Security Evaluation of MCUS Defenses

Naif Saleh Almakhdhub$^{1,3,4,5}$

Saurabh Bagchi$^{1,3}$ Mathias Payer$^{2,4}$
Current State of Security

Target:
Embedded systems and IoT
Running Microcontroller Systems (MCUS)

Attack:
Control-flow Hijacking

Internet of Things & MCUS

- The number of IoT devices is growing rapidly

- Many will be microcontroller based systems (IoT-MCUS)
  - Run single static binary image directly on the hardware
  - Can be with/without an OS (bare-metal)
  - Direct access to peripherals and processor
  - Small memory

- Examples:
  - WiFi System on Chip
  - Cyber-physical systems
  - UAVs
MCUS Challenges

Desktop

- ✓ Large virtual memory (GBs)
- ✓ Basic defenses (e.g., ASLR)
- ✓ DEP

MCUS

- × Small physical memory (MBs Flash, KBs RAM)
- × Basic defenses (e.g., ASLR)
- × DEP (Disabled → Fixable)

Memory

- Stack
- Heap
- Data
- Code

Flash (code)

- Code

RAM (Data)

- Stack
- Heap
- Data
Evaluation in Current MCUS Defenses

• Multiple defenses have been proposed
  • TyTan[DAC15], TrustLite[EurSys14], C-FLAT [CCS16], nesCheck[AsiaCCS17], SCFP[EuropS&P18], LiteHAX[ICCAD18], CFI CaRE [RAID17], ACES[SEC18], MINION [NDSS18], EPOXY [S&P17]

• How are they evaluated?
  • Ad-hoc evaluation

<table>
<thead>
<tr>
<th>Defense</th>
<th>Evaluation Type</th>
<th>Benchmark</th>
<th>Case Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>TyTan</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>TrustLite</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>C-FLAT</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>nesCheck</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SCFP</td>
<td></td>
<td>Dhrystone[1]</td>
<td>✓</td>
</tr>
<tr>
<td>LiteHAX</td>
<td></td>
<td>CoreMark[2]</td>
<td>✓</td>
</tr>
<tr>
<td>CFI CaRE</td>
<td></td>
<td>Dhrystone[1]</td>
<td>✓</td>
</tr>
<tr>
<td>ACES</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Minion</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>EPOXY</td>
<td></td>
<td>BEEBS[3]</td>
<td>✓</td>
</tr>
</tbody>
</table>

Open Issues in MCUS security

• **Realistic IoT benchmarks and evaluation framework**
  • Enable effective evaluation and comparison
  • Reduce burden on researchers

• **Strong defenses for control-flow hijacking**
  • Existing solutions are impractical or have limited security guarantees
BenchIoT: A Security Benchmark for The Internet of Things

(Published at DSN19)
IoT-MCUS Evaluation (Ideally)

1. Defense Mechanism A

2. Benchmark foo

A standardized software application

3. Evaluation Metrics
IoT-MCUS Evaluation (Reality)

1. Defense Mechanism A

2. Benchmark foo
   - Different benchmarks
   - Different Metrics

3. A’s Evaluation Metrics
   - Comparison is not feasible
   - Evaluation is limited and tedious

4. Defense Mechanism B

5. Benchmark bar

6. B’s Evaluation Metrics
Why not use existing benchmark?

- Current benchmarks are rigid and simplistic
  - Many are just one file with simple application
  - Metrics are limited and cumbersome to collect
  - Hardware dependent
- Do not use peripherals
- No network connectivity
Proposed Solution: BenchIoT

- BenchIoT provides a suite of benchmark applications and an evaluation framework

- A realistic set of IoT benchmarks
  - Mimics common IoT characteristics, e.g., tight coupling with sensors and actuators
  - Works for both with/without an OS

- Our evaluation framework is versatile and portable
  - A software based approach
  - Can collect metrics related to security and resource usage

- Targeted Architecture: ARMv7-M (Cortex-M3,4, and 7 processors)
## Comparison Between BenchIoT and Other Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Task Type</th>
<th>Network Connectivity</th>
<th>Peripherals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sense</td>
<td>Compute</td>
<td>Actuate</td>
</tr>
<tr>
<td>BEEBS [2]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dhrystone [1]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoreMark [3]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IoTMark [4]</td>
<td>✓</td>
<td>✓</td>
<td>Partially (Bluetooth only)</td>
</tr>
<tr>
<td>SecureMark [5]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BenchIoT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

BenchIoT: Overview

Compile & link

User Configuration files

Benchmark Binary

Can use a different benchmark

Evaluation Framework

Run benchmark on board

Collect dynamic metrics

Parse the benchmark binary

Collect static metrics

Metric collector runtime library

Results file

Benchmark Binary

Can use a different benchmark
BenchIoT Design Feature: (1) Hardware agnostic

- Applications often depend on the underlying vendor & board.
  - Memory is mapped differently on each board.
  - Peripherals are different across boards.

- For Operating systems:
  - Mbed OS(C++)

[Diagram showing layers: Hardware -> MCU Registers -> CMSIS (Cortex Microcontroller Software Interface Standard) -> HAL Library (Hardware Abstraction Layer) -> Mbed -> Application]

Portable

Vendor & board dependent

Hardware
BenchIoT Design Feature: (2) Reproducibility

• Applications are event driven
  • Example: User enters a pin
  • Problem: This is inconsistent (e.g., variable timing)

• Solution: Trigger interrupt from software
  • Creates deterministic timing
  • Allows controlling the benchmarking execution
BenchIoT Design Feature: (2) Reproducibility

Normal application

```c
/* Pseudocode */
1. void benchmark(void){
2.     do_some_computation();
3.     ...
4.     ...
5.     wait_for_user_input();
6.     read_user_input();
7.     ...
8. 
9. }
```

This is not deterministic

BenchIoT

```c
/* Pseudocode */
1. void benchmark(void){
2.     do_some_computation();
3.     ...
4.     ...
5.     trigger_interrupt();
6.     ...
7.     read_user_input();
8.     ...
9. 
10. }
```

Deterministic
BenchIoT Design Feature: (3) Metrics

• Allows for measurement of 4 classes of metrics: Security, performance, energy, and memory
BenchIoT Design Feature: (3) Metrics

- **Security**
  - Total privileged cycles
  - Privileged Thread cycles
  - SVC cycles
  - Max Data region ratio
  - Max Code region ratio
  - DEP
  - ROP resiliency
  - # of indirect calls

- **Performance & Energy**
  - Total execution cycles
  - CPU sleep cycles
  - Total energy

- **Memory**
  - Stack+Heap usage
  - Total RAM usage
  - Total Flash usage

- : Static metric
- : Dynamic metric
### Set of Benchmark Applications

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Task Type</th>
<th>Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sense</td>
<td>Compute</td>
</tr>
<tr>
<td>Smart Light</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Smart Thermostat</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Smart Locker</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Firmware Updater</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connected Display</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Boards without non-common peripherals can still run the benchmark**
BenchIoT Evaluation: Defense Mechanisms

**ARM’s Mbed-μVisor**

- Unprivileged
  - Application code
- Privileged
  - μVisor + OS
  - A hypervisor that enforces the principle of least privilege

**Remote Attestation (RA)**

- Verifies the integrity of the code present on the device
- Uses a real-time task that runs in a separate thread
- Isolates its code in a secure privileged region
- Hashed code block
- 25ms

**Data Integrity (DI)**

- Sensitive Data
  - Isolates sensitive data to a secure privileged region
  - Disables the secure region after the data is accessed
The goal is to demonstrate BenchIoT effectiveness in evaluation

- **Non-goal**: To propose a new defense mechanism

ARM’s Mbed-\(\mu\)Visor and Remote Attestation (RA) require an OS

Data Integrity (DI) is applicable to Bare-Metal (BM) and OS benchmarks
BenchIoT Evaluation: Defense Mechanisms

- Comparable
- Evaluation is automated and extensible

**ARM’s Mbed-µVisor**

**Remote Attestation (RA)**

**Data Integrity (DI)**

BenchIoT Benchmarks

BenchIoT Evaluation Framework

**ARM’s Mbed-µVisor Evaluation**

**RA Evaluation**

**DI Evaluation**
Performance Results

Number of cycles in (Billions/Millions)

Evaluated without the display peripheral
Privileged Execution Minimization Results

- **Overhead as % of the insecure baseline application**

  - Almost the entire application runs as privileged for all defenses except uVisor.
  
  - uVisor is the most effective defense in reducing privileged execution.
# Code Injection Evaluation

<table>
<thead>
<tr>
<th>Defense</th>
<th>Data Execution Prevention (DEP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mbed-uVisor</td>
<td>✗ (Heap)</td>
</tr>
<tr>
<td>Remote Attestation (OS)</td>
<td>✓</td>
</tr>
<tr>
<td>Data Integrity (OS)</td>
<td>✗</td>
</tr>
<tr>
<td>Data Integrity (Bare-metal)</td>
<td>✗</td>
</tr>
</tbody>
</table>
Energy Consumption Results

Overhead as % over baseline

All defenses had modest runtime overhead

uVisor had no sleep cycles ≈ 20% energy overhead

26
Measurement Overhead

Average Overhead → 1.2%

Percentage of total execution cycles
BenchIoT: Summary

• Benchmark suite of five realistic IoT applications.
  • Demonstrates network connectivity, sense, compute, and actuate characteristics.
  • Applies to systems with/without an OS.

• Evaluation framework:
  • Covers security, performance, memory usage, and energy consumption.
  • Automated and extensible.

• Evaluation insights:
  • Defenses can have similar runtime overhead, but a large difference in energy consumption.

• Open source:
  • https://github.com/embedded-sec/BenchIoT
Open Issues in MCUS security

• Realistic IoT benchmarks and evaluation framework
  • Enable effective evaluation and comparison
  • Reduce burden on researchers

• Strong defenses for control-flow hijacking
  • Existing solutions are impractical or have limited security guarantees
μRAI: Securing Embedded Systems with Return Address Integrity

(Published at NDSS20)
MCUS Defenses for Return Addresses (Conceptual)

Return Address Integrity + Low runtime overhead + No special hardware

Usage

Location

Runtime Overhead

10%

Limited Security Guarantees

Special hardware required
Without extra hardware

Safe Stack +
Software Fault
Isolation

Randomized Safe Stack

Shadow Stack +
TEE

μRAI

CFI

Shadow Stack +
MPU

Security

Integrity

High Overhead
MCUS Defenses for Return Addresses (Related Work)

- **Components:**
  - **Return Address Integrity + Low runtime overhead + No special hardware**
  - **Limited Security Guarantees**
  - **CFI CaRE (Shadow stack) [RAID17]**
  - **μRAI**

**Overview:**
- Special hardware required
- Without extra hardware

**Usage**
- C-FLAT [CCS16]
- Minion [NDSS18]
- LiteHAX [ICCAD18]
- μXOM [SEC19]
- EPOXY (SafeStack) [S&P17]
- μArmor [EuroS&P19]

**Location**
- RECFISH [ECRTS 2019]

**Runtime Overhead**
- 10%
- High Overhead
Return Address Integrity (RAI)

• Every attack requires corrupting a return addresses by *overwriting* it

• Main limitation of defenses → return addresses are in *writable memory*
  • Example: Information hiding

• Key solution is to *prevent an attacker from corrupting return addresses*.

RAI Property:

• Ensure the return address is never writable except by an authorized instruction.
• Return addresses are never pushed to the stack or any writable memory by an adversary.
Threat Model & $\mu$RAI Protection

Normal application

Unprivileged

• Func1
• Func2
• Func3

Privileged

• Func4
• Func5
• Func6
• Func7

$\mu$RAI

• main
• Func1
• Func2
• Func3
• Func4
• Func5
• Func6
• Func7

$\mu$RAI Protection

- Reads from memory
- Writes to memory
- Knows the code layout
- Targets backward-edges

Corrupt return address

Corrupt return address or corrupt sensitive Memory Mapped IO (MMIO)

: Normal function
: Callable within exception handler
: MMIO
: State register encoding
: Software-Fault Isolation (SFI)
Evaluation

- Five MCUS applications on Cortex-M4:
  - PinLock
  - FatFs_uSD
  - FatFs_RAM
  - LCD_uSD
  - Animation

- CoreMark benchmark[1]
  - Standard MCUS performance benchmark

Integrating BenchIoT with μRAI

Evaluation Framework

1. Run benchmark on board
2. Collect dynamic metrics
3. Parse the benchmark binary
4. Collect static metrics
5. Parse the benchmark binary
6. Collect static metrics

User Configuration files

μRAI Binary

Compile & link

μRAI Benchmark

Results file

Not used

Metric collector runtime library

Used μRAI benchmark
Security Evaluation Using PinLock: Unlock The Lock

**Buffer overflow**
- RW Memory
- buffer
- No return address to overflow...

**Arbitrary write**
- RW Memory
- ... ptr ...
- No return address to write to...

**Stack pivot**
- RW Memory
- SP
- SP
- ... buff ...
- No return address to pop from the stack

**State Register** + **Jump Table**
- Jump return_location1
- ...
Security Evaluation Using PinLock: Unlock The Lock

**Buffer overflow**
- No return address to overflow...

**Arbitrary write**
- No return address to write to...

**Stack pivot**
- No return address to pop from the stack

✓ μRAI prevents all control-flow hijacking attack scenarios targeting return addresses

- State Register
- Jump Table
  - Jump return_location1
  - ...
- RX Memory
- Privileged Sensitive MMIO and Safe region
Performance results

- Requiring full-SFI results in high overhead $\rightarrow$ **average of 130.5%**
- $\mu$RAI results in low overhead $\rightarrow$ **average of 0.1%**
μRAI: Enforce RAI for exception handlers

• Exception handlers execute with privileges
  • Can disable the MPU → enable code injection
  • Can corrupt exception stack frame → break RAI property

• Solution:
  • Apply SFI only to functions callable by exception handlers
  • Limit SFI overhead compared to full-SFI

How to dynamically measure the protected store instructions?
μRAI’s Overview

- Call Graph Analyzer
  - Generates caller/callees for each function
- Encoder
  - Generates function keys and encodes SR
- Instrumentation

LLVM IR

- Added a special system call before each store instruction protected with SFI
- Special system call increments a counter
- Used a special binary for the measurement

Source code
## Store Instructions Protected with EH-SFI

<table>
<thead>
<tr>
<th>App</th>
<th># of Store instruction</th>
<th>Static</th>
<th>Total</th>
<th>(Static/Total)%</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PinLock</td>
<td></td>
<td>56</td>
<td>516</td>
<td>10.9</td>
<td>7</td>
</tr>
<tr>
<td>FatFs_uSD</td>
<td></td>
<td>99</td>
<td>1,802</td>
<td>5.5</td>
<td>906K</td>
</tr>
<tr>
<td>FatFs_RAM</td>
<td></td>
<td>7</td>
<td>1,116</td>
<td>0.6</td>
<td>7</td>
</tr>
<tr>
<td>LCD_uSD</td>
<td></td>
<td>99</td>
<td>2,814</td>
<td>3.5</td>
<td>48K</td>
</tr>
<tr>
<td>Animation</td>
<td></td>
<td>99</td>
<td>2,760</td>
<td>3.6</td>
<td>66K</td>
</tr>
<tr>
<td>CoreMark</td>
<td></td>
<td>56</td>
<td>1,024</td>
<td>5.5</td>
<td>7</td>
</tr>
</tbody>
</table>
μRAI: Conclusion

• Control-flow hijacking on MCUS is a threat

• μRAI secures MCUS against control-flow hijacking
  • Enforces the RAI property for MCUS → protects backward edges
  • Complemented with type-based CFI → end-to-end code pointer protection

• Presents a portable encoding scheme
  • Does not require special hardware features (only a register and an MPU)
  • Applicable to other systems

• Low runtime overhead

https://github.com/embedded-sec/uRAI
Conclusion & Discussion

• BenchIoT is a first step to enable security evaluation on MCUS

• Discussion and future direction
  • Usability
  • Porting to different RTOSs
  • Supporting different compilers
  • ...

• BenchIoT:
  
  https://github.com/embedded-sec/BenchIoT

• μRAI:
  
  https://github.com/embedded-sec/uRAI
Discussion