Microcontrollers Deserve Protection Too

Amit Levy

with:
Michael Andersen, Tom Bauer, Sergio Benitez, Bradford Campbell, David Culler, Prabal Dutta, Philip Levis, Pat Pannuto, Laurynas Riliskis
Microcontrollers
Microcontrollers

• Tightly integrated hardware
  – A single IC computer (memory, CPU, I/O peripherals)

• Typically:
  – Small amount of code memory (<= 512KB flash)
  – Small amount of RAM (1 – 128KB)
  – Low speed CPU (<<< 80MHz)
  – Low power consumption (μA sleep currents)
  – Little/no hardware support for isolation
Tock*: An Operating System for Microcontrollers

- Designed for multi-programming microcontrollers
- Strict application/kernel boundary
- Written in Rust
  - Safe but low-level programming language
  - Allows compile guarantees on contributed kernel code
- Leverage new hardware features, e.g.
  - Memory Protection Unit (MPU)
  - Faster processor speeds
  - DMA, many more I/O peripherals, etc...

* Codename
Today's Microcontroller “Operating Systems”

• “Operating System”
  - Hardware abstraction layer
  - Libraries for complex/common tasks
    - e.g. 6lowpan, Bluetooth, virtual timers etc'
  - No isolation
Outline

• Why?
  – New use cases
  – New developers
  – New hardware

• What?
  – Untrusted application sandboxing
  – Protection from drivers/kernel modules

• How?
  – Hardware sandbox for apps
  – Language-level sandbox for drivers
  – Zero dynamic allocation in the kernel
Outline

• Why?
  – New use cases
  – New developers
  – New hardware

• What?
  – Untrusted application sandboxing
  – Protection from drivers/kernel modules

• How?
  – Hardware sandbox for apps
  – Language-level sandbox for drivers
  – Zero dynamic allocation in the kernel
New Use Cases

- Ubiquitous computing
  - Fitness bands, medical devices, “smart home”
- Programmable platforms
  - Smart watches
  - Drones
  - Undoubtedly more to come
New Developers

Cost + tools make hardware development more accessible

- Small teams/startups
  - Iterative development process
  - Rapid deployment
- Hobbyists
  - One off applications, modding
- Not necessarily embedded systems experts
Old Hardware - telosb

- Based on MSP430
  - 16-bit
  - 8Mhz, 10Kb RAM, 48Kb code
- 802.15.4 radio
- Power draw
  - 5.1 μA idle
  - 1.8mA idle
New Hardware

- Based on ARM Cortex-M
  - 32-bit
  - 40Mhz, 64Kb RAM, 128Kb code
- 802.15.4 and Bluetooth radios
- Power draw
  - 2.3-13.0 μA idle
  - 8 mA active
  - < 25% on realistic workloads
- Many more peripherals:
  - USB, several USARTS, SPIs and I2Cs, AES accelerator…
- Memory Protection Unit (MPU)
Why a new Operating System?

• New Use Cases
  – Software updates on my medical device
  – Third-party apps

• New Developers
  – Non expert developers building highly personal/sensitive products

• New Hardware
  – Different power profile → different tradeoffs
  – Some hardware support for multi-programming
Why a new Operating System?

THINK QUALITY!
DO IT RIGHT
THE FIRST TIME
WE DON’T ALWAYS GET
A SECOND CHANCE
Outline

● Why?
  - New use cases
  - New developers
  - New hardware

● What?
  - Untrusted application sandboxing
  - Protection from contributed drivers/kernel modules

● How?
  - Hardware sandbox for apps
  - Language-level sandbox for drivers
  - Zero dynamic allocation in the kernel
Third-party apps

- Dynamically loadable
- May run concurrently
- Example: Pebble watch
  - 3rd party app ecosystem
    - E.g. pedometer, 2-factor auth, weather info
    - Must protect sensor data as well as other app data
- Potentially malicious threat model
- Need to *sandbox* against arbitrary behavior
Contributed Drivers

- E.g. storage system, network stack, LCD screen, sensors
- Written by the product/platform developer or non-core kernel developers
- Need low-latency access to hardware
  - Bitbanging devices, latency sensitive network stacks etc
- Not modeled as malicious, but potentially buggy
  - Shouldn't bring down the system
- “If I upgrade this flash driver, will my glucose monitor give me bad results?”
- Compiled into the kernel
  - Need compile-time guarantees
What Protections does Tock Provide?

• Applications:
  – Isolated from each other and from the kernel
  – A buggy application cannot bring down the rest of the system

• Drivers:
  – Can reason about behavior at compile time
  – (Relatively) easy to write non-buggy code with help from the compiler
  – Buggy driver cannot interfere with other critical code
  – Model ownership of hardware resources explicitly
Outline

• Why?
  – New use cases
  – New developers
  – New hardware

• What?
  – Untrusted application sandboxing
  – Protection from drivers/kernel modules

• How?
  – Hardware sandbox for apps
  – Language-level sandbox for drivers
  – Zero dynamic allocation in the kernel
Tock: Overview

- Hardware separation between apps and kernel (MPU)
- Kernel written in safe language (Rust), apps written in any language
  - C, Lua, Rust, etc
  - Helps balance safety with low-level access and ease of use.
- Drivers written in language-level sandbox
  - Unique and exclusive access to underlying hardware
Tock: Architecture

Hardware Sandbox

Syscall Interface

Rust

Device Drivers
(Rust language sandbox)

Kernel Core

App1.c

App1.lua

App1.rs
MPU: Hardware App Sandbox

• Enforce read/write/execute on applications for different portions of memory
• No virtual addressing but much finer grained than MMU
• On Cortex-M4:
  – Up to 64 different regions
  – Region size between 32B and 4KB
• Can isolate application memory from each other
• Can allocate sensitive kernel data structures in “application space”
• Can expose specific peripherals directly to applications
MPU: Hardware App Sandbox

Memory Mapped I/O

Kernel Stack

Second App Memory

First App Memory

App code

Peripheral

high address

low address

App specific
Kernel memory
Language-level Sanbox: Goals

- Device drivers cannot interfere with each other
  - E.g. a network stack cannot muck with readings from a glucose sensor

- Single threaded execution model
  - Simpler to write correct code
  - Kernel/hardware does not have to worry about concurrent access bugs
  - Much faster processing speeds make this feasible within time contraints
Language-level Sanbox: Why Rust?

• No runtime system
  – Not garbage collected, zero-cost safety abstractions

• Memory Safety
  – Eliminates a large class of bugs: dangling pointers, double-frees, pointer arithmetic errors, etc

• Type Safety
  – Can expose low-level hardware interfaces through safe interfaces

• Strict Aliasing
  – Unique references and read/write references obviate many concurrency bugs
Zero Dynamic Kernel Allocation

• Embedded OSs generally avoid dynamic allocation for good reason
  – Hard to determine in the lab if something will crash in the field
  – No swapping, so no way to deal with memory overflows
• But problematic with dynamically loaded applications
  – A new app may use drivers differently
  – E.g. different number of timers, more buffer, etc
Tock: Dynamic Allocation

Three ways kernel allocates memory:

• Statically
  – Size determined at compile-time

• Kernel stack
  – Maximum size can be determined at compile-time via analysis

• Application memory
  – Fined grained MPU allows dynamic sizing of app-specific kernel-heap.
Tock: Dynamic Allocation

Example kernel allocations in application space:

- Linked list nodes
- Virtual timer structs
- Network stack buffers
Summary

● Traditional embedded systems are outdated:
  − New hardware
  − New use cases
  − New generation of developers

● Security should be a main goal of any new system

● Leverage hardware protection to isolate third-party applications

● Leverage advances in programming languages to make kernel more secure
Challenges & Questions

- Will this work?
  - Maintain reliability and power constraints unique to embedded devices
  - While making security accessible to non-expert developers

- Dynamically updating the kernel/drivers?

- How do we leverage multi-microcontroller platforms?

- MGC security - Applications that span embedded devices, gateways (e.g. smartphones) and the cloud