Tock Operating System:
Security Model and Implications

Philip Levis
Stanford Platform Lab Annual Review
February 13, 2020

Embedded Software is a Weak Link

- Runs on small, low-power microcontrollers
  - 32-bit ARM CortexM @10-140 MHz
  - 16-256kB RAM, 256kB-1MB code
  - No virtual memory
- Custom, application-specific code written in C
  - Low code-reuse
  - Low testing coverage
- All code must be trusted
  - No isolation: any code can directly all access hardware registers
  - No distinction between “kernel” and “application”

Embedded Software is Security Critical

Tock

- Tock: First secure embedded operating system
  - Key idea: write kernel in Rust, a type-safe systems language
- Platform Lab research project
  - Collaboration between Stanford, UC Berkeley, Michigan
  - Lead: Amit Levy, graduated in 2018 (now Princeton faculty)
- Uses today: Helium, Google (OpenTitan, OpenSK), Oxide (server firmware)

Security Model

- Evolving from research to production requires a much more precise statement of the security model
  - OpenTitan: root of trust on computing devices
  - OpenSK: FIDO two-factor device
  - Oxide: firmware can't be the weak link
- But we need the OS to remain generally and broadly usable in embedded systems
  - Disallowing X because one use case needs it won't scale
  - Allowing Y because one case needs it breaks least privilege
- Basic tension between security and usability

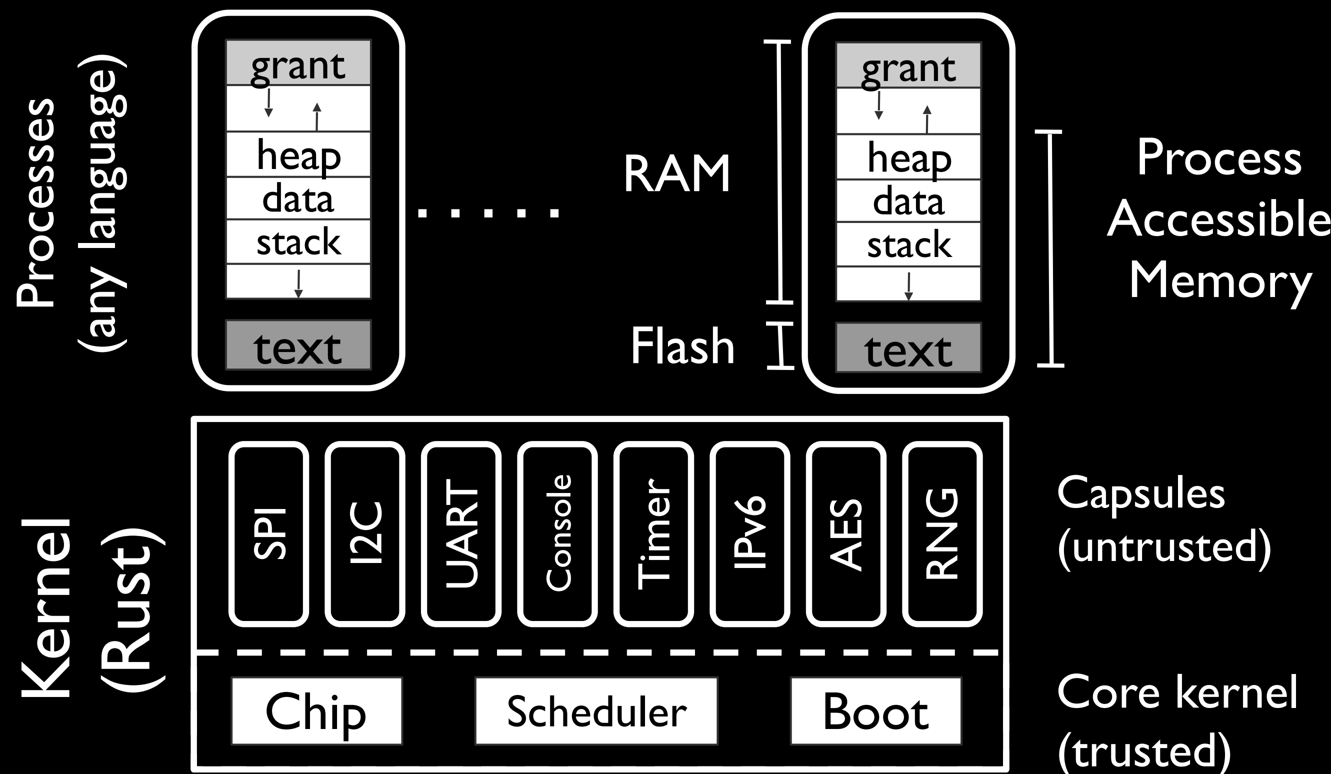
Summary

- Impose a standard security model
  - E.g., no heap allocation, kernel extensions can't control processes
- Some operations violate standard security model
  - E.g., being able to allocate memory, stop a process
- Calling sensitive operations requires having a capability from trusted code, enforced by Rust type system
  - Statically checkable
  - Leverage Tock's model of trusted and untrusted code
  - Two classes of capabilities: functional and global

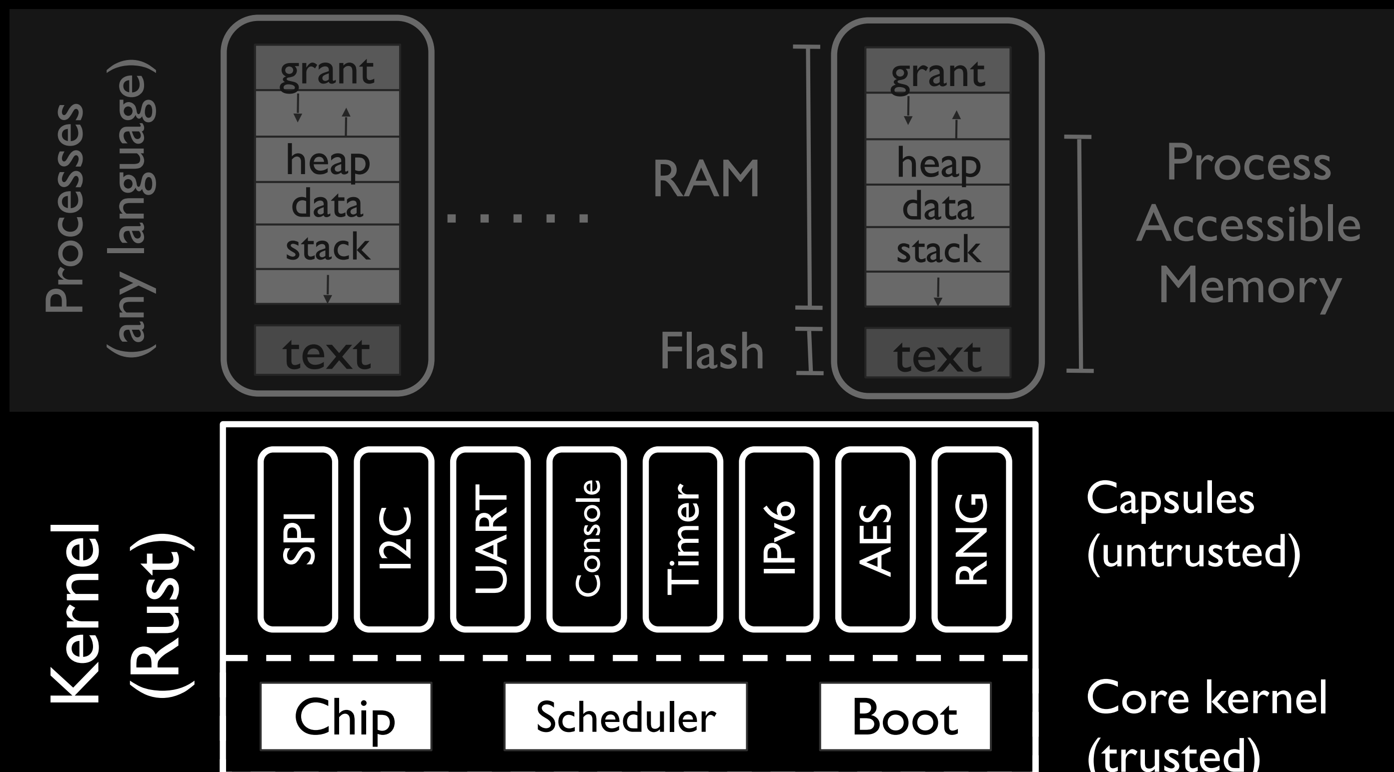
Outline

- Introduction
- Tock overview
- Security model
- Allowing exceptions
- Next steps

Tock Architecture



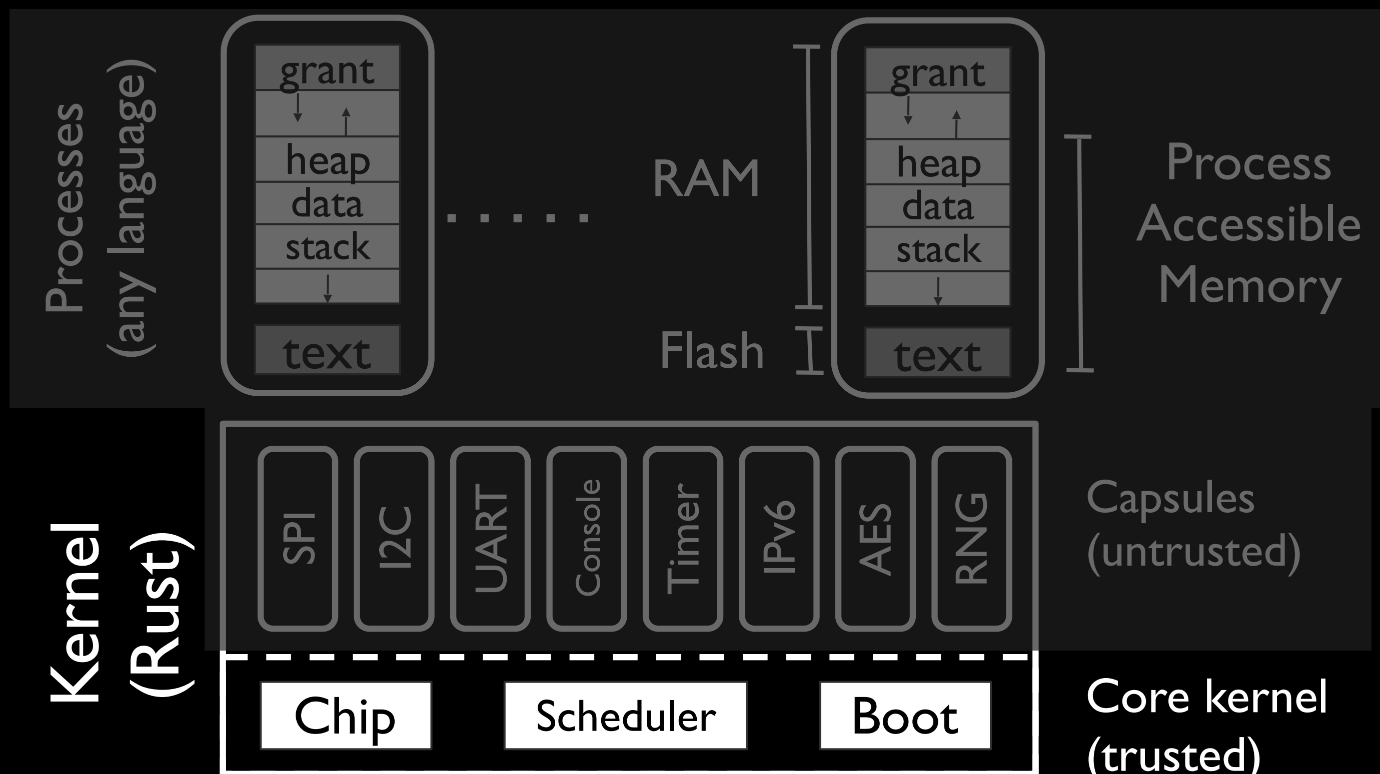
Tock Architecture



Rust Language

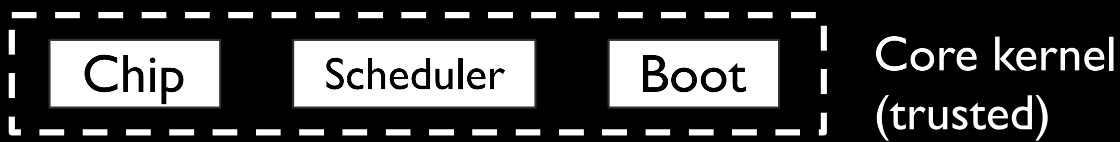
- The Tock kernel is written in Rust, a type-safe systems language
  - Strongly type-safe code: no buffer overflows, access errors, wraparound bugs
  - Fast: within 30% of C in our use cases, overhead due to safety due diligence
  - No garbage collector: uses "lifetimes" and affine type system
- Some kernel code cannot be type safe
  - Address of a memory mapped IO register taken from datasheet
  - Context switches
  - Interrupt handlers
  - System calls and processing userspace pointers
- Use **`unsafe`** keyword to step outside type system
  - Core kernel uses **`unsafe`** 102 times (most uses declare unsafe functions), has 31 unsafe code blocks
  - Chip peripherals use **`unsafe`** to construct a typed structure for register access from an address

Tock Core Kernel

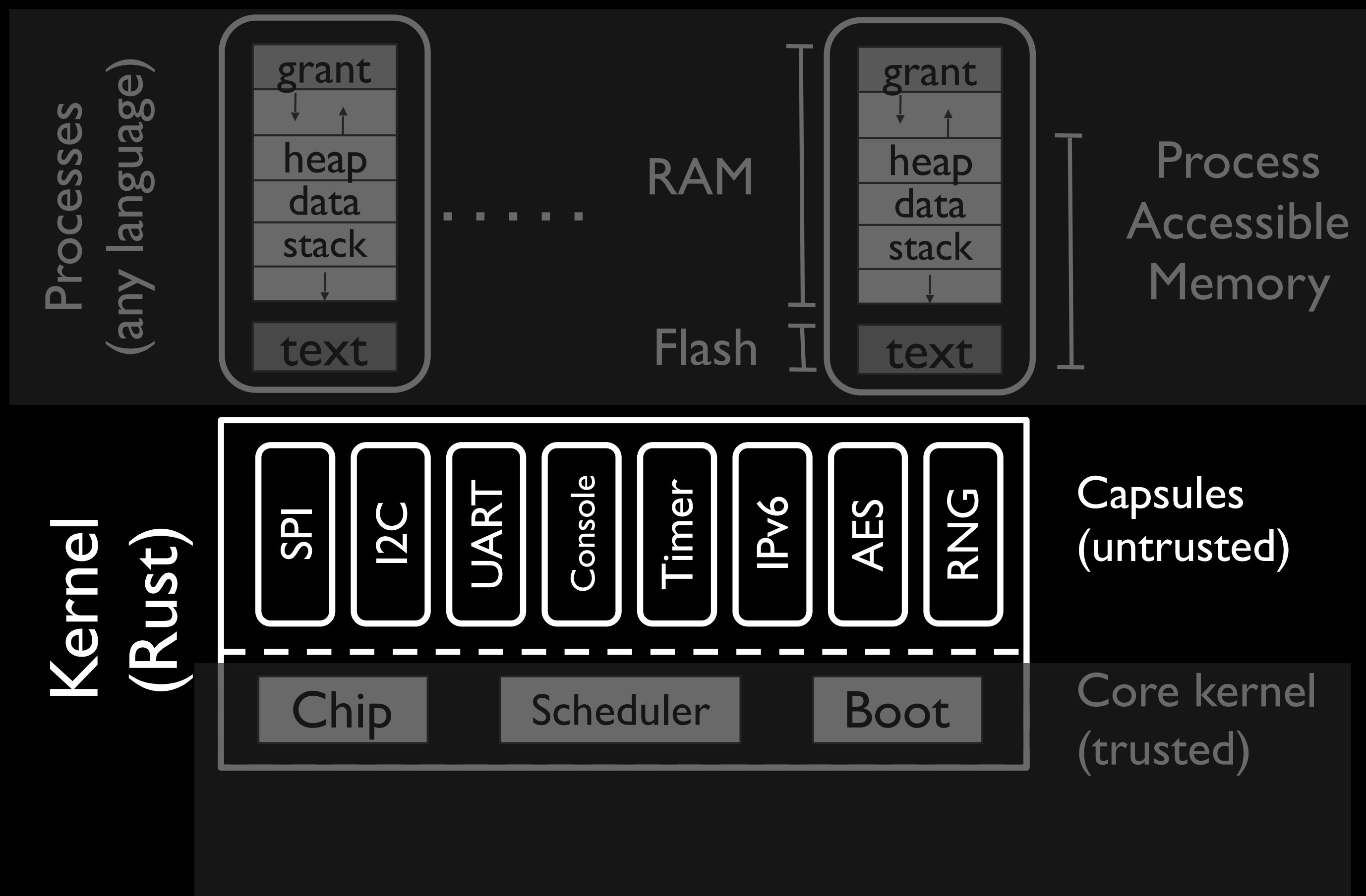


Core Kernel and Peripheral Drivers

- This code may use the unsafe keyword: we trust it
  - kernel/ is the core kernel code, context switches, low-level data types
  - chips/ are chip peripherals (UART, etc.); unsafe is hopefully one line converting a memory address into a type set of register structures
  - boards/ is the board-specific code, including the boot sequence (which is unsafe)

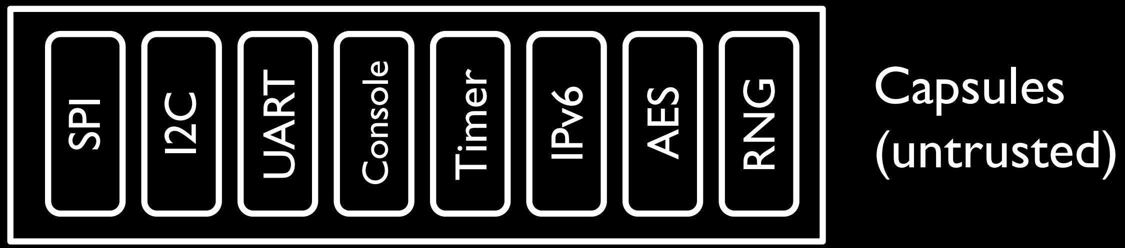


Tock Capsules

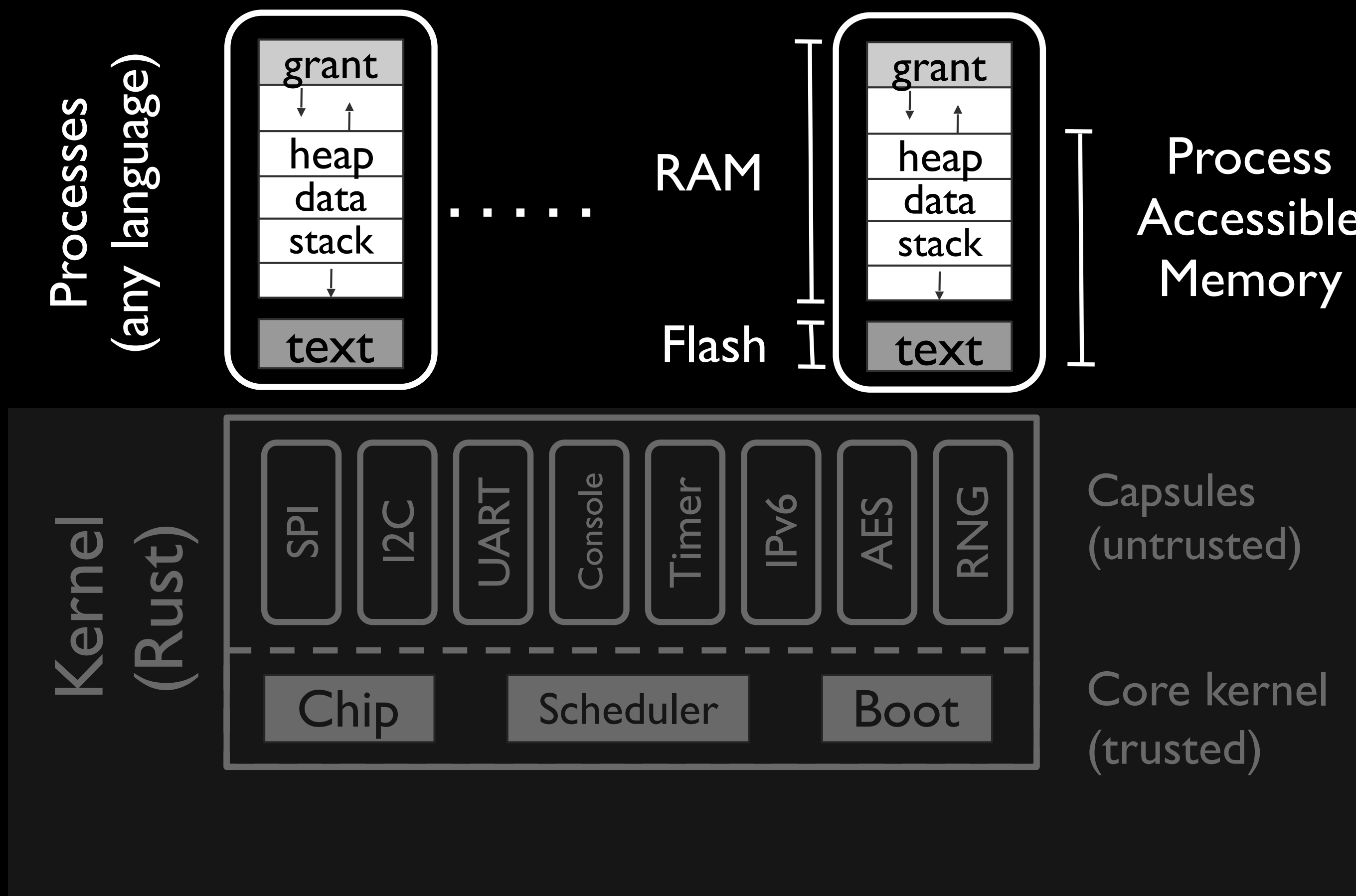


Tock Capsules

- Kernel code (Rust) that *cannot* use the unsafe keyword
- Reusable pieces of kernel code that are chip and board independent
  - Example: virtual_alarm turns one hardware alarm into many software alarms
  - Example: rf233 implements an RF233 radio driver on top of a SPI bus and GPIO pins
  - Example: rng provides system call interface to hardware random number generator



Tock Processes



Tock Processes

- Tock supports processes written in any language (i.e, assembly)
- Kernel isolates processes using memory protection unit (MPU)
- Kernel has no dynamic memory pool/malloc (for dependability)
  - When the kernel needs to dynamically allocate memory for a process, it allocates it from a special region of process RAM, called the *grant* region
  - If a process exits/faults, the grant region is automatically freed
  - References cannot escape grant region
- Some capsules implement system calls to allow processes call into kernel
  - Kernel responsible for multiplexing/virtualizing its operations

Outline

- Introduction
- Tock overview
- Security model
- Allowing exceptions
- Next steps

TockWorld 4

- TockWorld: yearly workshop for Tock developers
- Two major work items in November 2019
  - Updating system call API
  - Johnathan van Why (Google) led the effort to clearly define Tock's security model

Application Security

- Confidentiality: Secrets may not be accessed by other applications or untrusted capsules.

Application Security

- Confidentiality: Secrets may not be accessed by other applications or untrusted capsules.
- Integrity: Data may not be modified by other applications or untrusted capsules except when the application allows.

Application Security

- Confidentiality: Secrets may not be accessed by other applications or untrusted capsules.
- Integrity: Data may not be modified by other applications or untrusted capsules except when the application allows.

Examples: IPC, passing buffer into kernel

Application Security

- Confidentiality: Secrets may not be accessed by other applications or untrusted capsules.
- Integrity: Data may not be modified by other applications or untrusted capsules except when the application allows.
- Availability: Applications may not deny service to one another, except that finite resources may be granted in a first-come, first-served order.

Application Security

- Confidentiality: Secrets may not be accessed by other applications or untrusted capsules.
- Integrity: Data may not be modified by other applications or untrusted capsules except when the application allows.
- Availability: Applications may not deny service to one another, except that finite resources may be granted in a first-come, first-served order.

Example: TCP port

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.
- Integrity: Applications and untrusted capsules may not modify kernel data except through APIs intentionally exposed by the owning code.

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.
- Integrity: Applications and untrusted capsules may not modify kernel data except through APIs intentionally exposed by the owning code.

Can only invoke APIs which they have explicitly been given access to through a reference. Type safety and trait-centric API designs mean usually only trusted code passes these references, e.g., at boot.

Example: Writing a Log

```
trait LogWrite {...}
trait LogWriteClient {...}
impl LogWrite for LogStorage {...}
```

- A capsule can only write to the log if it has a reference to an implementer of LogWrite (e.g., LogStorage)
- There are no global variables: a capsule can only have a reference if it was passed one (at boot)

```
let test = LogStorageTest::new(&log_storage,
                               &mut BUFFER,
                               VirtualMuxAlarm::new(mux_alarm),
                               &TEST_OPS);
```

- Strict, narrow APIs mean references don't float around

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.
- Integrity: Applications and untrusted capsules may not modify kernel data except through APIs intentionally exposed by the owning code.
- Availability: Applications cannot starve the kernel of resources or otherwise perform denial-of-service attacks against the kernel. Untrusted capsule code may deny service to trusted kernel code. Virtualized kernel abstractions should be designed to prevent starvation.

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.
- Integrity: Applications and untrusted capsules may not modify kernel data except through APIs intentionally exposed by the owning code.
- Availability: Applications cannot starve the kernel of resources or otherwise perform denial-of-service attacks against the kernel. Untrusted capsule code may deny service to trusted kernel code. Virtualized kernel abstractions should be ... .

Kernel is single-threaded.

Kernel Security

- Confidentiality: Secrets may not be accessed by applications or untrusted capsules.
- Integrity: Applications and untrusted capsules may not modify kernel data except through APIs intentionally exposed by the owning code.
- Availability: Applications cannot starve the kernel of resources or otherwise perform denial-of-service attacks against the kernel. Untrusted capsule code may deny service to trusted kernel code. Virtualized kernel abstractions should be designed to prevent starvation.

Starvation is Inevitable

- Most analog-to-digital converters support high-frequency sampling
  - Audio/mic: "Sample pin A7 with a 3.3V reference at 40kHz"
- Samples in some use cases need not only high frequency but also low jitter: driven off a fixed clock
  - Can't stop/change sampling without introducing jitter
  - If a client starts, it might sample forever: this will starve other clients
- Don't want to disallow audio sampling, but also don't want to allow any capsule that performs high frequency sampling to do so forever.

Outline

- Introduction
- Tock overview
- Security model
- Allowing exceptions
- Next steps

Allowing Exceptions

- There are a lot of things that most capsules shouldn't be able to do (availability)
  - Sample ADC forever
  - Start/stop processes
  - Allocate memory from processes
  - Invoke kernel main loop
  - Open network connections
- But some capsules need to!

Allowing Exceptions

- There are a lot of things that most capsules shouldn't be able to do (availability)
  - Sample ADC forever: audio sampling
  - Start/stop processes: process management capsule
  - Allocate memory from processes: system call drivers
  - Invoke kernel main loop: scheduler
  - Open network connections: services
- But some capsules need to!

Capabilities

- Some in-kernel operations are especially sensitive and impact dependability
  - Allocating memory from a process
  - Calling main loop
  - Loading/start/stopping process
  - Under development: network communication
- Could control access by requiring a reference to structure that implements feature
  - Requires many references; uglier code and wasted RAM; poor for fine-grained control
- Tock uses *capabilities* to describe and restrict these operations within the kernel
  - A Tock capability is a zero-sized structure that can only be created by trusted code
  - Can be passed but not copied
  - Calling sensitive functions requires a reference to corresponding capability
  - Checked at compile time, requires no RAM and no extra code

Capabilities Example

kernel/src/capabilities.rs:

```
pub unsafe trait ProcessManagementCapability {}
```

kernel/src/sched.rs:

```
/// Run a closure on every valid process. This will iterate the array of
/// processes and call the closure on every process that exists.
crate fn process_each<F>(&self, closure: F) where F: Fn(&process::ProcessType) {...}

/// process_each, when called from outside kernel (by a capsule) requires
/// a ProcessManagementCapability
pub fn process_each_capability<F>(&'static self,
                                  _capability: &capabilities::ProcessManagementCapability,
                                  closure: F) where F: Fn(usize,
                                                          &process::ProcessType) {...}
```

Capabilities Example

```
pub fn process_each_capability<F>(&'static self,
                                 _capability: &ProcessManagementCapability,
                                 closure: F) where F: Fn(usize,&ProcessType) {...}


impl<'a, C: ProcessManagementCapability> ProcessConsole<'a, C> {
    pub fn new(..., capability: C) -> ProcessConsole<'a, C> {
        ProcessConsole {
          ...
          capability: capability,
        }
    }
}


self.kernel.process_each_capability(&self.capability,
                                    |_i, proc| { ... function ... });
```

Allowing Exceptions

- There are a lot of things that most capsules shouldn't be able to do (availability)
  - Sample ADC forever: AdcDurationCapability
  - Start/stop processes: ProcessManagementCapability
  - Allocate memory from processes: MemoryAllocationCapability
  - Invoke kernel main loop: MainLoopCapability
  - Open network connections: NetworkCapability
- Boot sequence (or other trusted code) creates and distributes these capabilities

Outline

- Introduction
- Tock overview
- Security model
- Allowing exceptions
- Next steps

Security Through Types

- Rust's type system allows us to statically verify security properties and restrictions
  - Functional: references to objects limit access to APIs
  - Global: capabilities limit access to operations that violate security model
- Current work: network security policies
  - E.g., enforcing that a capsule can only open connections to certain endpoints/ports
  - Parameterized capabilities are not zero-sized and require runtime checks

OpenTitan

- Consortium led by Google to create an open-source hardware root of trust MCU
  - All designs and RTL are public
  - RISC-V (rv32i) design
- Tock is currently OS of record for OpenTitan
  - First port by Alastair Sinclair (Western Digital) from only public information
  - Tock OpenTitan working group has formed (Google, Oxide, Western Digital, a few academics)
- OpenTitan is still in flux: tight collaboration between Tock developers and chip designers

Questions