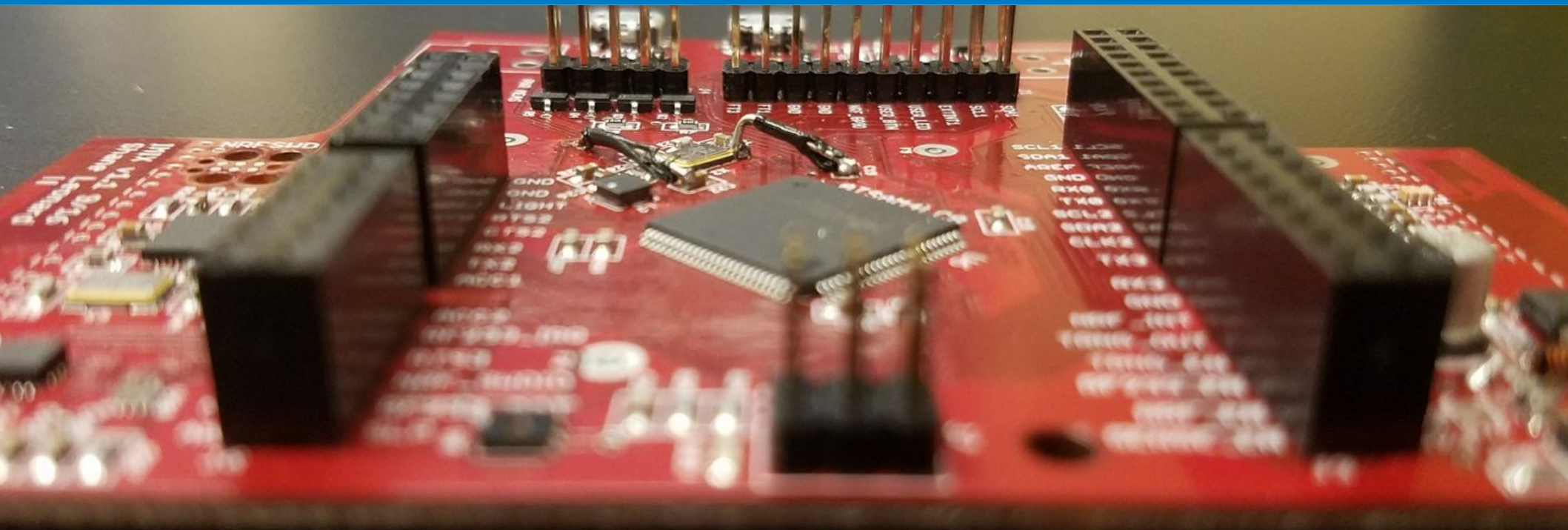
Amit Levy

SITP Retreat,

June 22, 2018

Tock: A Secure Operating System for Microcontrollers



Limitations of Microcontroller Software

- Low memory: ~64 kB RAM
- No virtual memory
- Can’t use Linux!
- No isolation

TinyOS freeRTOS

Contiki

The Open Source OS for the Internet of Things

- USB authentication keys have multiple functions
- Sensor networks run several experiments at once
- Fitness watches support different activities

Multi-User Device: Signpost

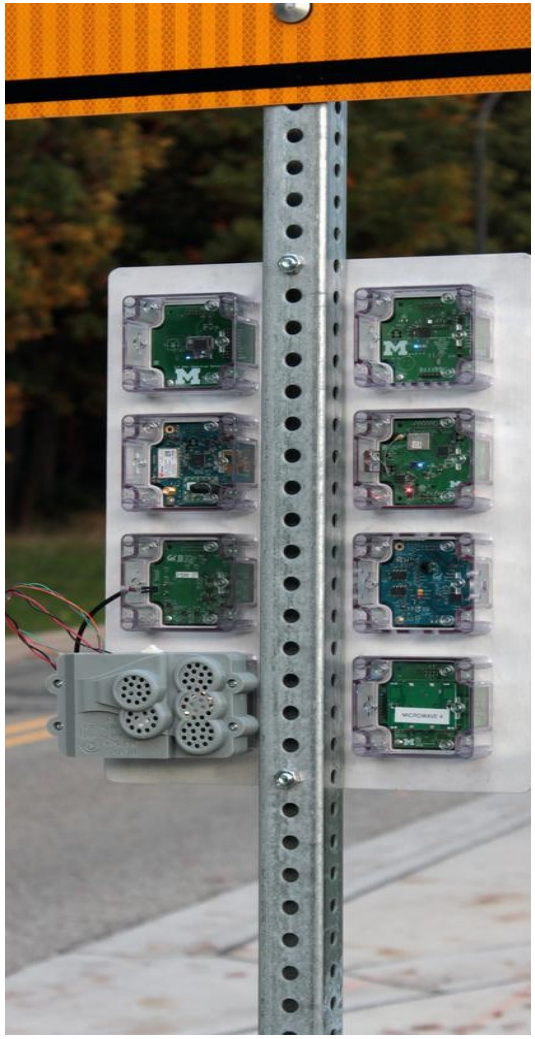
Modular city-scale sensing

- Tracking Ambient conditions
- Pedestrian density
- Noise monitoring

8 pluggable modules

- Microcontroller + Sensors

Several applications per module

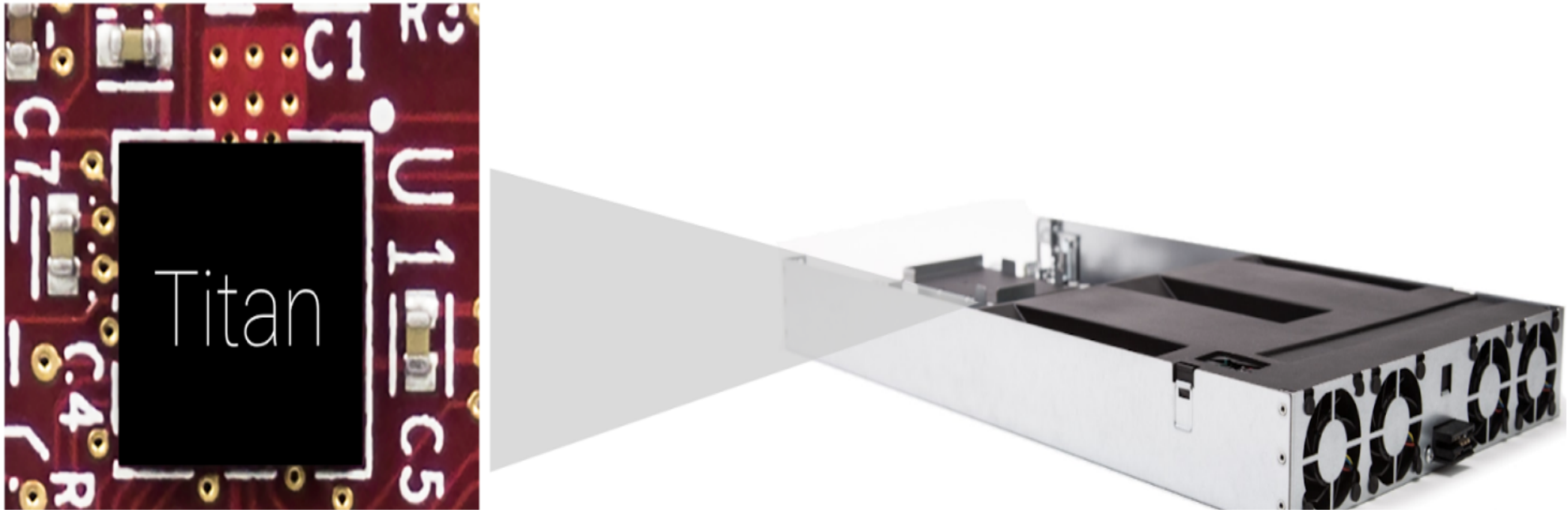


Currently deployed @ U.C. Berkeley

OTA Updates: Helium

- End-to-end IoT “solution”
- Programmable module
  - Long-range radio
  - MCU runs customer services + applications
- Abandoned a previous version that used Lua
- Next version uses Tock

Security Sensitive Device: Google Titan



- Google's Titan chip: security hardened MCU
- Server root-of-trust, authentication
- Without Tock: a handful of experts audit all code
- Open source port of Tock started during internship

Thesis: Embedded Devices are Multiprogrammed

- Multiple users running applications concurrently
- Applications updated dynamically
  - Small payloads better
  - Buggy updates shouldn’t brick devices
- Security sensitive devices want *least privilege*

Tock Embedded OS

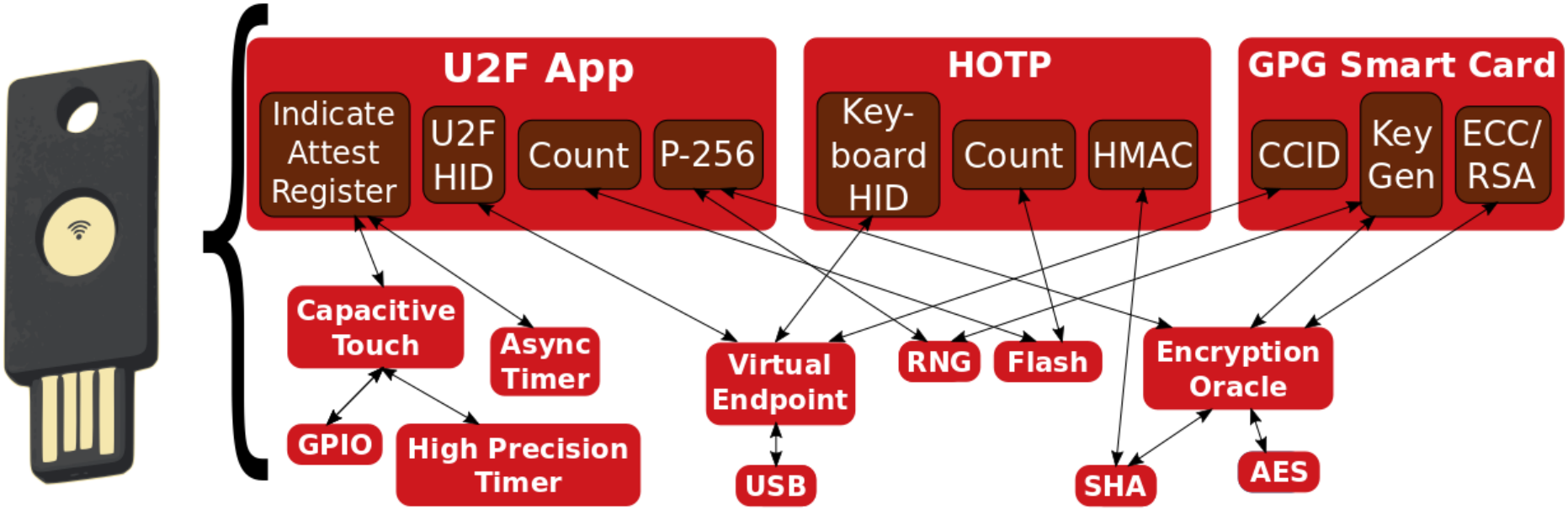
- Growing open-source community
  - 883 GitHub followers, >100 mailing list subscribers
  - 54 contributors (so far) to main project
  - ~20 contributors to out-of-tree HW ports
  - >100 developers trained at Tock tutorials
- Growing HW support
  - ARM Cortex-Ms: Atmel SAM4L, Nordic NRF5x, TI CC26xx & TM4C129x, NXP MK66, STM32
  - RISC-V port at a secret facility outside Boston

Multiprograming a Microcontroller

1. Memory & Performance Isolation
2. Power & Clock Control
3. Peripheral communication busses
4. Future work

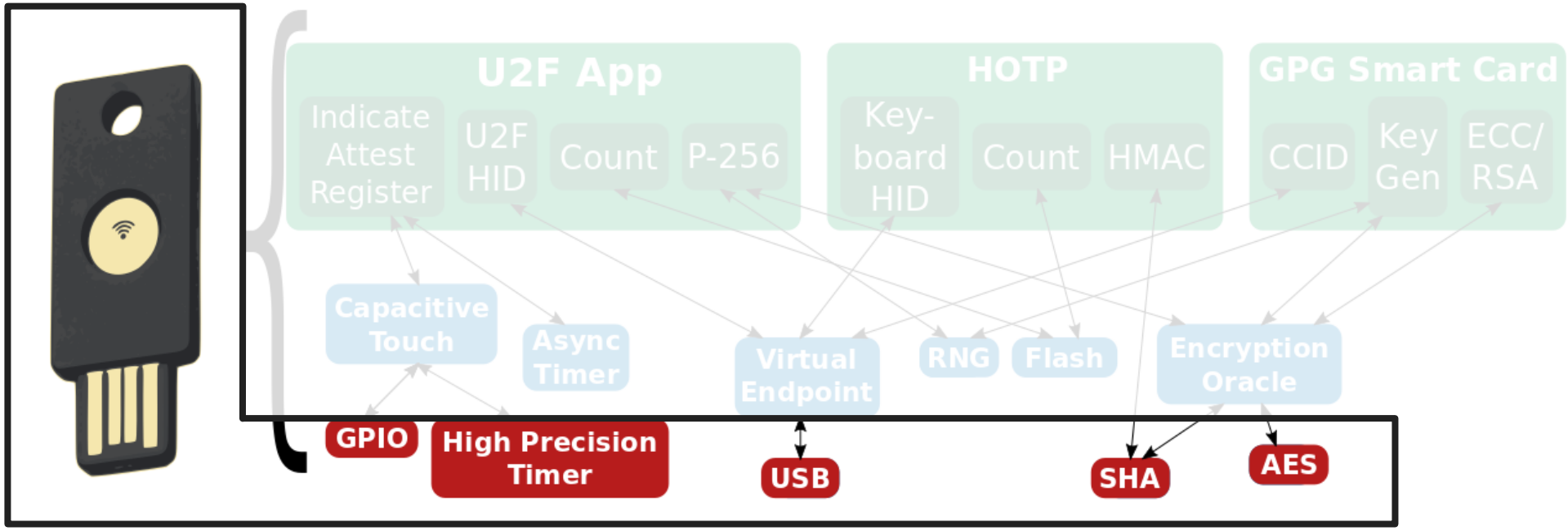
Example: USB Authentication Key

- Multiple independent applications
- No programmability in favor of security
- Result: Handful of programmers control software stack



Platform (~10 developers)

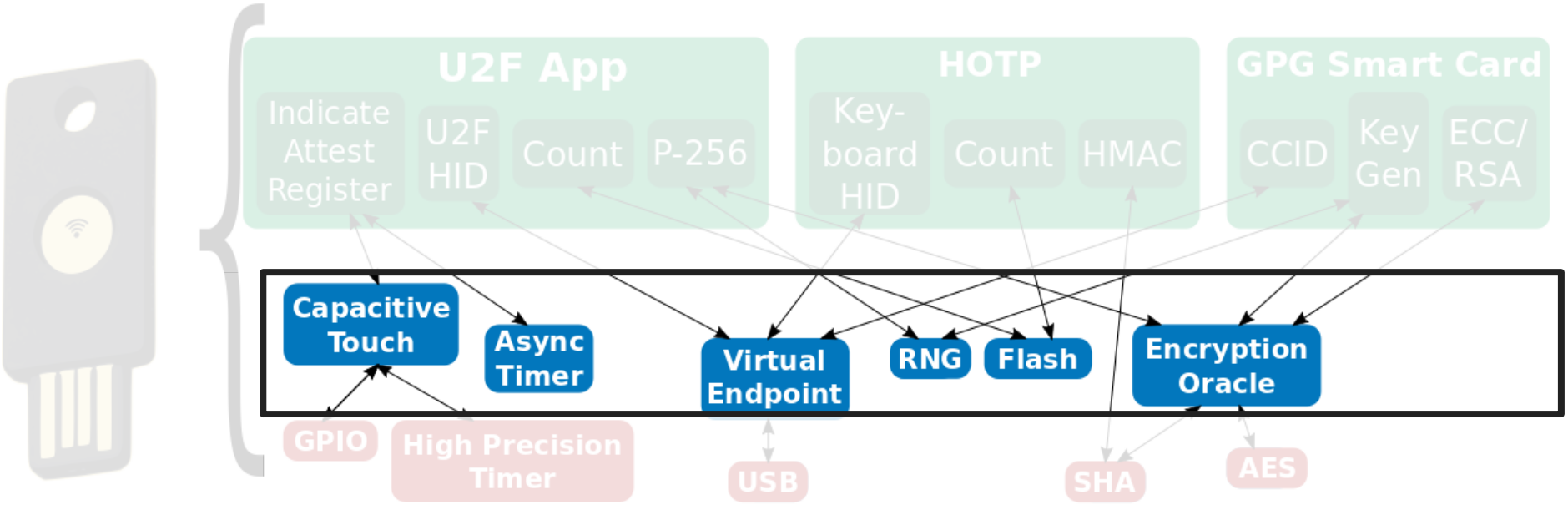
- Build the hardware
- Responsible for TCB: core kernel, MCU-specific code
- Trusted: complete control over firmware & hardware



Goal: possible to correctly extend TCB

OS Services (~1000 developers)

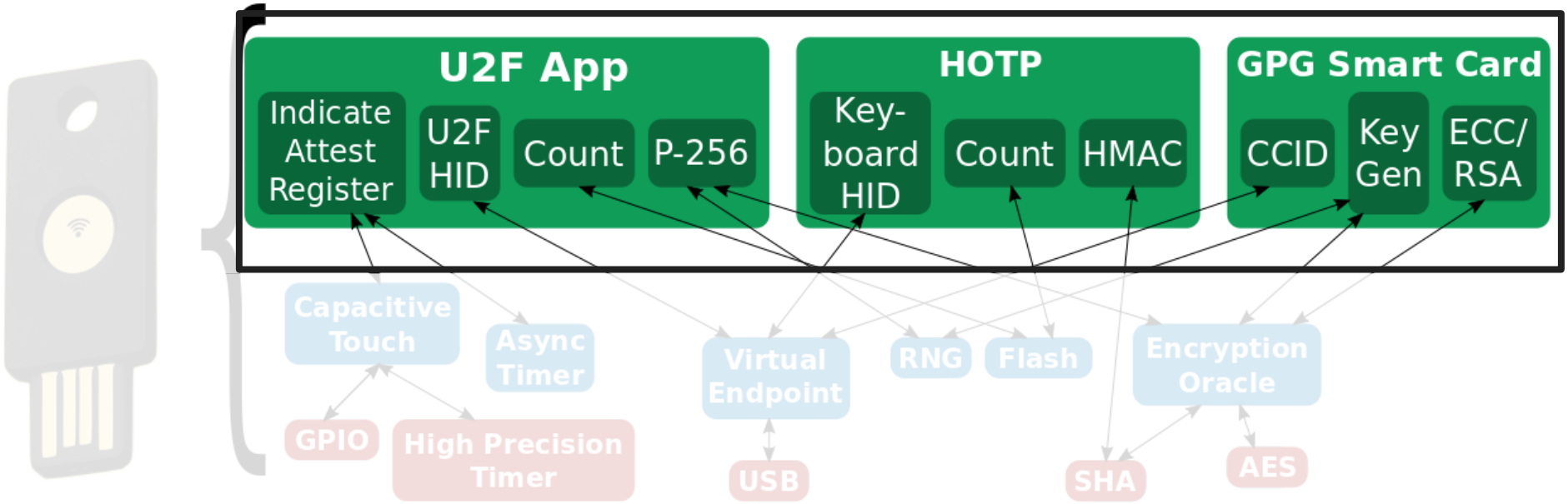
- Most OS services come community
  - Device drivers, networking protocols, timers...
- Platform provider can audit but won't catch all bugs



Goal: protect kernel from safety violations

Applications (~20M developers)

- Implement end-user functionality
- "Third-party" developers: unknown to platform provider
- Potentially *malicious*



Goal: end-users can install 3rd-party apps

Need New Isolation Techniques

- With 64 kB, `malloc` a serious threat to system stability
- No virtual memory
- Still need to solve driver isolation
  - GC'd languages & hardware isolation too resource heavy

New Tools Available

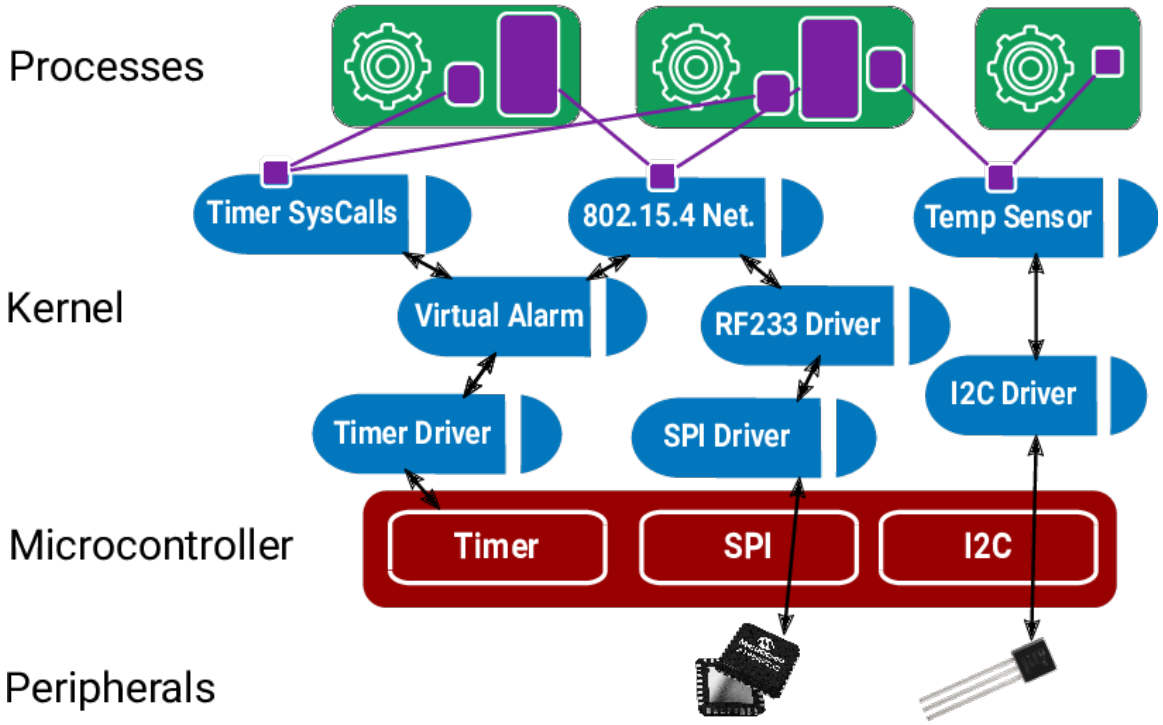
Memory Protection Unit (MPU)

- Protection bits for 8 memory regions
- Isolate processes for applications

Rust

- Non-GC'd type-safe systems language
- Prevent safety violations in kernel at very low cost

Tock: A Microcontroller OS



- **Processes**: Use the Memory Protection Unit
- **Capsules**: Type-safe Rust API for *safe* driver development
- **Grants**: Bind dynamic kernel resources to process lifetime

Tock's Isolation Mechanisms

Totally untrusted

Processes

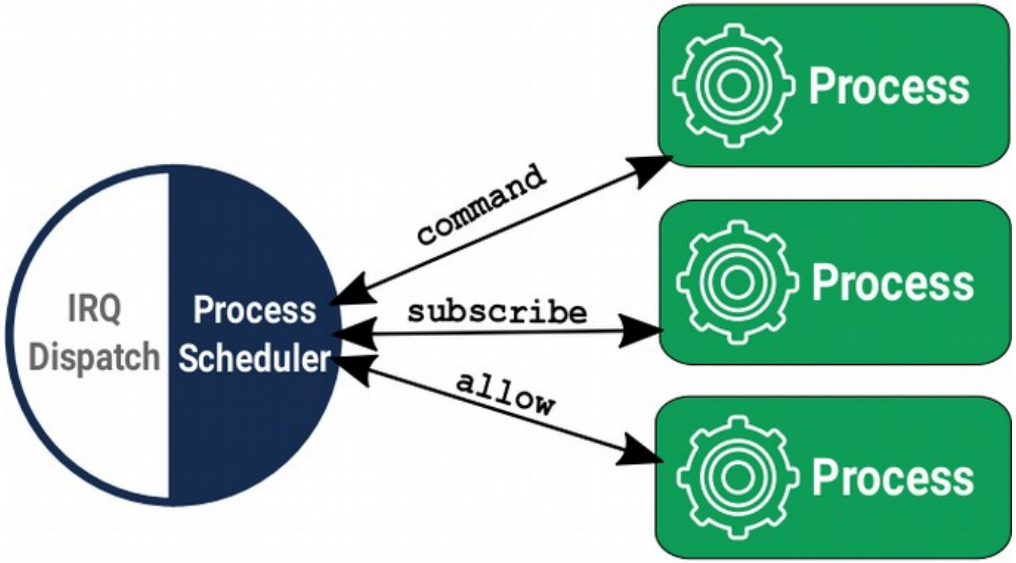
- Standalone executable in any language
- Isolation enforced at runtime
- Higher overhead
- Applications

Trusted for liveness, not safety

Capsules

- Rust code linked into kernel
- Isolation enforced at compile-time
- Lower overhead
- Used for device drivers, protocols, timers...

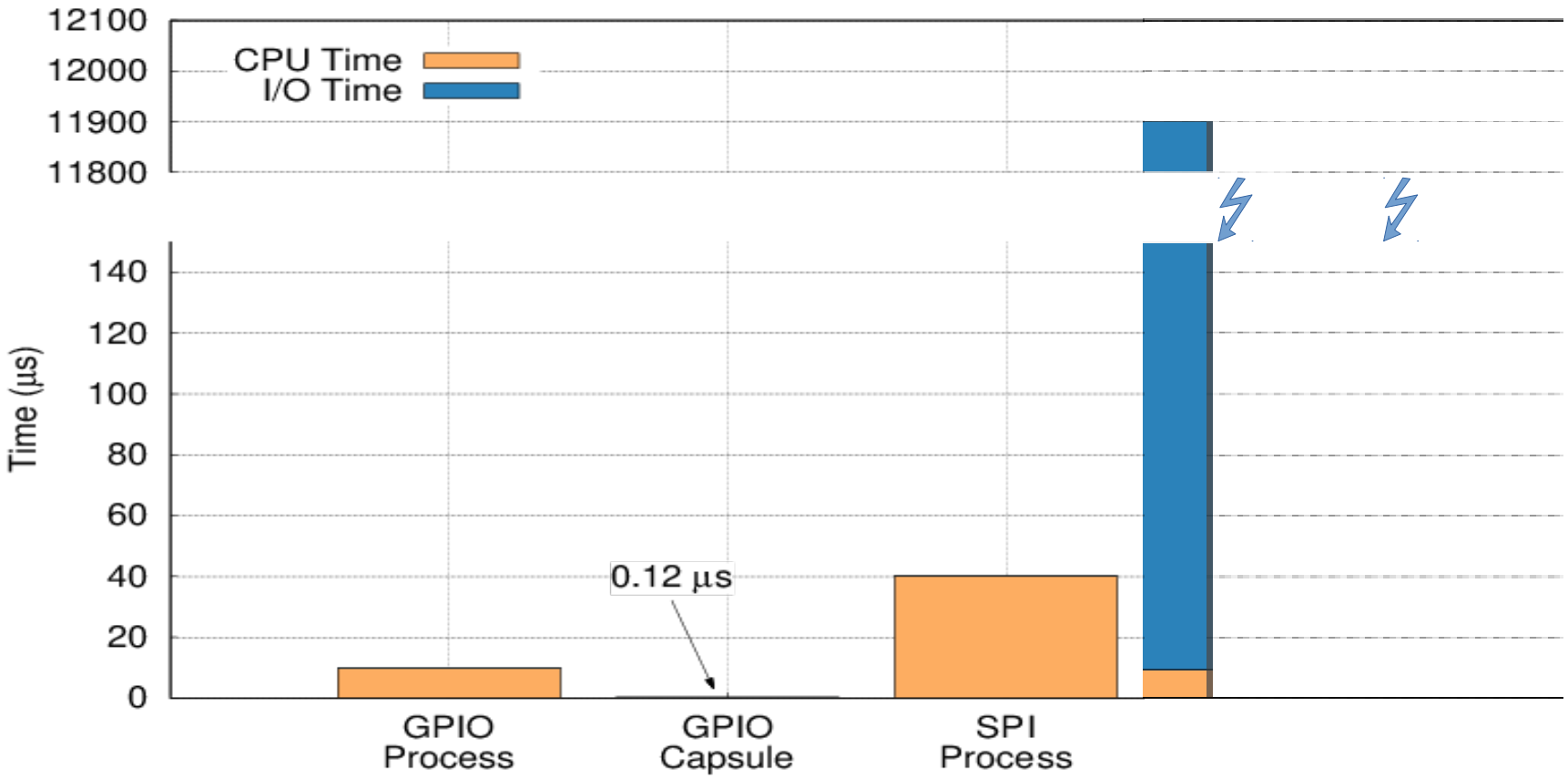
Processes



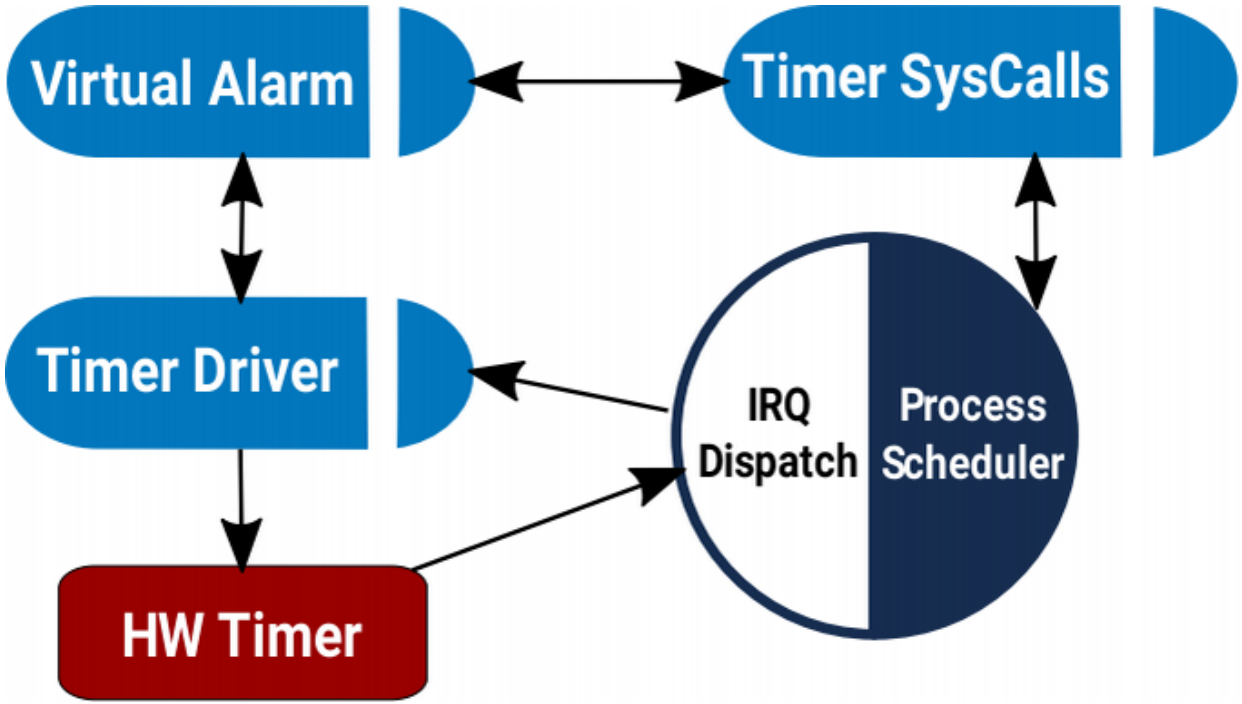
- Hardware-isolated concurrent programs in *any* language
  - MPU to protect memory regions without virtualization
  - Independent stack, heap, static variables
- Run dynamically, compiled with position independent code
- Scheduled preemptively
- System calls & IPC for communication

Process Overhead

- Dedicated memory region (at least a stack)
- Context switch for communication (340 cycles)



Capsules



- A Rust module and structs
- Single-threaded event-loop with asynchronous I/O
- Single stack
- No heap
- Communicate via references & method calls, often inlined

Capsule Resource Overhead

Example 1: “blink”

	ROM size (B)	RAM size (B)
Tock	3208	916
TinyOS	5296	72

Example 2: Networked sensor

	ROM size (B)	RAM size (B)
Tock	41744	9704
TinyOS	39604	10460

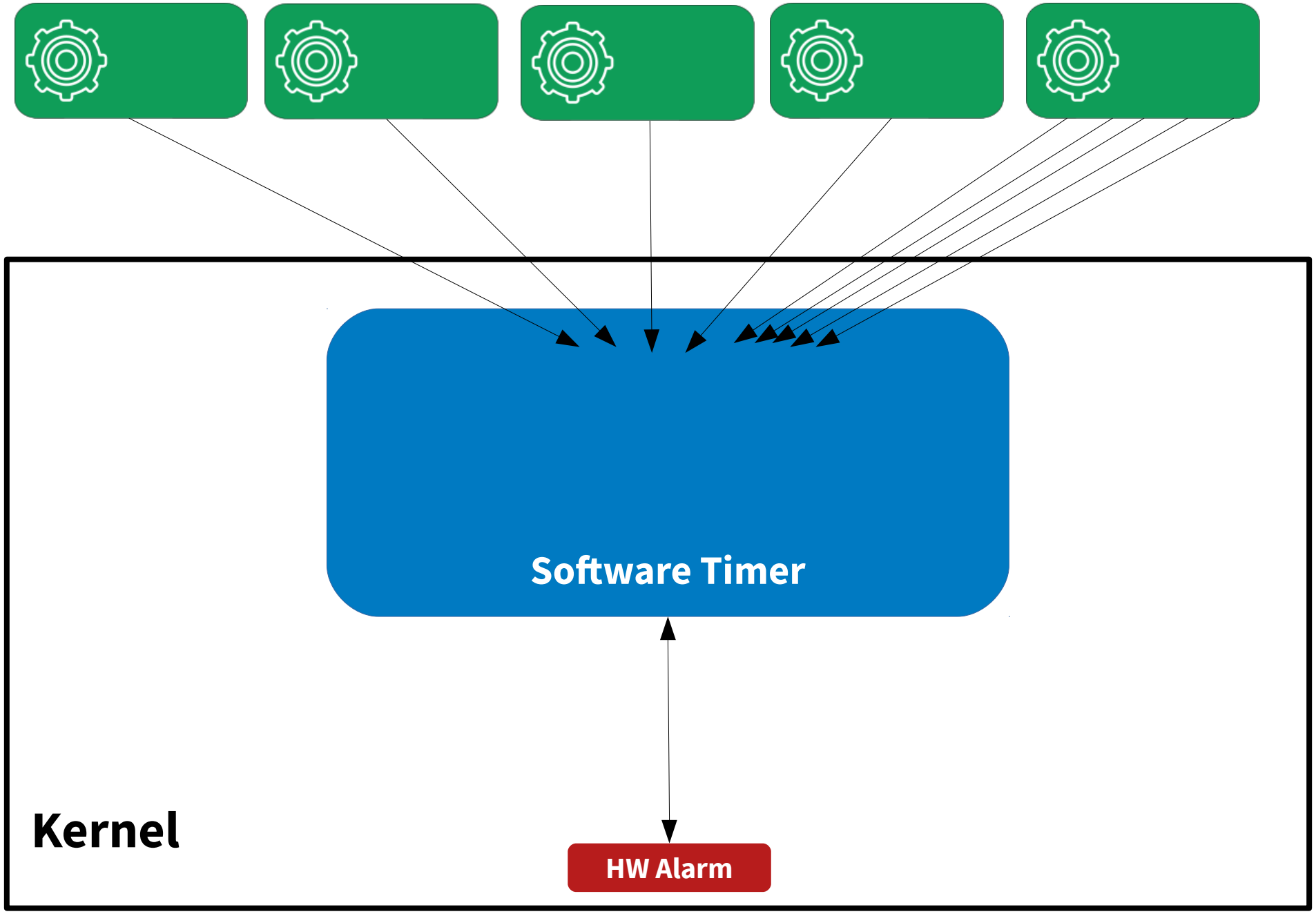
Capsule Isolation

```
struct DMAChannel {
    length: u32,
    base_ptr: *const u8,
}

impl DMAChannel {
  fn send_buffer(&self, buf: &'static [u8]) {
    self.length = buf.len();
    self.base_ptr = buf.as_ref();
  }
}
```

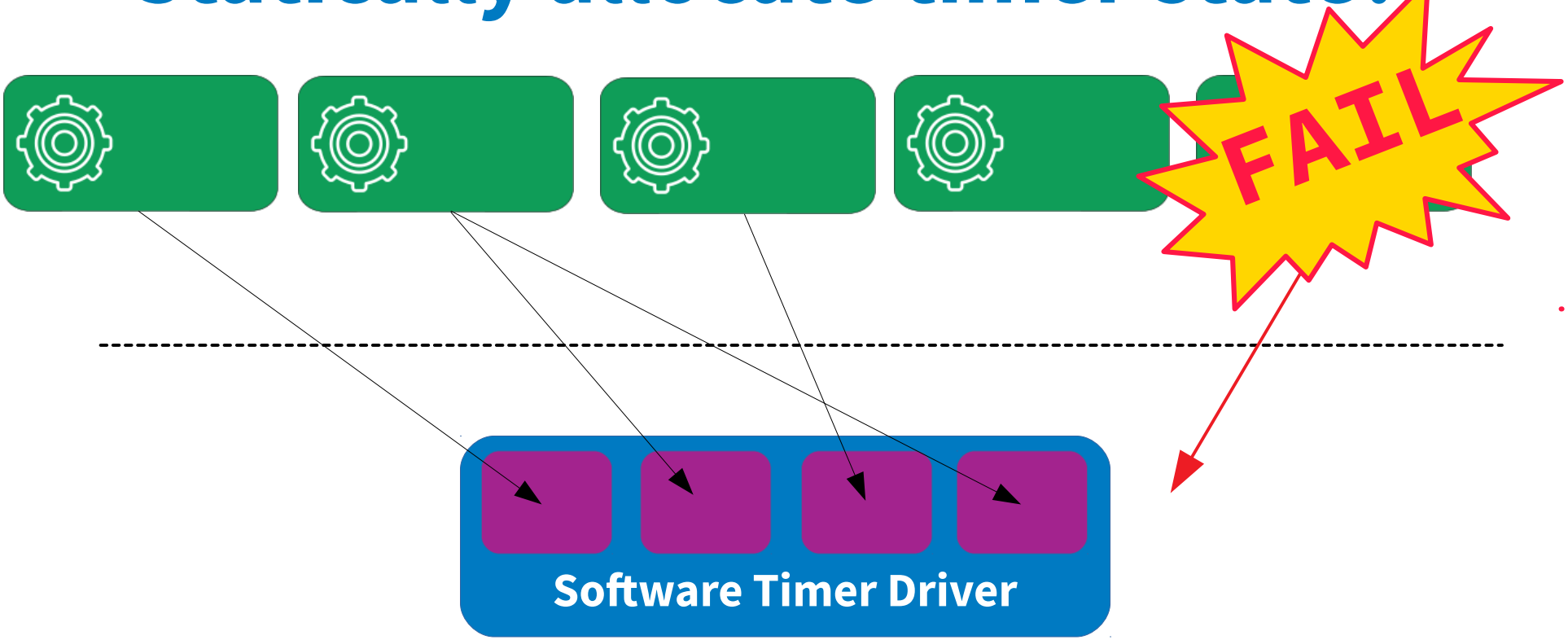
- Exposes the DMA base pointer and length as a Rust *slice*
- Type-system guarantees user has access to memory
- Won't be deallocated before DMA completes

Safe Dynamic Kernel Allocation

Working Example: Software Timer

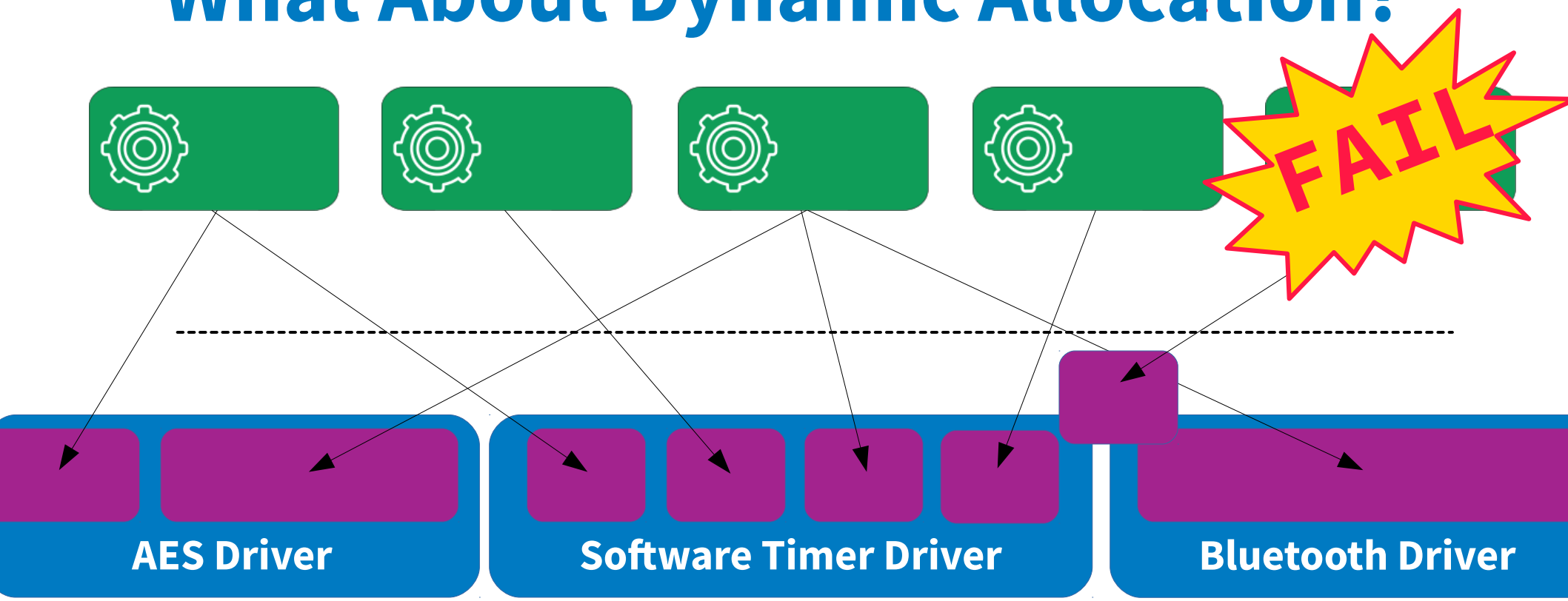


Statically allocate timer state?



Static allocation must trade off memory efficiency and maximum concurrency

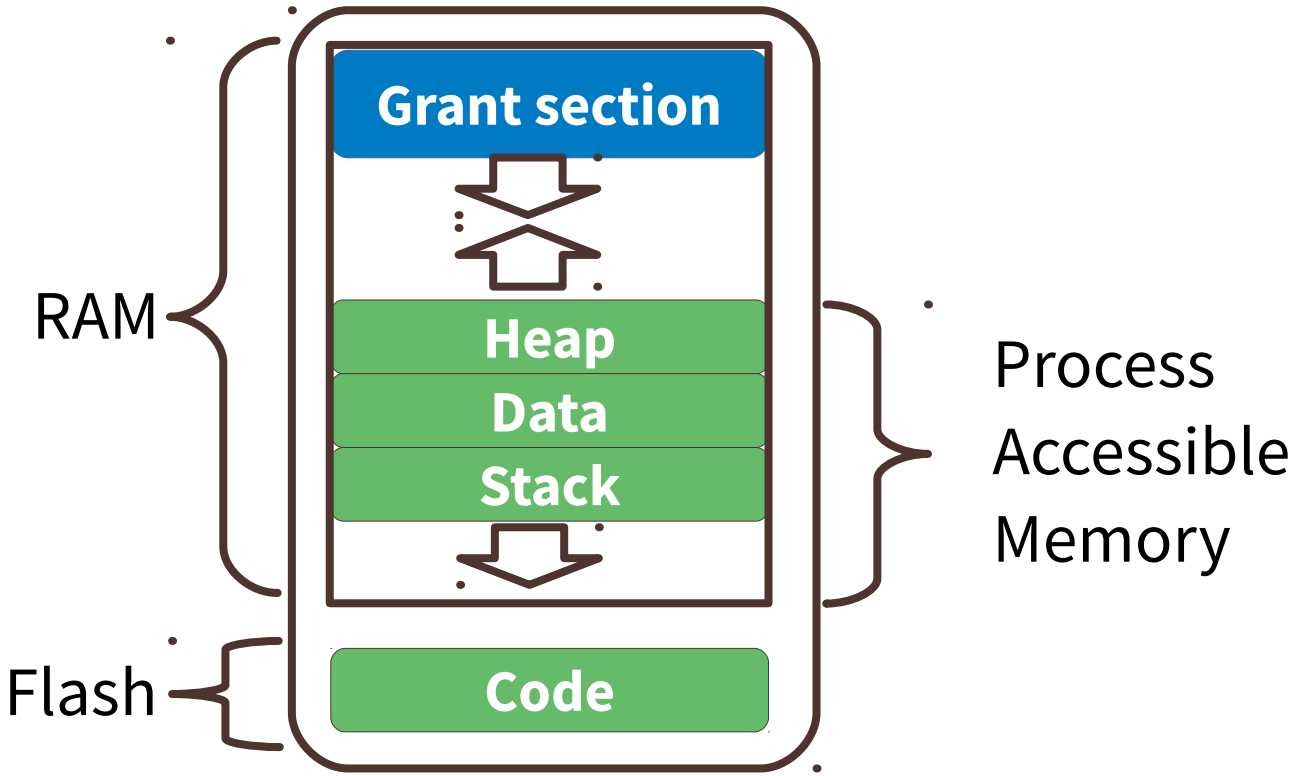
What About Dynamic Allocation?



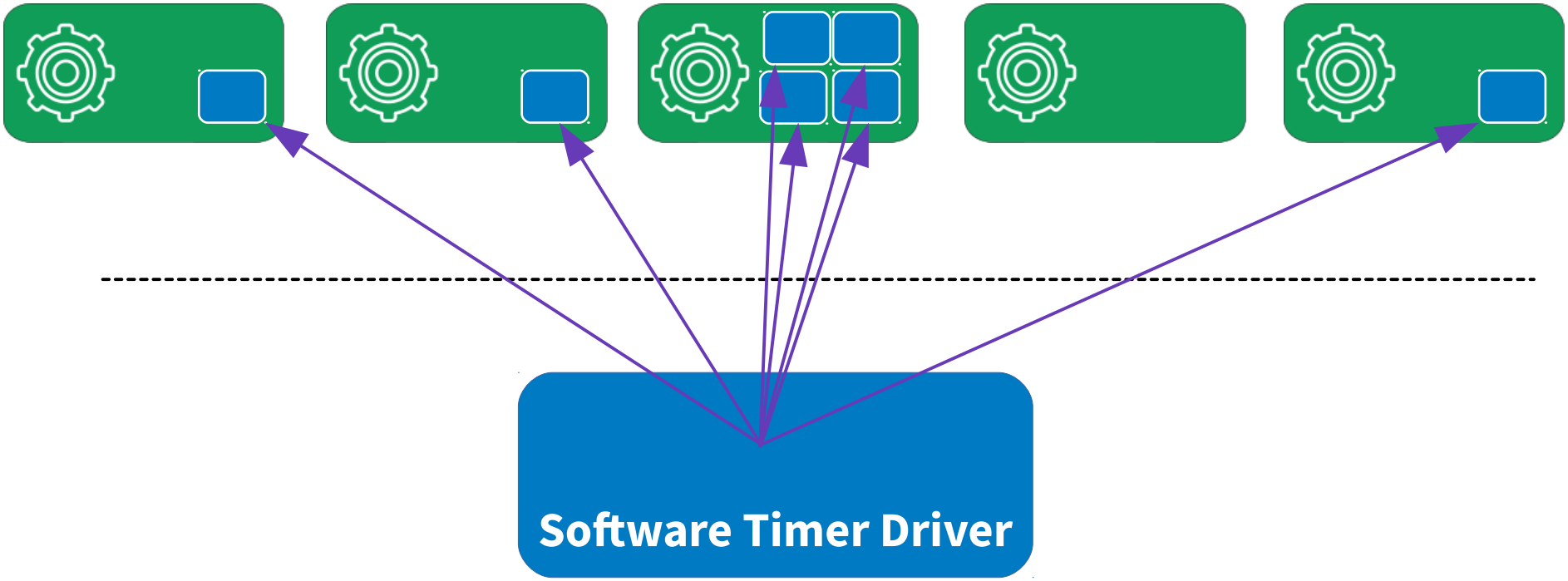
Can lead to unpredictable shortages.
One process's demands impacts capabilities of others.

Grants: Per-Process Kernel Heaps

- Allocations for one process do not affect others
- System proceeds if *one* grant section is exhausted
- All process resources freed on process termination



Grants: Per-Process Kernel Heaps



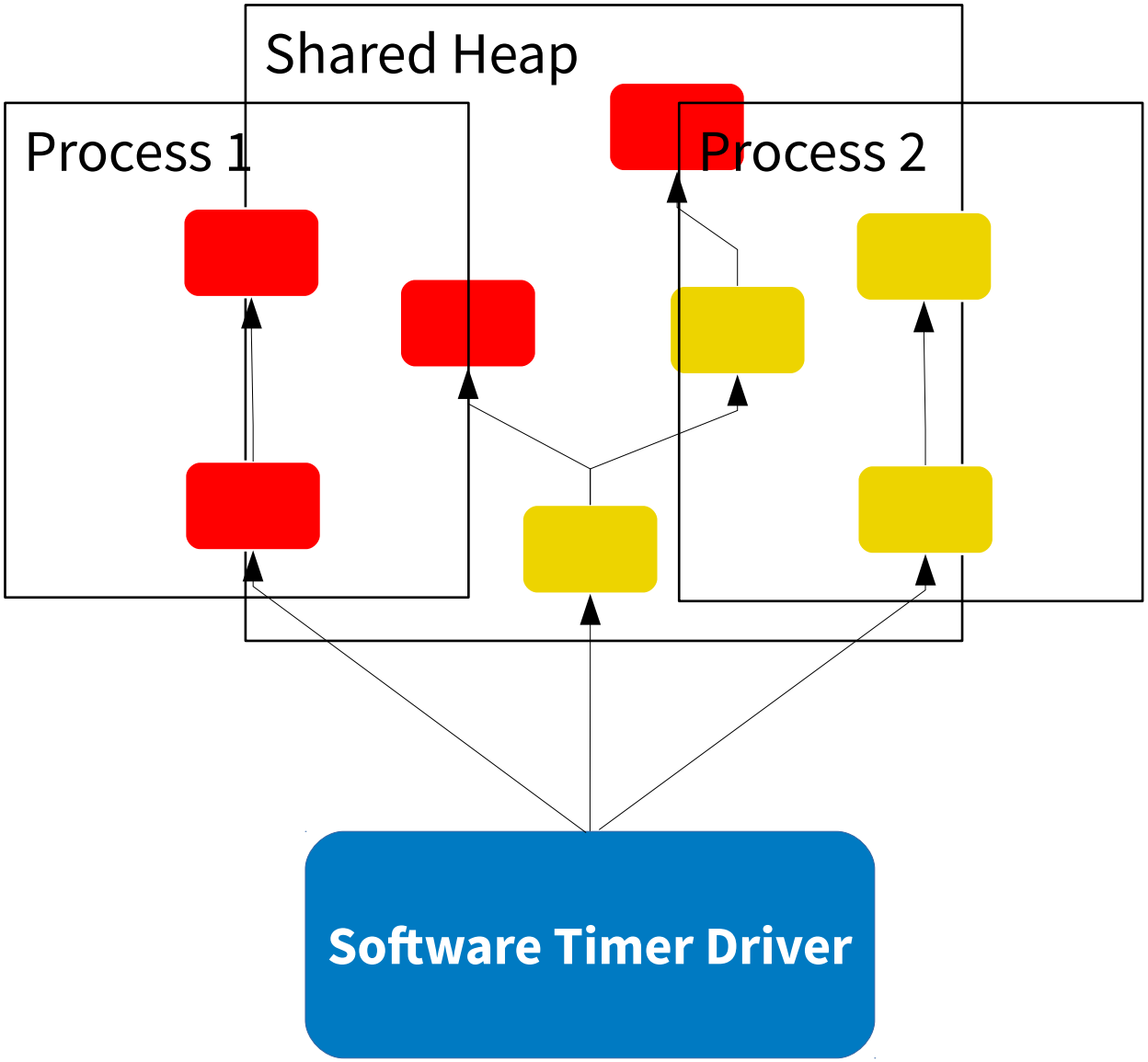
Grants balance safety and reliability of static allocation with flexibility of dynamic allocation

Grants use the type-system to ensure references only accessible when process is live

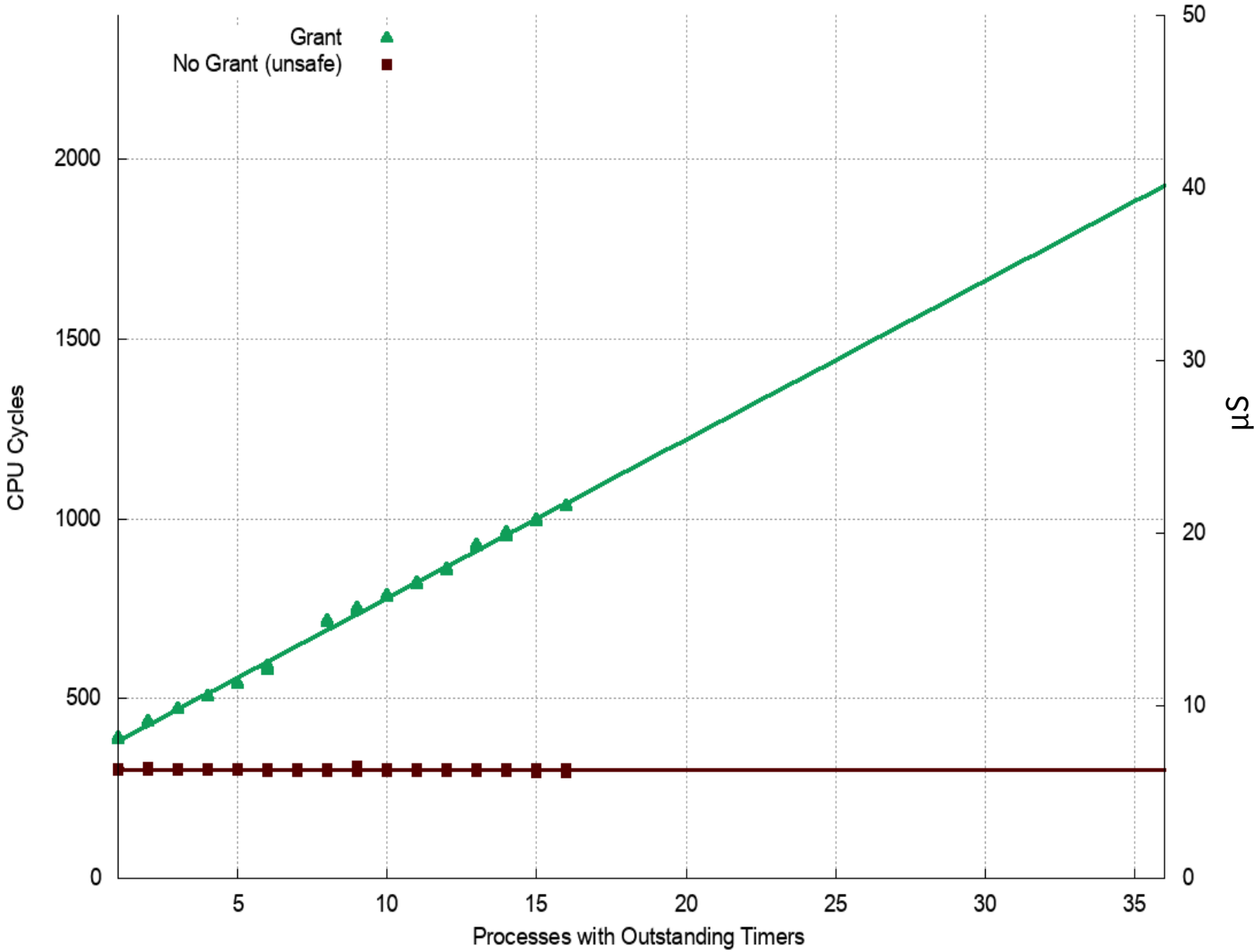
```
// Can't operate on timer data here
timer_grant.enter(process_id, |timer| {
    // Can operate on timer data here
    if timer.expiration > cur_time {
        timer.fired = true;
    }
});

// timer data can't escape here
```

```
fn enter<'a, F>(&'a self, pid: ProcId, f: F) →
    where F: for<'b> FnOnce(&'b mut T)
```

Grants: No Cross-Process Structures



Grants: No Cross-Process Structures



Multiprograming a Microcontroller

1. Memory & Performance Isolation

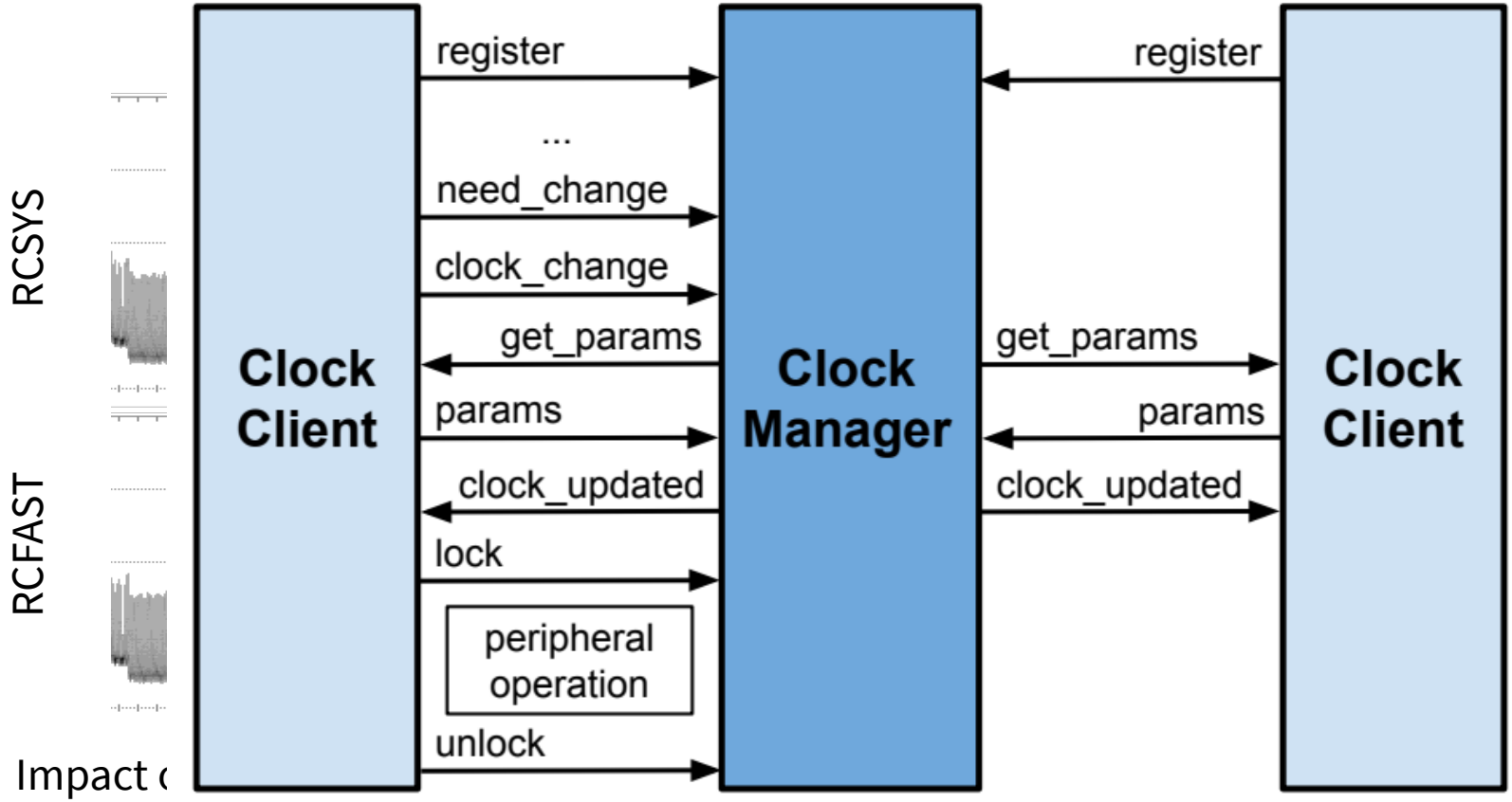
2. **Power & Clock Control**

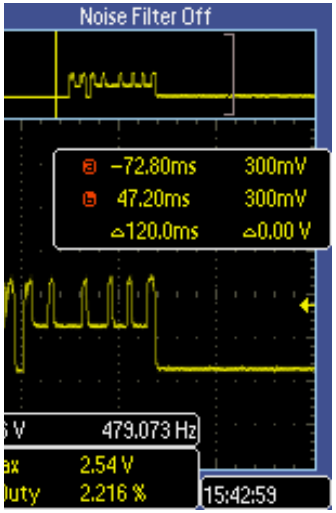
3. Peripheral communication busses

4. Future work

Power State & Clock Control

- Power draw combo of duty cycle and active draw
  - What's the deepest sleep state we can drop to?
  - Which clock should drive active peripherals?
- Multiprogramming makes both harder!





Impact (... 'orkload

Multiprograming a Microcontroller

1. Memory & Performance Isolation

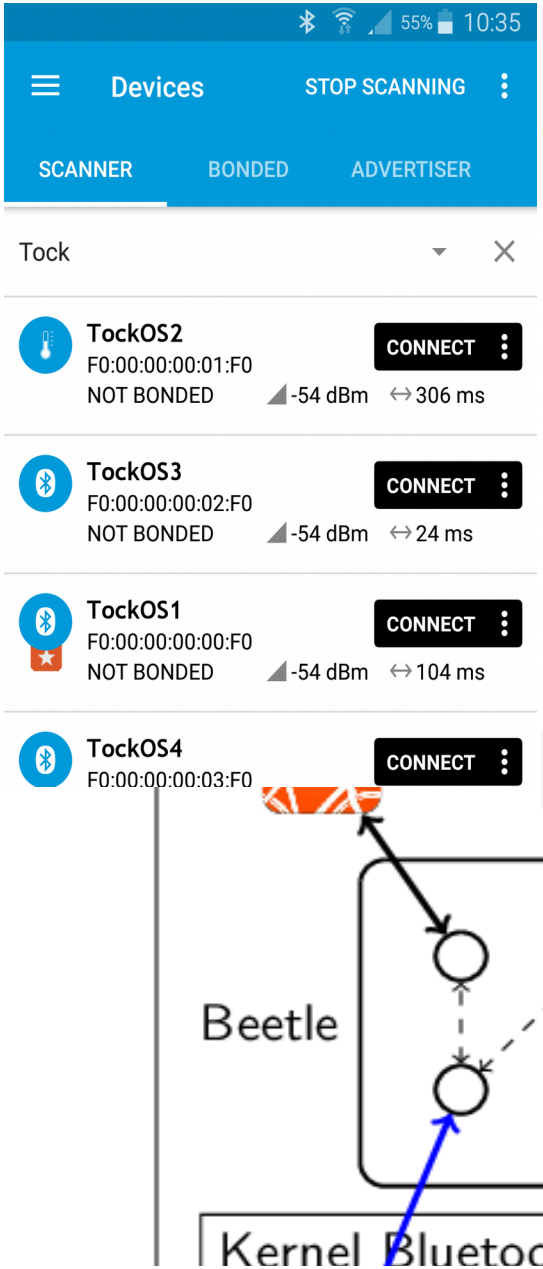
2. Power & Clock Control

3. **Peripheral communication busses**

4. Future work

Multiprogramming a Peripheral

- A *platform* must support all possible applications
  - Efficient protocols may not be so efficient anymore
- Peripheral communication not designed for multiprogramming
  - How to restart individual components?
  - How to isolate services?
  - Virtualizing vs. Multiplexing



Multiprograming a Microcontroller

1. Memory & Performance Isolation
2. Power & Clock Control
3. Peripheral communication busses
4. **Future work**

Future Work

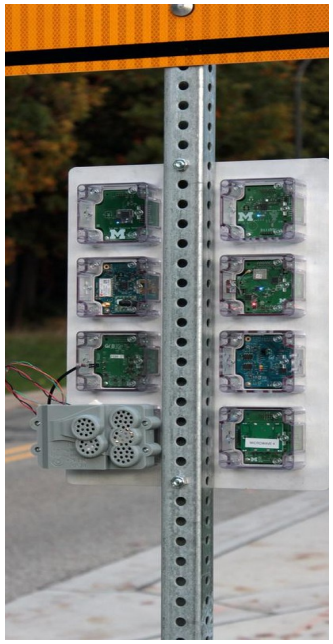
- Writing & Enforcing security policies
  - For the kernel: language-based capability system?
  - For processes: permissions without a file system?
  - For networked applications: cryptographic tokens?
- Debugging embedded applications
  - Security implications
  - Logging
- More applications, more hardware
  - RISC-V, x86
  - Wearables, sensor networks, security devices

Tock: A Secure Operating System for Microcontrollers

- Embedded devices are multiprogrammable
  - Security, Software Updates, Multi-tenancy
- Tension between isolation and resources
  - Traditional approaches insufficient for low memory
  - New programming languages & hardware features help
- Must also rethink: power management, networking, security policies...

Evaluating End-to-End

Evaluating with Practitioners



- Researchers working with Tock
- Half-day tutorials (~100 people)
- Open source community (45+ contributors)
- Embedded Systems class at Stanford

Security is an End-to-End Property

- Is threat model realistic?
- Can system builders extend TCB safely?
- Can developers build applications?

Realistic Threat Model?

	Signpost	Helium	Titan
Applications	Researchers	Customers	App Developers
Capsules	Module builders	Community, Helium Inc.	Product developers
Platform	Signpost authors	Helium Inc.	Titan team

Security is an End-to-End Property

- Is threat model realistic? ✓
- Can system builders extend TCB safely?
- Can developers build applications?

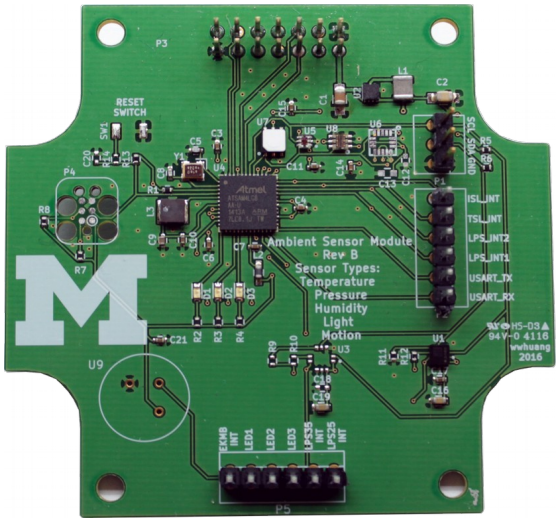
Safely Extend the TCB?

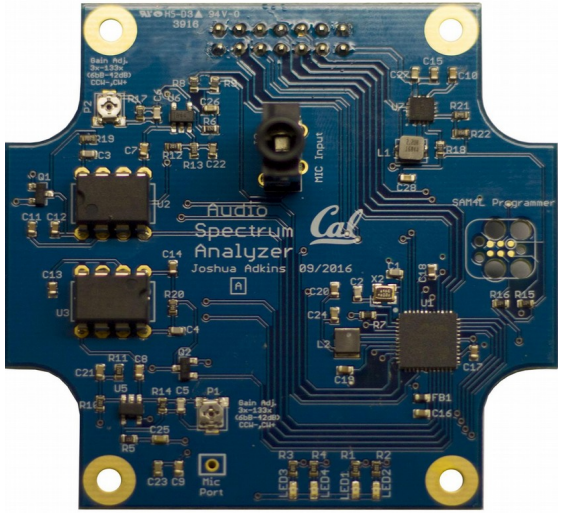
Rule out common errors by design:

- Synchronization with interrupt handlers
- Untrusted user pointers
- Use-after-free

Tock on Signpost

Each Signpost module runs a Tock kernel



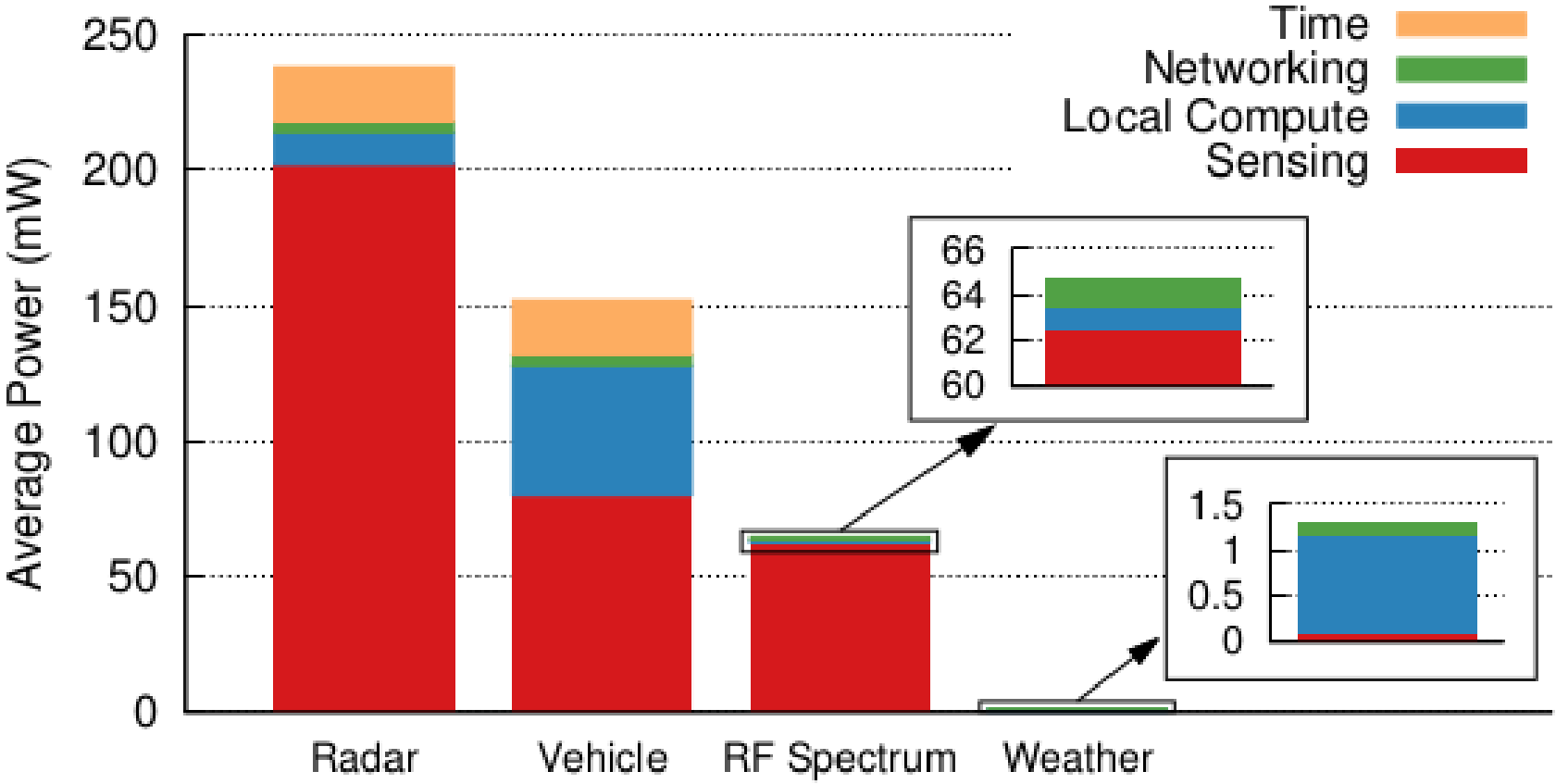


	Ambient Module	Audio Module
Process LoC	6990	6688
Capsules LoC	4479	3985
Platform LoC	3252	3244
	405 “unsafe”	381 “unsafe”

Security is an End-to-End Property

- Is threat model realistic? ✓
- Can system builders extend TCB safely? ✓
- Can developers build applications?

Writing Applications?



Energy consumption on Signpost applications*

*Adkins et al., IPSN'18

Future Work with Tock

Efficient Clock Management

- Can't statically choose clock domains
- Idea: Hide clock choices from app

Low-power wireless networking

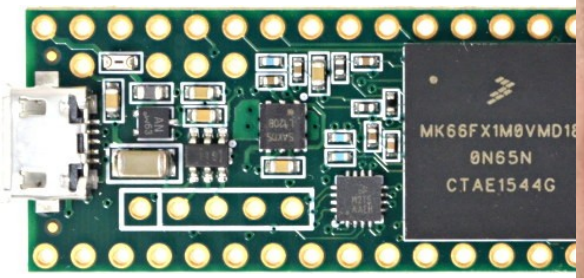
- 6lowpan implementation overheads
- Multi-application Bluetooth Low-Energy peripherals

Security Policies

- Specify access rights for processes/apps
- Enforce high-level policies on capsules

Tagline: An "App Store" for embedded systems

Teensy (Embedded Systems Class)



Platform LoC	4055
Unsafe LoC	170

Rust Features

- Type and memory safe
  - No buffer overflows, dangling pointers, type confusion, use-after-free…
- Compile-time enforced type system
  - No type artifacts at run time
- No garbage collection
  - Control over memory layout and execution
- Runtime behavior similar to C

Rust's Ownership

Key Property: Deallocate memory as soon as owner out of scope

```
{
    let x = Resource::new();
}
```

When the scope exits, x is no longer valid and the memory is "freed"

```
{
    let x = Resource::new();
    let y = x;
    println!("{}", y); // OK: value moved
    println!("{}", x); // compilation error!
}
```

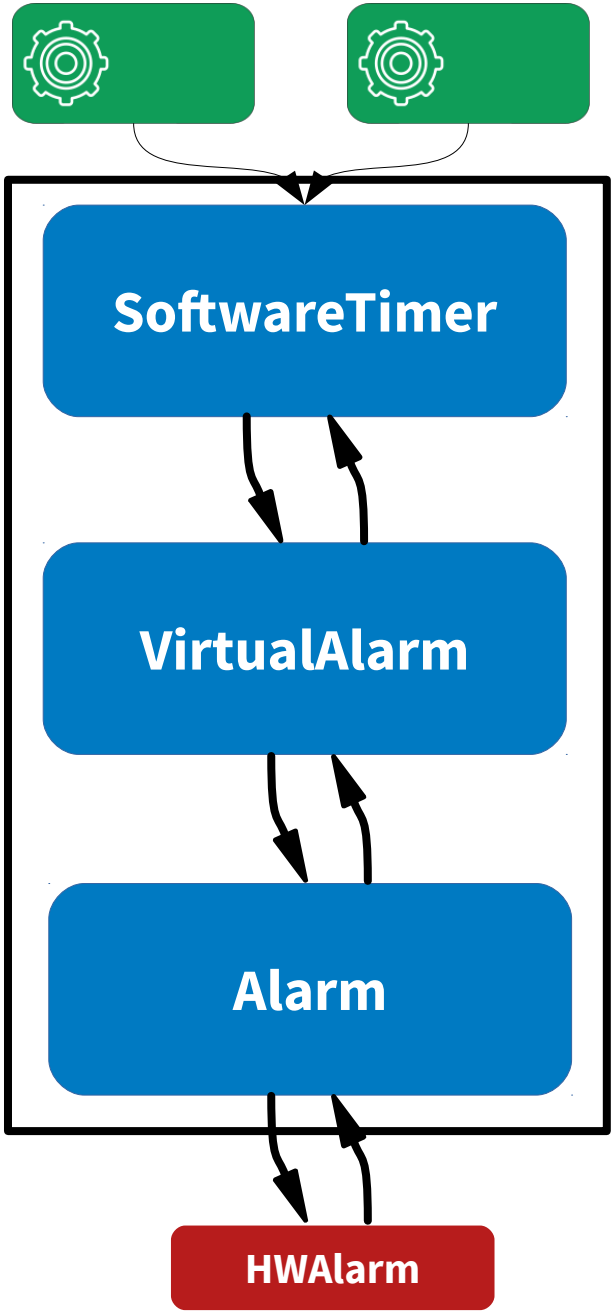
Borrowing

```rust
fn transform(x: &mut Resource) {
// the borrow is implicitly released.
}

let mut my_resource = Resource::new();
transform(&mut my_resource);
// my_resource still valid here
```

Just a pointer at runtime

- Mutable references (&mut) must be unique
- Shared references (&) cannot mutate the value

What About Circular Data-Structures?



Sum Types vs. Shared Mutability

```
enum NumOrPointer {
  Num(u32),
  Pointer(&mut u32)
}
```

```
$ rustc test.rs
error[E0506]: cannot assign to ‘external‘
because it is borrowed
```

```
// Note: will not compile
let external : &mut NumOrPointer;
if let Pointer(internal) = external {
  *external = Num(0xdeadbeef);
  *internal = 12345;
  // Kaboom: we’ve just written ‘12345‘
  // to the address ‘0xdeadbeef‘
}
```

Existential types for imperative languages, Dan Grossman, ESOP’02

Interior Mutability

Safe if we avoid mutability + aliasing concurrently

Examples from Rust core library:

- `Cell`: Copy-in/out or replace, no direct references
- `Mutex`: mutual-exclusion on internal reference

Can write our own with different semantics:

- `TakeCell`: non-blocking mutual-exclusion
  - Used in Tock for storing large buffers

Developer expresses: Which part of data is mutable?

```rust
pub struct SoftwareTimer {
  alarm: &VirtualAlarm,
  ...
}

pub struct VirtualAlarm {
  next_alarm: Cell<u32>,
  alarm: &HWAlarm,
  client: &SoftwareTimer,
  ...
}

pub struct HWAlarm {
  regs: [VolatileCell<u32>; 16],
  client: &VirtualAlarm,
  ...
}
```