Dynamic Execution and Integrity Engine

An IoT-Class Hardware Monitor for
Real-Time Fine-Grained Control-Flow Integrity
Security Considerations

Scenario

Real-Time IoT device attached to security-critical memory-mapped devices

Embedded Device

RISC-V Processor Core

FE DE EX ME WB

Dangerous MMIO Write Access

Smart Lock

Industrial Robot

Smart Drone
Security Considerations
Attacker Model

Code-Reuse Attacks: Attacker can alter control-flow (CF) instructions

- Tampering with return addresses
  - Return-oriented Programming Attacks
  - Return into LibC Attacks
- Tampering also with function-local Control Flow Instructions
  - Jump-oriented Programming Attacks

<table>
<thead>
<tr>
<th>Stack</th>
<th>Shadow Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x88</td>
<td>≠ 0x80</td>
</tr>
<tr>
<td>0x184</td>
<td>= 0x184</td>
</tr>
</tbody>
</table>

Basic Block A

Basic Block B

Basic Block C
Security Considerations
Guarantees

- DExIE will react to any CF instruction violating the currently active control flow constraints within a *guaranteed time interval*
- DExIE blocks *any* MMIO write following an illegal CF instruction
Security Considerations
Assumptions

Code-injection is not covered

- Can be mitigated by other measures: MMU, MPU, DEP, ROM, ...

```c
int ATTACKER_main()
{
    float evil = 1.0;
    if(evil){
        destroy_world();
    }
    return 0;
}
```

DExIE focuses on defense against code reuse attacks
Software / Hardware Partitioning
Software-Only vs. Hardware-Assisted vs. Hardware-Only

Instrument binary with extra instructions supervise CF instructions
Relatively easy, but high overhead (e.g., 2x slowdown)

```c
int main(){
    // Save return address duplicate or its hash to safe place
    int b=0;
    if(b) {
        get(R);
    }
    // Compare return address duplicate
    return 0;
}
```
Software / Hardware Partitioning
Software-Only vs. Hardware-Assisted vs. Hardware-Only
Software / Hardware Partitioning
Software-Only vs. Hardware-Assisted vs. Hardware-Only
Architectural Location for Integrity Unit

1. In-Pipeline

++ Lowest latency
-- Lowest portability

2. On-Chip

+ Slightly increased latency
+ Good portability

3. Off-Chip

++ Best Portability
-- Very long latency
-- Throughput limitation
Monitoring vs. Enforcement

Monitoring: One-way flow of observations, unit passively monitors violations
Monitoring vs. Enforcement

Enforcement: Closed loop operation, unit can actively prevent violations from taking effect
Parallelism of Execution and Checking
Serial vs. Parallel vs. Hybrid

Program Execution
Control Flow Evaluation
Program Execution
Control Flow Evaluation

CF Instruction

CF Instruction
Parallelism of Execution and Checking
Serial vs. Parallel vs. Hybrid
Parallelism of Execution and Checking
Serial vs. Parallel vs. Hybrid

Stall execution of instruction until checking is complete
Mitigating the Worst-Case Attack

RISC-V Core

Actors

DF DF DF WB

Signal Taps

Reset

Prevent Violations:
Stall, Reset, (Interrupt)

Observing *uncommitted* instructions

Enforcement Unit
DExIE

C. Spang, Y. Lavan, M. Hartmann, F. Meisel, A. Koch | TU Darmstadt | An IoT-Class Hardware Security Accelerator
Major Challenges for DExIE

- Speed, Real Time: Avoid stalls and long critical path
- Guarantee Security: Be fast enough to block subsequent MMIO access
- Compatibility: Be compatible with multiple RISC-V cores
- Efficient: Compact memory layout supporting fast access

- Constraints memory storage:
  - Random access, no caching
  - Low latency (<= 1 clock cycle)
  - Tight layout
## Solution: Encoding CF Constraints as FSMs

### Hierarchical Approach

<table>
<thead>
<tr>
<th>C Code</th>
<th>RISC-V Assembly</th>
<th>Function-local FSMs</th>
<th>Hierarchy of FSMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getR(){</td>
<td>144: &lt;getR&gt;</td>
<td>144-160 State 0</td>
<td>144-160 State 0</td>
</tr>
<tr>
<td>int=42;</td>
<td>144-15c: non CFI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return i;</td>
<td>160: ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int main(){</td>
<td>164: &lt;main&gt;</td>
<td>164-17c State 0</td>
<td>164-17c State 0</td>
</tr>
<tr>
<td>int b=0;</td>
<td>164-178: non CFI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(b) {</td>
<td>17c: beqz 184</td>
<td>Untaken branch</td>
<td>Untaken branch</td>
</tr>
<tr>
<td>get(R);</td>
<td>180: jal&lt;getR&gt;</td>
<td>taken branch</td>
<td>taken branch</td>
</tr>
<tr>
<td>}</td>
<td>184-194: non CFI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td>198: ret</td>
<td>184-198 State 2</td>
<td>184-198 State 2</td>
</tr>
</tbody>
</table>

**C Code**

```c
int getR(){
    int=42;
    return i;
}

int main(){
    int b=0;
    if(b) {
        get(R);
    }
    return 0;
}
```

**RISC-V Assembly**

144: <getR>
144-15c: non CFI
160: ret
164: <main>
164-178: non CFI
17c: beqz 184
180: jal<getR>
184-194: non CFI
198: ret

**Function-local FSMs**

- State 0
- State 1
- State 2

**Hierarchy of FSMs**

- Call 144
- Ret 184
- Untaken branch
- Taken branch
Solution: Encoding CF Constraints as FSMs

Memory Layout

Transition Table

Function 0

<table>
<thead>
<tr>
<th>Transition 0</th>
<th>...</th>
<th>Transition M</th>
</tr>
</thead>
</table>

Function N

<table>
<thead>
<tr>
<th>Transition 0</th>
<th>...</th>
<th>Transition P</th>
</tr>
</thead>
</table>

Local Address Mapping

<table>
<thead>
<tr>
<th>Address 0</th>
<th>...</th>
<th>Address K</th>
</tr>
</thead>
</table>

Global Address Mapping

<table>
<thead>
<tr>
<th>Address 0</th>
<th>...</th>
<th>Address L</th>
</tr>
</thead>
</table>
Evaluation: Clock Frequency Cost
Depends on Micro-Architecture of Monitored CPU

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Frequency Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>VexRiscv</td>
<td>-42.55%</td>
</tr>
<tr>
<td>Taiga</td>
<td>-32.50%</td>
</tr>
<tr>
<td>PicoRV32</td>
<td>-67.35%</td>
</tr>
<tr>
<td>Piccolo</td>
<td>+5.8%</td>
</tr>
</tbody>
</table>

Frequency drop:
-42.55%
-32.50%
-67.35%
+5.8%
Evaluation: Area Cost
For different FPGA Resources

![Bar chart showing area cost for different FPGA resources: VexRiscv, Taiga, PicoRV32, Piccolo. The chart compares BRAM Overhead, Register Overhead, and LUT Overhead.]
Evaluation: Execution Time Costs
Wall-Clock Execution Time

Wall Clock Execution Time in Seconds

- VexRiscv:
  - Ud DExIE: 74.4%
  - Ud: 73.9%
  - Matmult-Int DExIE: 74.1%
  - Matmult-Int: 71.7%

- Taiga:
  - Ud DExIE: -48.15%

- PicoRV32:
  - Ud DExIE: -134.5%
  - Ud: 5.48%
  - Matmult-Int DExIE: 74.4%
  - Matmult-Int: 73.9%
  - Edn DExIE: 74.1%
  - Edn: 71.7%
  - Aha-Mont64 DExIE: 5.48%
  - Aha-Mont64: 48.15%

Statically predictable stall cycles:
- < 2%
- < 10.4%
- = 0%
- < 3%
Conclusion

DExIE is …

• … real-time capable, as all stalls are statically predictable for WCET
• … generally faster than SW-instrumented code (has less than 2x slowdown)
• … portable and can easily be attached to different IoT-class processors
• … smaller than the guard processor approach (which would use 2x area)
• … flexible, as it can enforce CF at multiple granularities

• Questions?