MTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>16.09%</td>
<td>2.18%</td>
<td>6.85%</td>
<td>1.87%</td>
</tr>
<tr>
<td>175.vpr</td>
<td>29.51%</td>
<td>3.26%</td>
<td>7.71%</td>
<td>1.43%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>36.89%</td>
<td>3.08%</td>
<td>13.66%</td>
<td>-0.11%</td>
</tr>
<tr>
<td>197.parser</td>
<td>16.11%</td>
<td>2.27%</td>
<td>7.61%</td>
<td>1.53%</td>
</tr>
<tr>
<td>254.gap</td>
<td>12.19%</td>
<td>1.04%</td>
<td>6.53%</td>
<td>0.71%</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>14.52%</td>
<td>2.65%</td>
<td>3.63%</td>
<td>0.84%</td>
</tr>
<tr>
<td>300.twolf</td>
<td>26.71%</td>
<td>2.97%</td>
<td>7.37%</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

- Negative value is artifact from minor differences between runs
Security experiments

• With synthesized attacks

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Attacks</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow stack</td>
<td>RIPE</td>
<td>✔</td>
</tr>
<tr>
<td>Heap metadata protection</td>
<td>Heap exploit</td>
<td>✔</td>
</tr>
<tr>
<td>VTable protection</td>
<td>VTable hijacking</td>
<td>✔</td>
</tr>
<tr>
<td>Code pointer separation (CPS)</td>
<td>RIPE</td>
<td>✔</td>
</tr>
<tr>
<td>Code pointer separation (CPS)</td>
<td>Format string exploit</td>
<td>✔</td>
</tr>
<tr>
<td>Kernel protection</td>
<td>Privilege escalation</td>
<td>✔</td>
</tr>
<tr>
<td>Private key leak prevention</td>
<td>Heartbleed</td>
<td>✔</td>
</tr>
</tbody>
</table>
Impacts on security solutions

- Security
  - Hardware-enforced isolation

- Simplicity
  - No data shadowing

- Usability
  - Implementation/port is very easy

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow Stack</td>
<td>C++ (LLVM 3.3)</td>
<td>4</td>
</tr>
<tr>
<td>VTable Protection</td>
<td>C++ (LLVM 3.3)</td>
<td>40</td>
</tr>
<tr>
<td>CPS</td>
<td>C++ (LLVM 3.3)</td>
<td>41</td>
</tr>
<tr>
<td>Kernel Protection</td>
<td>C (Linux 3.14.41)</td>
<td>70</td>
</tr>
<tr>
<td>Library Protection</td>
<td>C (glibc 2.22)</td>
<td>10</td>
</tr>
<tr>
<td>Heartbleed Prevention</td>
<td>C (OpenSSL 1.0.1a)</td>
<td>2</td>
</tr>
</tbody>
</table>
Impacts on security solutions (cont.)

• Efficiency
  • GCC (-O2)
  • Clang (-O0)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Shadow stack (GCC)</th>
<th>SS+CPS (Clang)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>1.12%</td>
<td>2.42%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.76%</td>
<td>3.54%</td>
</tr>
<tr>
<td>254.gap</td>
<td>3.34%</td>
<td>13.23%</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>3.05%</td>
<td>4.61%</td>
</tr>
</tbody>
</table>
Security analysis

• Attack surface
  • Inaccuracy of data-flow analysis
  • Deputy attacks

• Best practices
  • CFI is necessary (e.g., CPS + shadow stack)
  • Recursive protection of pointers
  • Guarantee the trustworthiness of the written value
  • Use runtime memory safety technique to compensate inaccuracy of static analysis
Limitations and Improvements

• Direct memory access (DMA)
  • Attacker could directly alter tag table in memory

• Further optimizations
  • Add tag prefetch, improved cache, etc. to complement an OoO core

• Dynamic Code Generation
  • Not currently supported, but tags could be used to secure JIT’ed code
Q & A

Thank you!