A Software Solution for Hardware Vulnerabilities

Komail Dharsee  
*University of Rochester*

Ethan Johnson  
*University of Rochester*

John Criswell  
*University of Rochester*
Hardware Bugs!

- AMD recently shipped new chips to fix the FMA3 bug.
- User-space workloads can cause system-wide denial of service.
- Manufacturers can disable buggy hardware features.
  - Features (and associated instructions) become unavailable!
Hardware Bugs in All Shapes and Sizes

- Hand-written machine code may freely trigger hardware bugs
- User space code can cause system-wide disruption
- Current hardware-enforced privilege is insufficient
  - Unsafe privileged (kernel mode) execution
  - Control Flow Integrity and Corrupted Instruction Pointers
Our Contributions

- Hardware Bug Survey
- Bug Mitigation System Design

*Image Copyright, 1995 Pixar*
Outline

1. Background
   ▶ Software Mediation with Hardware Bugs
   ▶ Secure Virtual Architecture (SVA)
   ▶ SVA’s Virtual Instruction Set (V-ISA)

2. Design
   ▶ Safe & Verified Code Generation
   ▶ Run-time Checks
   ▶ SVA-OS Configuration

3. Hardware Bug Survey Results

4. Conclusions
Security Critical Hardware Bugs

- Hardware can have bugs due to design or manufacturing flaws.
- Attackers can leverage hardware bugs to violate security policies.
- We analyze processor bugs triggered through system configuration or execution.
Software Mediation of Hardware Bugs

- Systems should not allow execution of arbitrary native code.
- Utilize a small layer of software to restrict code expressiveness.
- Such software can be designed in tandem with hardware.
Utilizing Secure Virtual Architecture (SVA)

- SVA is a compiler-based virtual machine
- All code on an SVA system is compiled to SVA’s Virtual ISA
- SVA enforces security policies by inserting run-time checks.
  - Control Flow Integrity
  - Memory Safety
  - Software Fault Isolation
- SVA-OS provides an interface to low level operations
  - Context Switching
  - MMU Configuration
Benefits of SVA’s Virtual Instruction Set Architecture (V-ISA)

- Translation from V-ISA to N-ISA done ahead-of-time or just-in-time
- Separation of program memory
- Local branches which are calculated at run-time cannot be expressed
  - Local Control Flow Graph (CFG) doesn’t need to be computed
- Single Static Assignment (SSA) enables efficient data flow analysis
1. Background
   ▶ Software Mediation with Hardware Bugs
   ▶ Secure Virtual Architecture (SVA)
   ▶ SVA’s Virtual Instruction Set (V-ISA)

2. **Design**
   ▶ Safe & Verified Code Generation
   ▶ Run-time Checks
   ▶ SVA-OS Configuration

3. Hardware Bug Survey Results

4. Conclusions
Three methods for SVA to thwart security critical processor bugs

1. Safe & Verified Code Generation
2. Run-time Checks
3. SVA-OS Configuration
Safe & Verified Code Generation

- Modify code generators, ensuring unsafe instructions are not generated
- All code running on the system is compiled to the SVA V-ISA
- FSINCOS $\rightarrow$ FSIN, FCOS
Safe & Verified Code Generation: Example

1. `fld qword ptr [rbp + qword-24]`
2. `fsincos`
3. `fstp qword ptr [qword%eax]`
Safe & Verified Code Generation: Example

1. `fld qword ptr [rbp + qword -24]`
2. `fsincos`
3. `fstp qword ptr [qword % eax]`
Safe & Verified Code Generation: Example

1. `fld qword ptr [rbp + qword -24]`
2. `fsincos`
3. `fistp qword ptr [qword%eax]`
### Safe & Verified Code Generation: Example

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>fld qword ptr [rbp + qword-24]</code></td>
</tr>
<tr>
<td>2</td>
<td><code>fsincos</code></td>
</tr>
<tr>
<td>3</td>
<td><code>fstp qword ptr [qword%eax]</code></td>
</tr>
</tbody>
</table>
Safe & Verified Code Generation: Example

```
1. fld qword ptr [rbp + qword-24]
2. fsin
3. fcos
4. fstp qword ptr [qword%eax]
```
Safe & Verified Code Generation: Example

1. `fld qword ptr [rbp + qword-24]`
2. `fsin`
3. `fcos`
4. `fstp qword ptr [qword%eax]`
Safe & Verified Code Generation: Example

1. `fld qword ptr [rbp + qword-24]`
2. `fsin`
3. `fcos`
4. `fstp qword ptr [qword%eax]`
Safe & Verified Code Generation

- Code Verification through post-generation static analysis

\[ \text{ProgramPDA} \cap \text{BugDescFSA} \rightarrow \text{BugTriggerPDA} \]
Run-time Checks

- Necessary to deter bugs unthwarted by static analysis
- Native code may put the processor into an unsafe state
- SVA can insert run-time checks during code generation
- Run-time checks can alter processor state
  - Bug trigger mitigations
  - Recovery Routines
- syscall Bug: Disallow breakpoint on syscall.
### Run-time Checks: Example

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>movl $0, -4(%rbp)</td>
<td>Move immediate 0 to -4(%rbp)</td>
</tr>
<tr>
<td>2</td>
<td>movl %edi, -8(%rbp)</td>
<td>Move %edi to -8(%rbp)</td>
</tr>
<tr>
<td>3</td>
<td>movq %rsi, -16(%rbp)</td>
<td>Move %rsi to -16(%rbp)</td>
</tr>
<tr>
<td>4</td>
<td>movl $5, -24(%rbp)</td>
<td>Move immediate 5 to -24(%rbp)</td>
</tr>
<tr>
<td>5</td>
<td>syscall</td>
<td>System call</td>
</tr>
<tr>
<td>6</td>
<td>cmpl $0, -24(%rbp)</td>
<td>Compare immediate 0 to -24(%rbp)</td>
</tr>
<tr>
<td>7</td>
<td>jle .LBB0_2</td>
<td>Jump if less than or equal to .LBB0_2</td>
</tr>
</tbody>
</table>
Run-time Checks: Example

```assembly
1  movl  $0, -4(%rbp)
2  movl  %edi, -8(%rbp)
3  movq  %rsi, -16(%rbp)
4  movl  $5, -24(%rbp)
5  syscall
6  cmpl  $0, -24(%rbp)
7  jle .LBB0_2
```
Run-time Checks: Example

1. movl $0, -4(%rbp)
2. movl %edi, -8(%rbp)
3. movq %rsi, -16(%rbp)
4. movl $5, -24(%rbp)
5. syscall
6. cmpl $0, -24(%rbp)
7. jle .LBB0_2
Run-time Checks: Example

1. movl $0, -4(%rbp)
2. movl %edi, -8(%rbp)
3. movq %rsi, -16(%rbp)
4. movl $5, -24(%rbp)
5. syscall
6. cmpl $0, -24(%rbp)
7. jle .LBB0_2
Run-time Checks: Example

```
1             movl $0, -4(%rbp)
2             movl %edi, -8(%rbp)
3             movq %rsi, -16(%rbp)
4             movl $5, -24(%rbp)
5             syscall
6             cmpl $0, -24(%rbp)
7             jle .LBB0_2
```
Run-time Checks: Example

1. `movl $0, -4(%rbp)`
2. `movl %edi, -8(%rbp)`
3. `movq %rsi, -16(%rbp)`
4. `movl $5, -24(%rbp)`
5. `syscall`
6. `cmpl $0, -24(%rbp)`
7. `jle .LBB0_2`
Run-time Checks: Example

movl  $0, -4(%rbp)
movl  %edi, -8(%rbp)
movq  %rsi, -16(%rbp)
movl  $5, -24(%rbp)
nop
syscall
cmpl  $0, -24(%rbp)
jle   .LBB0_2
Run-time Checks: Example

1. movl $0, -4(%rbp)
2. movl %edi, -8(%rbp)
3. movq %rsi, -16(%rbp)
4. movl $5, -24(%rbp)
5. nop
6. syscall
7. cmpl $0, -24(%rbp)
8. jle .LBB0_2
Run-time Checks: Example

```assembly
 1  movl $0, -4(%rbp)
 2  movl %edi, -8(%rbp)
 3  movq %rsi, -16(%rbp)
 4  movl $5, -24(%rbp)
 5  nop
 6  syscall
 7  cmpl $0, -24(%rbp)
 8  jle .LBB0_2
```
Run-time Checks: Example

1. movl $0, -4(%rbp)
2. movl %edi, -8(%rbp)
3. movq %rsi, -16(%rbp)
4. movl $5, -24(%rbp)
5. nop
6. syscall
7. cmpl $0, -24(%rbp)
8. jle .LBB0_2
Run-time Checks: Example

```
1  movl  $0,  -4(%rbp)
2  movl  %edi,  -8(%rbp)
3  movq  %rsi,  -16(%rbp)
4  movl  $5,  -24(%rbp)
5  nop
6  syscall
7  cmpl  $0,  -24(%rbp)
8  jle   .LBB0_2
```
### Run-time Checks: Example

```
1. movl $0, -4(%rbp)
2. movl %edi, -8(%rbp)
3. movq %rsi, -16(%rbp)
4. movl $5, -24(%rbp)
5. nop
6. syscall
7. cmpl $0, -24(%rbp)
8. jle .LBB0_2
```
Run-time Checks: Example

```
1     movl  $0,  -4(%rbp)
2     movl  %edi,  -8(%rbp)
3     movq  %rsi,  -16(%rbp)
4     movl  $5,  -24(%rbp)
5     nop
6     syscall
7     cmpl  $0,  -24(%rbp)
8     jle   .LBB0_2
```
### Run-time Checks: Example

```
1  movl $0, -4(%rbp)
2  movl %edi, -8(%rbp)
3  movq %rsi, -16(%rbp)
4  movl $5, -24(%rbp)
5       nop
6       syscall
7  cmpl $0, -24(%rbp)
8  jle .LBB0_2
```
SVA-OS Configuration

- SVA-OS provides an interface to the kernel for low-level tasks.
- SVA-OS prevents OS from expressing unsafe system configurations.
- AMD Erratum #744: Trap registers not restored on power state transition.
- SVA-OS disables the trap registers; abstracts away enabling capabilities.
1. Background
   - Software Mediation with Hardware Bugs
   - Secure Virtual Architecture (SVA)
   - SVA’s Virtual Instruction Set (V-ISA)

2. Design
   - Safe & Verified Code Generation
   - Run-time Checks
   - SVA-OS Configuration

3. **Hardware Bug Survey Results**

4. Conclusions
Bug Mitigation Survey Results

- Run-time Checks: 20%
- Grammar Checks: 13%
- SVA Cannot Mitigate: 7%
- Insufficient Documentation: 13%
- SVA-OS: 47%

*Excludes bugs in unsupported hardware virtualization features*
Conclusions

- Design mitigates 80% of supported processor bugs
- Bug Survey shows high SVA-OS implementation priority

- Address bugs which trigger under “complex set[s] of internal timing conditions.” [1]
- Evaluate performance overhead
- Experiment with highly verified minimalistic components
Advanced Micro Devices Inc.
Revision guide for AMD family 11h processors, December 2011.