Securing Real-Time Microcontroller Systems through Customized Memory View Switching

Chung Hwan Kim*, Taegyu Kim, Hongjun Choi, Zhongshu Gu†, Byoungyoung Lee, Xiangyu Zhang, Dongyan Xu

*NEC Laboratories America
†IBM Research
Security of Real-time Microcontrollers

• Safety-critical embedded and cyber-physical systems

• Security is often overlooked as a trade off

• Demand both real-time guarantee and security
Missing Memory Protection of RT Microcontrollers

• No process memory isolation
  • No MMU, no virtual memory
  • Memory space shared by all processes
  • Memory-mapped I/O (MMIO)
Control Parameter Attack
Missing Memory Protection of RT Microcontrollers

- No kernel memory isolation
  - Hardware and RTOS support
    - Privileged and unprivileged processor modes
    - Memory Protection Unit (MPU)
  - Many real-time microcontroller systems do not employ it
    - Verified with 67 commodity systems
    - Impact on real-time constraints
- Frequent mode switching

![Graph showing time in μsec (Log-scale) for different tasks]
Minion: Customized Memory View Enforcement

• Key ideas
  • Break physical memory space into per-process *memory views*
  • Use the memory views as *access control rules* during run-time
  • Execute RTOS and applications in the same mode (unprivileged)
  • Run a tiny *view switcher* in privileged mode to enforce views

![Diagram showing memory view enforcement process](image)
Memory View Tailoring

- **Memory view**: Memory required for a process to run correctly
- Find the physical memory regions **essential** for each process
- Static firmware analysis (LLVM IR)
- Code injection/reuse, data corruption, physical device abuse

For each process:

- **Code Reachability Analysis**
- **Data Accessibility Analysis**
- **Device Accessibility Analysis**

Access control rules:

<table>
<thead>
<tr>
<th>#</th>
<th>Base</th>
<th>Size</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Code Reachability Analysis

• Find all **reachable** functions from the entry functions

• **Entry functions**
  • Start function & interrupt handlers
  • Identified by analyzing a few RTOS functions

• Indirect calls?
  • **Inter-procedural points-to analysis**

• Build a list of executable memory regions for each process

<table>
<thead>
<tr>
<th>Value</th>
<th>PointsTo</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value X</td>
<td>PointsTo: { bar }</td>
<td></td>
</tr>
<tr>
<td>Value Y</td>
<td>PointsTo: { foo }</td>
<td></td>
</tr>
<tr>
<td>Value Z</td>
<td>PointsTo: { bar }</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Address Range</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>08004970-08004988</td>
<td></td>
</tr>
<tr>
<td>irq_handler</td>
<td>08088050-080880cc</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>0800498c-08004a7c</td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>08004a84-08004ad6</td>
<td></td>
</tr>
<tr>
<td>baz</td>
<td>08004ad8-08004b4c</td>
<td></td>
</tr>
</tbody>
</table>
Data Accessibility Analysis

• Global data
  • **Forward slicing** based on inter-procedural value flow graph
  • Build a list of global data for each process

• Stack and heap data
  • Memory pool **size profiling** with annotated memory allocator
  • **Per-process memory pool allocation**

- Global data:
  - LDR r8, GlobA
  - STR r2, GlobA

- Stack and heap data:
  - LDR r0, GlobB
  - STR r5, GlobB
  - LDR r2, GlobC

- Memory pool size profiling:
  - GlobA: 200010f0-200010f4, RW
  - GlobB: 20014618-20014638, RW
  - GlobC: 080b3428-080b3440, R
Device Accessibility Analysis

- A few patterns cover most MMIO operations
- MMIO addresses are **embedded** in the firmware
- Case 1
  ```c
  #define DEVICE_X 0x50000804
  void dev_reset(struct dev *priv) {
      uint32_t val;
      val = (1 << 2) | (1 << 4);
      *(uint32_t *)DEVICE_X = val;
      ...
  }
  ```

- Case 2
  ```c
  #define IRQ_A 1
  #define IRQ_B 2
  #define NVIC_A 0xe000e100
  #define NVIC_B 0xe000e104

  int irqinfo (int irq, uint32_t *addr) {
      if (irq == IRQ_A) {
          *addr = NVIC_A;
      } else if (irq == IRQ_B) {
          *addr = NVIC_B;
      ...
  }

  int enable_irq (int irq) {
      uint32_t addr, val;
      if (irqinfo(irq, &addr) == OK) {
          val = *(uint32_t *)addr;
          val |= (1 << 1);
          *(uint32_t *)addr = val;
      }
  }
  ```

From NuttX RTOS (simplified)
Device Accessibility Analysis

• Find load and store instructions with an MMIO address
• **Backward slicing** on inter-procedural value flow graph
• Build a list of **peripheral-mapped** memory regions for each process

![Diagram](image)

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE_X 50000804-50000808</td>
<td>W</td>
</tr>
<tr>
<td>NVIC_A e000e100-e000e104</td>
<td>RW</td>
</tr>
<tr>
<td>NVIC_B e000e104-e000e108</td>
<td>RW</td>
</tr>
</tbody>
</table>
Run-time Memory View Enforcement

<table>
<thead>
<tr>
<th>#</th>
<th>Base</th>
<th>Size</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RTOS

P1

P2

Unprivileged

Privileged

Process Switch

View Switcher

Re-Configure

Memory view

MPU
Evaluation with Attack Cases

- Tested on a commodity UAV
  - 3DR IRIS+
- Found 4 new vulnerabilities in the firmware (confirmed and fixed)
- 76% memory space reduction

- 8 realistic attack cases

<table>
<thead>
<tr>
<th>Name</th>
<th>Attack surface</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process termination</td>
<td>RTOS function</td>
<td>✓</td>
</tr>
<tr>
<td>Control parameter attack</td>
<td>Control parameter</td>
<td>✓</td>
</tr>
<tr>
<td>RC disturbance</td>
<td>RC configuration</td>
<td>✓</td>
</tr>
<tr>
<td>Servo operation</td>
<td>Driver function</td>
<td>✓</td>
</tr>
<tr>
<td>Soft timer attack</td>
<td>Hardware timer</td>
<td>✓</td>
</tr>
<tr>
<td>Hard timer attack</td>
<td>Hardware timer</td>
<td>✓</td>
</tr>
<tr>
<td>Memory remapping</td>
<td>Flash patch unit</td>
<td>✓</td>
</tr>
<tr>
<td>Interrupt vector overriding</td>
<td>Interrupt vector</td>
<td>✓</td>
</tr>
</tbody>
</table>

- All 8 attack cases blocked
- Zero violation of real-time constraints
Attack Under Minion’s Protection
Performance Impact

• 31 real-time tasks with deadlines: 2% overhead
• All deadline constraints satisfied
Conclusion

• Memory protection in RT microcontrollers

• **Minion**: New architecture to bring memory isolation to RT microcontroller systems

• Significant memory space reduction with maintained RT responsiveness

• Attack cases and vulnerability discovery
Thank you!
Questions?

https://github.com/chungkim/minion

Chung Hwan Kim <chungkim@nec-labs.com>

This work is sponsored by ONR.