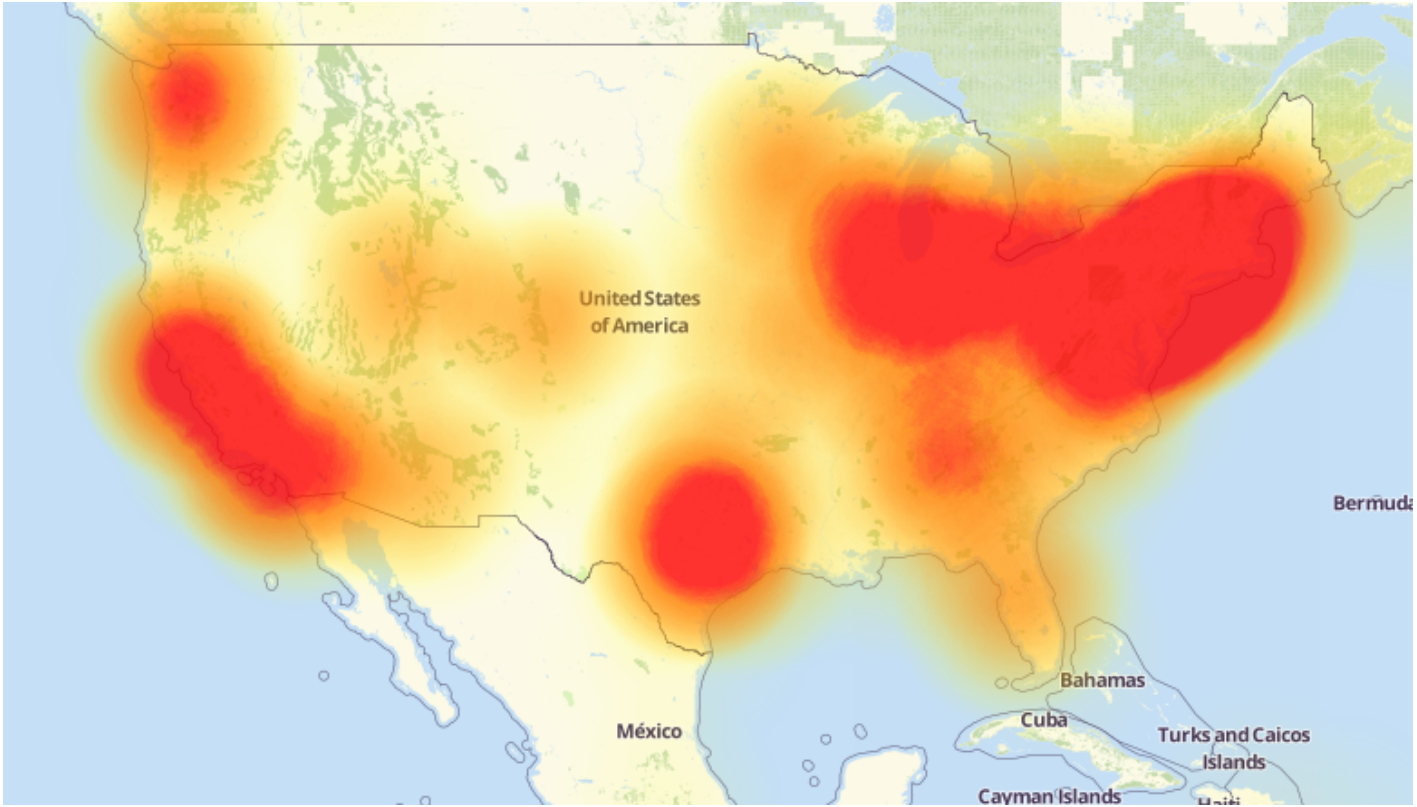
A Case for Type-System Based Network Security

Hudson Ayers, Armin Namavari, Philip Levis
Stanford University

NGOSCPS ‘19 - Montreal

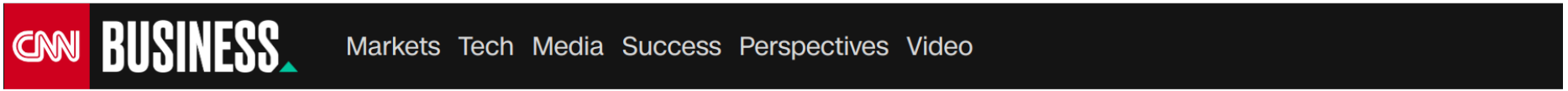
Network Security for IoT - Why it Matters



Mirai Botnet

- Enabled by factory-set passwords on Unix-based IoT devices with open ssh and telnet interfaces
- Most targets were IP cameras, home routers, and DVR’s
- **Infected devices could attack any nodes on the Internet: exponential spread**
- ~500,000 infected devices [1]
- Made inaccessible: Github, Twitter, Reddit, Netflix, Airbnb, etc.

Network Security for IoT - Why it Matters



FDA confirms that St. Jude's cardiac devices can be hacked

St. Jude’s cardiac pacemaker attack

- Attackers could issue shocks or change pacing remotely using RF commands from a distance
- Enabled by 24-bit RSA authentication + fixed override
- **Additionally, patient medical information was sometimes transmitted unencrypted**

Network Security for IoT - Why it Matters

German parents told to destroy doll that can spy on children

German watchdog classifies My Friend Cayla doll as 'illegal espionage apparatus' and says shops and owners could face fines

- **Unencrypted transmission of data**
- Unauthenticated access to device

Takeaways

- Most IoT attacks occur over the network, rather than via direct physical access to the device
- Important to restrict the damage that can be done even on a system with compromised components
- Encrypting communication by default is extremely important

Our Position

An embedded Operating System’s network security should be defined in that languages type system.

Why?

- Tightly localized enforcement of network security → reduced audit space
- Compile-time checks reduce cost in code size and memory
- Puts security inline with code: keeps security at the forefront of the developers mind

Survey of embedded security approaches

- Process isolation
- File access permissions
- Namespaces
- Netfilter / iptables → in kernel firewall

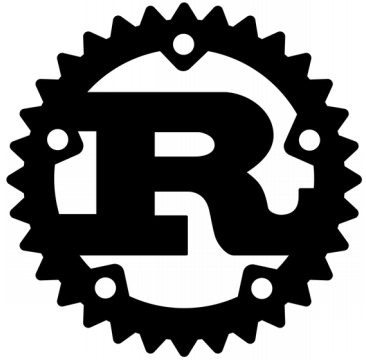
- Capability based security
- Formally verified micro-kernel
- “Provably” bug free
- Kernel extension is a laborious process

Type system based security

Core concept: Capability-based security where each capability is a unique type in the OS programming language

- The compiler can track where capabilities are distributed
- Compiler enforces that only trusted code can *instantiate or modify* these types
- Example: written in Rust for the Tock Operating System

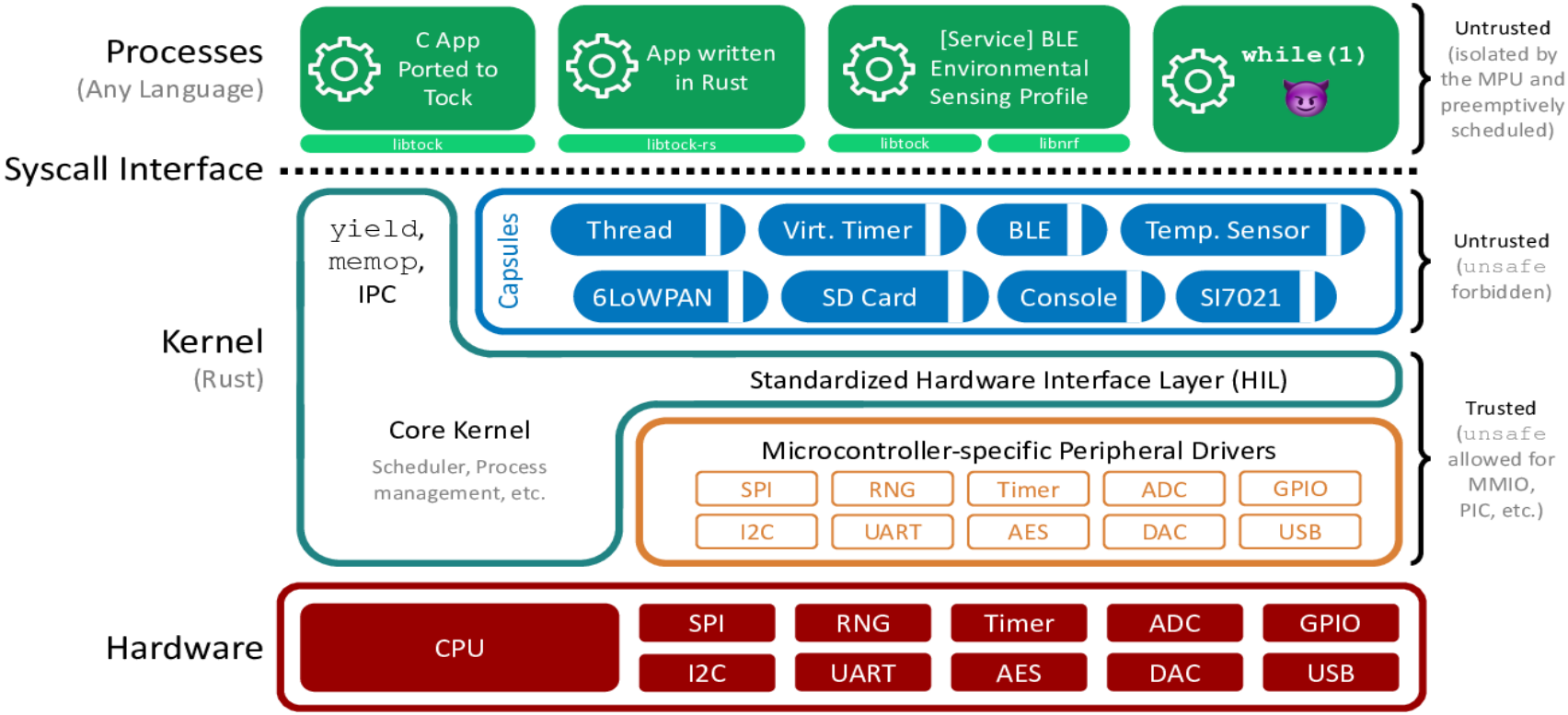
Introduction to Rust



Systems programming language focused on safety.

- Type safe
- Performance comparable to C/C++ (No GC)
- Memory Safe + Thread Safe: Does not allow null pointers, dangling pointers, or data races in *safe* code. **Unsafe** code (demarcated by the `unsafe` keyword) allows developers to break these guarantees
- Ownership: For a given object, can have either multiple immutable references or a single mutable reference

Tock Embedded Operating System



Example Network Capabilities

(1) “Capsule A cannot listen to the radio”

(2) “Capsule A cannot use the socket interface”

(3) “Capsule C can not send data without first encrypting it”

(4) “No two capsules can bind to the same UDP port”

(5) “Capsule B can only communicate with IP addresses 10.0.0.5 and 146.72.80.1”

(6) “Capsule D can talk to any node on the link-local network”

Tock today: Static capabilities

```
macro_rules! create_capability {
  ($T: ty) => {{
    struct Cap;
    unsafe impl $T for Cap {}
    Cap
  };};
}
...
let send_unencrypted = create_capability!(capabilities::SendUnencryptedCapability);
```

Pros:

- Zero code size
- Explicit passing of capabilities

Cons:

- Not extensible to non-binary capabilities

Static Capabilities - Not Enough

Static Capabilities cannot cover cases such as:

- “Capsule A can only communicate with IP addresses 10.0.0.5 and 146.72.80.1”

- “Capsule B can only communicate with ‘amazon.com’ and ‘nest.com’ domains”

- “Capsule C can bind to ports 500-1000”

Dynamic Capabilities - Leveraging the Type System

- Each dynamic capability is a type; the fields of that type encode the variable capabilities
- Example: Capability that controls which IP endpoints a capsule can communicate with
- The core kernel (trusted code) can create and modify dynamic capabilities
- Capsules (untrusted) can read dynamic capabilities

net_permissions.rs (core kernel)

```
#[derive(Copy, Clone)]
pub enum AddrRange {
    Any, // Any address
    Range(u32, u32),
    Addr(u32),
}

pub struct IpPermission {
    remote_addrs: AddrRange, // not pub!
}
```

net_permissions.rs (core kernel)

```
impl IpPermission {
        pub unsafe fn new(addrs: AddrRange) -> IpPermission {
            IpPermission {
                remote_addrs: addrs,
            }
        }
        pub fn get_remotes(&self) -> AddrRange {
            self.remote_addrs
        }
    }
}
```

ip.rs (capsule)

```
pub fn ip_send(dest: u32, permission: IpPermission, msg: String) -> Result<u8, ()> {
    let valid = match permission.get_remotes() {
        AddrRange::Range(start, end) => {
            if dest >= start && dest <= end {
                true
            } else {
                false
            }
        },
    };
    if valid {
        sixlowpan_send(...);
        return Ok(0);
    }
    Err(())
}
```

trusted_capsule.rs (capsule)

```
pub struct TrustedObject {
    permission: Option<IpPermission>,
}

impl TrustedObject {
    pub fn new(perm: IpPermission) -> TrustedObject {
        TrustedObject {
            permission: Some(perm),
        }
    }

    pub fn phone_home(self, msg: String) {
        self.permission.map(|perm| ip_send(63, perm, msg));
    }
}
```

untrusted_capsule.rs (capsule)

```
pub struct UntrustedObject {
    permission: Option<IpPermission>,
}

impl UntrustedObject {
    pub fn new(perm: IpPermission) -> UntrustedObject {
        UntrustedObject {
            permission: Some(perm),
        }
    }

    pub fn do_evil(self) {
        self.permission.map(|perm| ip_send(63, perm, "bad\n".to_string()));
    }
}
```

main.rs (Initialization Script) - correct use

```
let permission;
unsafe {
    permission = IpPermission::new(AddrRange::Any);
}
let result = ip_send(6, permission, "hello world\n".to_string());

let permission2;
unsafe {
    permission2 = IpPermission::new(AddrRange::Range(0, 100));
}

let trusted = TrustedObject::new(permission2);
trusted.phone_home("good_data\n".to_string());
```

untrusted_capsule.rs (Capsules) - Attempted misuse

```
pub fn do_evil(self) {
    ip_send(63, NULL, "bad_data\n".to_string());
}
```

Doesn’t Compile - no NULL in Rust

untrusted_capsule.rs (Capsules) - Attempted misuse

```
pub fn do_evil(self) {
        let permission = IpPermission::new(63);
    ip_send(63, permission, "bad_data\n".to_string());
}
```

Doesn’t Compile: `error[E0133]: call to unsafe function is unsafe and requires unsafe function or block`

untrusted_capsule.rs (Capsules) - Attempted misuse

```
pub fn do_evil(self) {
        let permission = IpPermission {
        remote_addrs: AddrRange::Any,
        }
    ip_send(63, permission, "bad_data\n".to_string());
}
```

Doesn’t Compile - `error[E0616]: field `remote_addrs` of struct `permissions::IpPermissions` is private`

untrusted_capsule.rs (Capsules) - Attempted misuse

```
pub fn do_evil(self, permission) {
        permission.remote_addrs = AddrRange::Any;
    ip_send(63, permission, "bad_data\n".to_string());
}
```

Doesn’t Compile - `error[E0616]: field `remote_addrs` of struct `permissions::IpPermissions` is private`

main.rs (Initialization Script) - Attempted misuse

```
let permission3;
unsafe {
    permission3 = IpPermission::new(AddrRange::Addr(24));
}

let untrusted = UntrustedObject::new(permission3);
untrusted.do_evil();
```

Send fails at runtime.

Benefits - Reduced Audit Space

Tock ADC Capsule:

```
//! Provides userspace applications with the ability to sample
//! analog signals.
//!
//! Usage
//! -----
//!
//! ```
//! let adc_channels = static_init!(
//!     [&'static sam4l::adc::AdcChannel; 6],
//!     [
//!         &sam4l::adc::CHANNEL_AD0, // A0
//!         &sam4l::adc::CHANNEL_AD1, // A1
//!         &sam4l::adc::CHANNEL_AD3, // A2
//!         &sam4l::adc::CHANNEL_AD4, // A3
//!         &sam4l::adc::CHANNEL_AD5, // A4
//!         &sam4l::adc::CHANNEL_AD6, // A5
//!     ]
//! );
//! let adc = static_init!(
//!     capsules::adc::Adc<'static, sam4l::adc::Adc>,
//!     capsules::adc::Adc::new(
//!         &mut sam4l::adc::ADC0,
//!         adc_channels,
//!         &mut capsules::adc::ADC_BUFFER1,
//!         &mut capsules::adc::ADC_BUFFER2,
//!         &mut capsules::adc::ADC_BUFFER3
//!     )
//! );
//! sam4l::adc::ADC0.set_client(adc);
//! ```

use core::cell::Cell;
use core::cmp;
use kernel::common::cells::{MapCell, OptionalCell, TakeCell};
use kernel::hil;
use kernel::{AppId, AppSlice, Callback, Driver, ReturnCode, Shared};

/// Syscall driver number.
use crate::driver;
pub const DRIVER_NUM: usize = driver::NUM::ADC as usize;

/// ADC application driver, used by applications to interact with ADC.
/// Not currently virtualized, only one application can use it at a time.
pub struct Adc<'a, A: hil::adc::Adc + hil::adc::AdcHighSpeed> {
    // ADC driver
    adc: &'a A,
    channels: &'a [&'a <A as hil::adc::Adc>::Channel],

    // ADC state
    active: Cell<bool>,
    mode: Cell<AdcMode>,

    // App state
    app: MapCell<App>,
    channel: Cell<usize>,
    callback: OptionalCell<Callback>,
    app_buf_offset: Cell<usize>,
    samples_remaining: Cell<usize>,
    samples_outstanding: Cell<usize>,
    next_samples_outstanding: Cell<usize>,
    using_app_buf1: Cell<bool>,

    // ADC buffers
    adc_buf1: TakeCell<'static, [u16]>,
    adc_buf2: TakeCell<'static, [u16]>,
    adc_buf3: TakeCell<'static, [u16]>,
}

/// ADC modes, used to track internal state and to signify to applications which
/// state a callback came from
#[derive(Copy, Clone, Debug, PartialEq)]
enum AdcMode {
    NoMode = -1,
    SingleSample = 0,
    ContinuousSample = 1,
    SingleBuffer = 2,
    ContinuousBuffer = 3,
}

/// Holds buffers that the application has passed us
#[derive(Default)]
pub struct App {
    app_buf1: Option<AppSlice<Shared, u8>>,
    app_buf2: Option<AppSlice<Shared, u8>>,
}

/// Buffers to use for DMA transfers
/// The size is chosen somewhat arbitrarily, but has been tested. At 175000 Hz,
/// buffers need to be swapped every 70 us and copied over before the next
/// swap. In testing, it seems to keep up fine.
pub static mut ADC_BUFFER1: [u16; 128] = [0; 128];
pub static mut ADC_BUFFER2: [u16; 128] = [0; 128];
pub static mut ADC_BUFFER3: [u16; 128] = [0; 128];

/// Functions to create, initialize, and interact with the ADC
impl<A: hil::adc::Adc + hil::adc::AdcHighSpeed> Adc<'a, A> {
    /// Create a new Adc application interface
    ///
    /// adc - ADC driver to provide application access to
    /// channels - list of ADC channels usable by applications
    /// adc_buf1 - buffer used to hold ADC samples
    /// adc_buf2 - second buffer used when continuously sampling ADC
    pub fn new(
        adc: &'a A,
        channels: &'a [&'a <A as hil::adc::Adc>::Channel],
        adc_buf1: &'static mut [u16; 128],
        adc_buf2: &'static mut [u16; 128],
        adc_buf3: &'static mut [u16; 128],
    ) -> Adc<'a, A> {
        Adc {
            // ADC driver
            adc: adc,
            channels: channels,

            // ADC state
            active: Cell::new(false),
            mode: Cell::new(AdcMode::NoMode),

            // App state
            app: MapCell::new(App::default()),
            channel: Cell::new(0),
            callback: OptionalCell::empty(),
            app_buf_offset: Cell::new(0),
            samples_remaining: Cell::new(0),
            samples_outstanding: Cell::new(0),
            next_samples_outstanding: Cell::new(0),
            using_app_buf1: Cell::new(true),

            // ADC buffers
            adc_buf1: TakeCell::new(adc_buf1),
            adc_buf2: TakeCell::new(adc_buf2),
            adc_buf3: TakeCell::new(adc_buf3),
        }
    }

    /// Store a buffer we've regained ownership of and return a handle to it
    /// The handle can have `map` called on it in order to process the data in
    /// the buffer
    ///
    /// buf - buffer to be stored
    fn replace_buffer(&self, buf: &'static mut [u16]) -> &TakeCell<'static, [u16]> {
        if self.adc_buf1.is_none() {
            self.adc_buf1.replace(buf);
            &self.adc_buf1
        } else if self.adc_buf2.is_none() {
            self.adc_buf2.replace(buf);
            &self.adc_buf2
        } else {
            self.adc_buf3.replace(buf);
            &self.adc_buf3
        }
    }

    /// Find a buffer to give to the ADC to store samples in
    ///
    /// closure - function to run on the found buffer
    fn take_and_map_buffer<F: FnOnce(&'static mut [u16])>(&self, closure: F) {
        if self.adc_buf1.is_some() {
            self.adc_buf1.take().map(|val| {
                closure(val);
            });
        } else if self.adc_buf2.is_some() {
            self.adc_buf2.take().map(|val| {
                closure(val);
            });
        } else if self.adc_buf3.is_some() {
            self.adc_buf3.take().map(|val| {
                closure(val);
            });
        }
    }

    /// Collect a single analog sample on a channel
    ///
    /// channel - index into `channels` array, which channel to sample
    fn sample(&self, channel: usize) -> ReturnCode {
        // only one sample at a time
        if self.active.get() {
            return ReturnCode::EBUSY;
        }

        // convert channel index
        if channel >= self.channels.len() {
            return ReturnCode::EINVAL;
        }
        let chan = self.channels[channel];

        // save state for callback
        self.active.set(true);
        self.mode.set(AdcMode::SingleSample);
        self.channel.set(channel);

        // start a single sample
        let res = self.adc.sample(chan);
        if res != ReturnCode::SUCCESS {
            // failure, clear state
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);

            return res;
        }

        ReturnCode::SUCCESS
    }

    /// Collected repeated single analog samples on a channel
    ///
    /// channel - index into `channels` array, which channel to sample
    /// frequency - number of samples per second to collect
    fn sample_continuous(&self, channel: usize, frequency: u32) -> ReturnCode {
        // only one sample at a time
        if self.active.get() {
            return ReturnCode::EBUSY;
        }

        // convert channel index
        if channel >= self.channels.len() {
            return ReturnCode::EINVAL;
        }
        let chan = self.channels[channel];

        // save state for callback
        self.active.set(true);
        self.mode.set(AdcMode::ContinuousSample);
        self.channel.set(channel);

        // start a single sample
        let res = self.adc.sample_continuous(chan, frequency);
        if res != ReturnCode::SUCCESS {
            // failure, clear state
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);

            return res;
        }

        ReturnCode::SUCCESS
    }

    /// Collect a buffer-full of analog samples
    /// Samples are collected into the first app buffer provided. The number of
    /// samples collected is equal to the size of the buffer "allowed"
    ///
    /// channel - index into `channels` array, which channel to sample
    /// frequency - number of samples per second to collect
    fn sample_buffer(&self, channel: usize, frequency: u32) -> ReturnCode {
        // only one sample at a time
        if self.active.get() {
            return ReturnCode::EBUSY;
        }

        // convert channel index
        if channel >= self.channels.len() {
            return ReturnCode::EINVAL;
        }
        let chan = self.channels[channel];

        // cannot sample a buffer without a buffer to sample into
        let mut app_buf_length = 0;
        let exists = self.app.map_or(false, |state| {
            app_buf_length = state.app_buf1.as_mut().map_or(0, |buf| buf.len());
            state.app_buf1.is_some()
        });
        if !exists {
            return ReturnCode::ENOMEM;
        }

        // save state for callback
        self.active.set(true);
        self.mode.set(AdcMode::SingleBuffer);
        self.app_buf_offset.set(0);
        self.channel.set(channel);

        // start a continuous sample
        let res = self.adc_buf1.take().map_or(ReturnCode::EBUSY, |buf1| {
            self.adc_buf2.take().map_or(ReturnCode::EBUSY, move |buf2| {
                // determine request length
                let request_len = app_buf_length / 2;
                let len1;
                let len2;
                if request_len <= buf1.len() {
                    len1 = app_buf_length / 2;
                    len2 = 0;
                } else if request_len <= (buf1.len() + buf2.len()) {
                    len1 = buf1.len();
                    len2 = request_len - buf1.len();
                } else {
                    len1 = buf1.len();
                    len2 = buf2.len();
                }

                // begin sampling
                self.using_app_buf1.set(true);
                self.samples_remaining.set(request_len - len1 - len2);
                self.samples_outstanding.set(len1 + len2);
                let (rc, retbuf1, retbuf2) = self
                    .adc
                    .sample_highspeed(chan, frequency, buf1, len1, buf2, len2);
                if rc != ReturnCode::SUCCESS {
                    // store buffers again
                    retbuf1.map(|buf| {
                        self.replace_buffer(buf);
                    });
                    retbuf2.map(|buf| {
                        self.replace_buffer(buf);
                    });
                }
                rc
            })
        });
        if res != ReturnCode::SUCCESS {
            // failure, clear state
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);
            self.samples_remaining.set(0);
            self.samples_outstanding.set(0);

            return res;
        }

        ReturnCode::SUCCESS
    }

    /// Collect analog samples continuously
    /// Fills one "allowed" application buffer at a time and then swaps to
    /// filling the second buffer. Callbacks occur when the in use "allowed"
    /// buffer fills
    ///
    /// channel - index into `channels` array, which channel to sample
    /// frequency - number of samples per second to collect
    fn sample_buffer_continuous(&self, channel: usize, frequency: u32) -> ReturnCode {
        // only one sample at a time
        if self.active.get() {
            return ReturnCode::EBUSY;
        }

        // convert channel index
        if channel >= self.channels.len() {
            return ReturnCode::EINVAL;
        }
        let chan = self.channels[channel];

        // cannot continuously sample without two buffers
        let mut app_buf_length = 0;
        let mut next_app_buf_length = 0;
        let exists = self.app.map_or(false, |state| {
            app_buf_length = state.app_buf1.as_mut().map_or(0, |buf| buf.len());
            next_app_buf_length = state.app_buf2.as_mut().map_or(0, |buf| buf.len());
            state.app_buf1.is_some() && state.app_buf2.is_some()
        });
        if !exists {
            return ReturnCode::ENOMEM;
        }

        // save state for callback
        self.active.set(true);
        self.mode.set(AdcMode::ContinuousBuffer);
        self.app_buf_offset.set(0);
        self.channel.set(channel);

        // start a continuous sample
        let res = self.adc_buf1.take().map_or(ReturnCode::EBUSY, |buf1| {
            self.adc_buf2.take().map_or(ReturnCode::EBUSY, move |buf2| {
                // determine request lengths
                let samples_needed = app_buf_length / 2;
                let next_samples_needed = next_app_buf_length / 2;

                // determine request lengths
                let len1;
                let len2;
                if samples_needed <= buf1.len() {
                    // we can fit the entire app_buffer request in the first
                    // buffer. The second buffer will be used for the next
                    // app_buffer
                    len1 = samples_needed;
                    len2 = cmp::min(next_samples_needed, buf2.len());
                    self.samples_remaining.set(0);
                    self.samples_outstanding.set(len1);
                } else if samples_needed <= (buf1.len() + buf2.len()) {
                    // we can fit the entire app_buffer request between the two
                    // buffers
                    len1 = buf1.len();
                    len2 = samples_needed - buf1.len();
                    self.samples_remaining.set(0);
                    self.samples_outstanding.set(len1 + len2);
                } else {
                    // the app_buffer is larger than both buffers, so just
                    // request max lengths
                    len1 = buf1.len();
                    len2 = buf2.len();
                    self.samples_remaining.set(samples_needed - len1 - len2);
                    self.samples_outstanding.set(len1 + len2);
                }

                // begin sampling
                self.using_app_buf1.set(true);
                let (rc, retbuf1, retbuf2) = self
                    .adc
                    .sample_highspeed(chan, frequency, buf1, len1, buf2, len2);
                if rc != ReturnCode::SUCCESS {
                    // store buffers again
                    retbuf1.map(|buf| {
                        self.replace_buffer(buf);
                    });
                    retbuf2.map(|buf| {
                        self.replace_buffer(buf);
                    });
                }
                rc
            })
        });
        if res != ReturnCode::SUCCESS {
            // failure, clear state
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);
            self.samples_remaining.set(0);
            self.samples_outstanding.set(0);

            return res;
        }

        ReturnCode::SUCCESS
    }

    /// Stops sampling the ADC
    /// Any active operation by the ADC is canceled. No additional callbacks
    /// will occur. Also retrieves buffers from the ADC (if any)
    fn stop_sampling(&self) -> ReturnCode {
        if !self.active.get() || self.mode.get() == AdcMode::NoMode {
            // already inactive!
            return ReturnCode::SUCCESS;
        }

        // clean up state
        self.active.set(false);
        self.mode.set(AdcMode::NoMode);
        self.app_buf_offset.set(0);

        // actually cancel the operation
        let rc = self.adc.stop_sampling();
        if rc != ReturnCode::SUCCESS {
            return rc;
        }

        // reclaim buffers
        let (rc, buf1, buf2) = self.adc.retrieve_buffers();

        // store buffers again
        buf1.map(|buf| {
            self.replace_buffer(buf);
        });
        buf2.map(|buf| {
            self.replace_buffer(buf);
        });

        // return result
        rc
    }
}

/// Callbacks from the ADC driver
impl<A: hil::adc::Adc + hil::adc::AdcHighSpeed> hil::adc::Client for Adc<'a, A> {
    /// Single sample operation complete
    /// Collects the sample and provides a callback to the application
    ///
    /// sample - analog sample value
    fn sample_ready(&self, sample: u16) {
        if self.active.get() && self.mode.get() == AdcMode::SingleSample {
            // single sample complete, clean up state
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);

            // perform callback
            self.callback.map(|callback| {
                callback.schedule(
                    AdcMode::SingleSample as usize,
                    self.channel.get(),
                    sample as usize,
                );
            });
        } else if self.active.get() && self.mode.get() == AdcMode::ContinuousSample {
            // sample ready in continuous sampling operation, keep state

            // perform callback
            self.callback.map(|callback| {
                callback.schedule(
                    AdcMode::ContinuousSample as usize,
                    self.channel.get(),
                    sample as usize,
                );
            });
        } else {
            // operation probably canceled. Make sure state is consistent. No
            // callback
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);
        }
    }
}

/// Callbacks from the High Speed ADC driver
impl<A: hil::adc::Adc + hil::adc::AdcHighSpeed> hil::adc::HighSpeedClient for Adc<'a, A> {
    /// Internal buffer has filled from a buffered sampling operation.
    /// Copies data over to application buffer, determines if more data is
    /// needed, and performs a callback to the application if ready. If
    /// continuously sampling, also swaps application buffers and continues
    /// sampling when neccessary. If only filling a single buffer, stops
    /// sampling operation when the application buffer is full.
    ///
    /// buf - internal buffer filled with analog samples
    /// length - number of valid samples in the buffer, guaranteed to be less
    ///          than or equal to buffer length
    fn samples_ready(&self, buf: &'static mut [u16], length: usize) {
        // do we expect a buffer?
        if self.active.get()
            && (self.mode.get() == AdcMode::SingleBuffer
                || self.mode.get() == AdcMode::ContinuousBuffer)
        {
            // we did expect a buffer. Determine the current application state
            self.app.map(|state| {
                // determine which app buffer to copy data into and which is
                // next up if we're in continuous mode
                let app_buf;
                let next_app_buf;
                if self.using_app_buf1.get() {
                    app_buf = state.app_buf1.as_mut();
                    next_app_buf = state.app_buf2.as_ref();
                } else {
                    app_buf = state.app_buf2.as_mut();
                    next_app_buf = state.app_buf1.as_ref();
                }

                // update count of outstanding sample requests
                self.samples_outstanding
                    .set(self.samples_outstanding.get() - length);

                // provide a new buffer and length request to the ADC if
                // necessary. If we haven't received enough samples for the
                // current app_buffer, we may need to place more requests. If we
                // have received enough, but are in continuous mode, we should
                // place a request for the next app_buffer. This is all
                // unfortunately made more complicated by the fact that there is
                // always one outstanding request to the ADC.
                let perform_callback;
                if self.samples_remaining.get() == 0 {
                    // we have already placed outstanding requests for all the
                    // samples needed to fill the current app_buffer

                    if self.samples_outstanding.get() == 0 {
                        // and the samples we just received are the last ones
                        // we need
                        perform_callback = true;

                        if self.mode.get() == AdcMode::ContinuousBuffer {
                            // it's time to switch to the next app_buffer, but
                            // there's already an outstanding request to the ADC
                            // for the next app_buffer that was placed last
                            // time, so we need to account for that
                            let samples_needed = next_app_buf.map_or(0, |buf| buf.len() / 2);
                            self.samples_remaining
                                .set(samples_needed - self.next_samples_outstanding.get());
                            self.samples_outstanding
                                .set(self.next_samples_outstanding.get());
                            self.using_app_buf1.set(!self.using_app_buf1.get());

                            // we also need to place our next request, however
                            // the outstanding request already placed for the
                            // next app_buffer might have completed it! So we
                            // have to account for that case
                            if self.samples_remaining.get() == 0 {
                                // oh boy. We actually need to place a request
                                // for the next next app_buffer (which is
                                // actually the current app_buf, but try not to
                                // think about that...). In practice, this
                                // should be a pretty uncommon case to hit, only
                                // occurring if the length of the app buffers
                                // are smaller than the length of the adc
                                // buffers, which is unsustainable at high
                                // sampling frequencies
                                let next_next_app_buf = &app_buf;

                                // provide a new buffer. However, we cannot
                                // currently update state since the next
                                // app_buffer still has a request outstanding.
                                // We'll just make a request and handle the
                                // state updating on next callback
                                self.take_and_map_buffer(|adc_buf| {
                                    let samples_needed =
                                        next_next_app_buf.as_ref().map_or(0, |buf| buf.len() / 2);
                                    let request_len = cmp::min(samples_needed, adc_buf.len());
                                    self.next_samples_outstanding.set(request_len);
                                    let (res, retbuf) =
                                        self.adc.provide_buffer(adc_buf, request_len);
                                    if res != ReturnCode::SUCCESS {
                                        retbuf.map(|buf| {
                                            self.replace_buffer(buf);
                                        });
                                    }
                                });
                            } else {
                                // okay, we still need more samples for the next
                                // app_buffer

                                // provide a new buffer and update state
                                self.take_and_map_buffer(|adc_buf| {
                                    let request_len =
                                        cmp::min(self.samples_remaining.get(), adc_buf.len());
                                    self.samples_remaining
                                        .set(self.samples_remaining.get() - request_len);
                                    self.samples_outstanding
                                        .set(self.samples_outstanding.get() + request_len);
                                    let (res, retbuf) =
                                        self.adc.provide_buffer(adc_buf, request_len);
                                    if res != ReturnCode::SUCCESS {
                                        retbuf.map(|buf| {
                                            self.replace_buffer(buf);
                                        });
                                    }
                                });
                            }
                        }
                    } else {
                        // but there are still outstanding samples for the
                        // current app_buffer (actually exactly one request, the
                        // one the ADC is currently acting on)
                        perform_callback = false;

                        if self.mode.get() == AdcMode::ContinuousBuffer {
                            // we're in continuous mode, so we need to start the
                            // first request for the next app_buffer

                            // provide a new buffer. However, we cannot
                            // currently update state since the current
                            // app_buffer still has a request outstanding. We'll
                            // just make a request and handle the state updating
                            // on next callback
                            self.take_and_map_buffer(|adc_buf| {
                                let samples_needed = next_app_buf.map_or(0, |buf| buf.len() / 2);
                                let request_len = cmp::min(samples_needed, adc_buf.len());
                                self.next_samples_outstanding.set(request_len);
                                let (res, retbuf) = self.adc.provide_buffer(adc_buf, request_len);
                                if res != ReturnCode::SUCCESS {
                                    retbuf.map(|buf| {
                                        self.replace_buffer(buf);
                                    });
                                }
                            });
                        }
                    }
                } else {
                    // we need to get more samples from the current app_buffer
                    perform_callback = false;

                    // provide a new buffer and update state
                    self.take_and_map_buffer(|adc_buf| {
                        let request_len = cmp::min(self.samples_remaining.get(), adc_buf.len());
                        self.samples_remaining
                            .set(self.samples_remaining.get() - request_len);
                        self.samples_outstanding
                            .set(self.samples_outstanding.get() + request_len);
                        let (res, retbuf) = self.adc.provide_buffer(adc_buf, request_len);
                        if res != ReturnCode::SUCCESS {
                            retbuf.map(|buf| {
                                self.replace_buffer(buf);
                            });
                        }
                    });
                }

                // next we should copy bytes to the app buffer
                app_buf.map(move |app_buf| {
                    // copy bytes to app buffer
                    // first, regain ownership of the buffer and then iterate
                    // over the data
                    self.replace_buffer(buf).map(|adc_buf| {
                        // The `for` commands:
                        //  * `chunks_mut`: get sets of two bytes from the app
                        //                  buffer
                        //  * `skip`: skips the already written bytes from the
                        //            app buffer
                        //  * `zip`: ties that iterator to an iterator on the
                        //           adc buffer, limiting iteration length to
                        //           the minimum of each of their lengths
                        //  * `take`: limits us to the minimum of buffer lengths
                        //            or sample length
                        // We then split each sample into its two bytes and copy
                        // them to the app buffer
                        for (chunk, &sample) in app_buf
                            .chunks_mut(2)
                            .skip(self.app_buf_offset.get() / 2)
                            .zip(adc_buf.iter())
                            .take(length)
                        {
                            let mut val = sample;
                            for byte in chunk.iter_mut() {
                                *byte = (val & 0xFF) as u8;
                                val = val >> 8;
                            }
                        }
                    });

                    // update our byte offset based on how many samples we
                    // copied
                    self.app_buf_offset
                        .set(self.app_buf_offset.get() + length * 2);
                });

                // if the app_buffer is filled, perform callback
                if perform_callback {
                    // actually schedule the callback
                    self.callback.map(|callback| {
                        let len_chan = ((app_buf.len() / 2) << 8) | (self.channel.get() & 0xFF);
                        callback.schedule(
                            self.mode.get() as usize,
                            len_chan,
                            app_buf.ptr() as usize,
                        );
                    });

                    // if the mode is SingleBuffer, the operation is
                    // complete. Clean up state
                    if self.mode.get() == AdcMode::SingleBuffer {
                        self.active.set(false);
                        self.mode.set(AdcMode::NoMode);
                        self.app_buf_offset.set(0);

                        // need to actually stop sampling
                        self.adc.stop_sampling();

                        // reclaim buffers and store them
                        let (_, buf1, buf2) = self.adc.retrieve_buffers();
                        buf1.map(|buf| {
                            self.replace_buffer(buf);
                        });
                        buf2.map(|buf| {
                            self.replace_buffer(buf);
                        });
                    } else {
                        // if the mode is ContinuousBuffer, we've just
                        // switched app buffers. Reset our offset to zero
                        self.app_buf_offset.set(0);
                    }
                }
            });
        } else {
            // operation was likely canceled. Make sure state is consistent. No
            // callback
            self.active.set(false);
            self.mode.set(AdcMode::NoMode);
            self.app_buf_offset.set(0);

            // still need to replace the buffer
            self.replace_buffer(buf);
        }
    }
}

/// Implementations of application syscalls
impl<A: hil::adc::Adc + hil::adc::AdcHighSpeed> Driver for Adc<'a, A> {
    /// Provides access to a buffer from the application to store data in or
    /// read data from
    ///
    /// _appid - application identifier, unused
    /// allow_num - which allow call this is
    /// slice - representation of application memory to copy data into
    fn allow(
        &self,
        _appid: AppId,
        allow_num: usize,
        slice: Option<AppSlice<Shared, u8>>,
    ) -> ReturnCode {
        match allow_num {
            // Pass buffer for samples to go into
            0 => {
                // set first buffer
                self.app.map(|state| {
                    state.app_buf1 = slice;
                });

                ReturnCode::SUCCESS
            }

            // Pass a second buffer to be used for double-buffered continuous sampling
            1 => {
                // set second buffer
                self.app.map(|state| {
                    state.app_buf2 = slice;
                });

                ReturnCode::SUCCESS
            }

            // default
            _ => ReturnCode::ENOSUPPORT,
        }
    }

    /// Provides a callback which can be used to signal the application
    ///
    /// subscribe_num - which subscribe call this is
    /// callback - callback object which can be scheduled to signal the
    ///            application
    fn subscribe(
        &self,
        subscribe_num: usize,
        callback: Option<Callback>,
        _app_id: AppId,
    ) -> ReturnCode {
        match subscribe_num {
            // subscribe to ADC sample done (from all types of sampling)
            0 => {
                // set callback
                self.callback.insert(callback);
                ReturnCode::SUCCESS
            }

            // default
            _ => ReturnCode::ENOSUPPORT,
        }
    }

    /// Method for the application to command or query this driver
    ///
    /// command_num - which command call this is
    /// data - value sent by the application, varying uses
    /// _appid - application identifier, unused
    fn command(
        &self,
        command_num: usize,
        channel: usize,
        frequency: usize,
        _appid: AppId,
    ) -> ReturnCode {
        match command_num {
            // check if present
            0 => ReturnCode::SuccessWithValue {
                value: self.channels.len() as usize,
            },

            // Single sample on channel
            1 => self.sample(channel),

            // Repeated single samples on a channel
            2 => self.sample_continuous(channel, frequency as u32),

            // Multiple sample on a channel
            3 => self.sample_buffer(channel, frequency as u32),

            // Continuous buffered sampling on a channel
            4 => self.sample_buffer_continuous(channel, frequency as u32),

            // Stop sampling
            5 => self.stop_sampling(),

            // default
            _ => ReturnCode::ENOSUPPORT,
```

Benefits - Reduced Audit Space

Tock ADC Component:

```
impl AdcComponent {
    pub fn new() -> AdcComponent {
        AdcComponent {}
    }
}

impl Component for AdcComponent {
    type Output = &'static adc::Adc<'static, sam4l::adc::Adc>;

    unsafe fn finalize(&mut self) -> Self::Output {
        let adc_channels = static_init!(
            [&'static sam4l::adc::AdcChannel; 6],
            [
                &sam4l::adc::CHANNEL_AD1, // AD0
                &sam4l::adc::CHANNEL_AD2, // AD1
                &sam4l::adc::CHANNEL_AD3, // AD2
                &sam4l::adc::CHANNEL_AD4, // AD3
                &sam4l::adc::CHANNEL_AD5, // AD4
                &sam4l::adc::CHANNEL_AD6, // AD5
            ]
        );
        let adc = static_init!(
            adc::Adc<'static, sam4l::adc::Adc>,
            adc::Adc::new(
                &mut sam4l::adc::ADC0,
                adc_channels,
                &mut adc::ADC_BUFFER1,
                &mut adc::ADC_BUFFER2,
                &mut adc::ADC_BUFFER3
            )
        );
        sam4l::adc::ADC0.set_client(adc);

        adc
    }
}
```

Benefits - Reduced Audit Space

Tock UDP Networking Capsule :

- 12 Files, 4300 Lines of Code

Tock ADC Capsule:

- 850 Lines of Code

Tock UDP Component:

- 142 Lines of code

Tock ADC Component:

- 33 Lines of Code

Benefits - Explicit Security

- Capsule only compiles if it is given all the capabilities it needs
- These capabilities are all in line with the capsule's instantiation
- Capability distribution can be viewed directly alongside the relationships between capsules
- Capabilities are superior to ACLs for handling the confused deputy problem
  - "The Confused Deputy (or why capabilities may have been invented)" - http://www.isti.tu-berlin.de/fileadmin/fg214/-368094-Pallavi_Jagannatha_confused_deputy.pdf

Open Problems / Next Steps

- Study the impact dynamic permissions have on efficiency/code size
- How to maintain separation of layers?
  - E.g. my UDP layer shouldn’t know anything about my permissions at the link layer
- Confused Deputy Problem
  - How do we know that capabilities won’t be shared beyond their intended use?
- Revocable Permissions
- Is expressing everything in the type system too restrictive?

Conclusion

- Security we want on IoT devices is very different than on POSIX
  - No users, no files → how should we express security and access policies?
  - Instead, may be the case that different components of code should have different abilities
  - Cannot interact with user to determine how to react in ambiguous scenarios
  - Should communication be default on or default off?
- IoT security models must be restrictive by default
  - Developers will follow path of least resistance
- IoT devices are infrequently updated, so security audits are even more important than on traditional systems
  - Accordingly, making audits easier is critical
  - Adding a single new library should not require auditing the entire system again
- Capabilities implemented using the type system provide a unique path towards achieving these goals at low cost