Microsoft

Synchronization in Tock OS and Pluton

Bobby Reynolds
Microsoft Pluton
September 5, 2025

Bobby Reynolds

Senior Engineering Manager

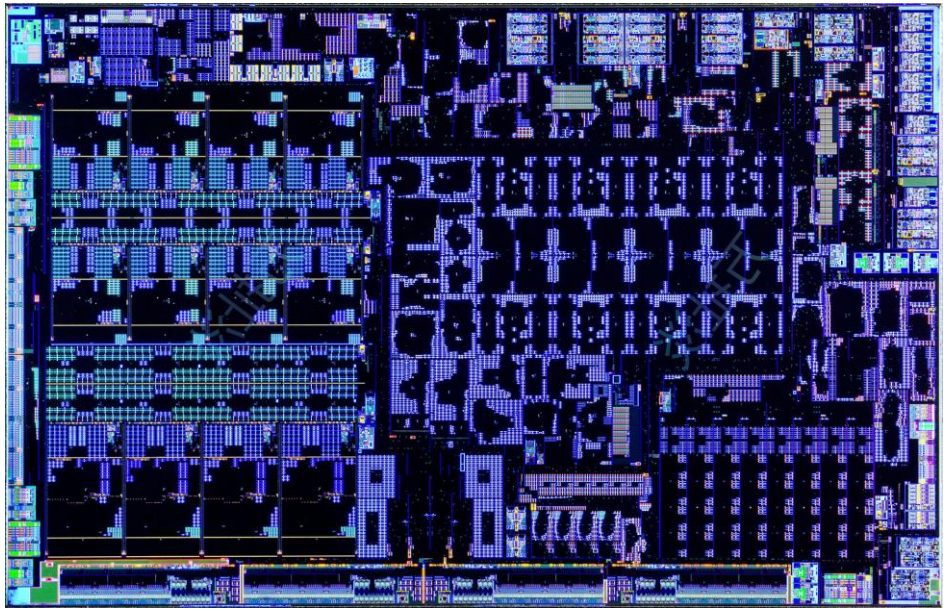
Pluton – 2022-Present

- Leads development of “Pluton OS”, an internal derivative of Tock used across the portfolio of Pluton devices.
- Works with IHV partners to define Pluton requirements and realize end-to-end product integration.

Mixed Reality – 2019-2022

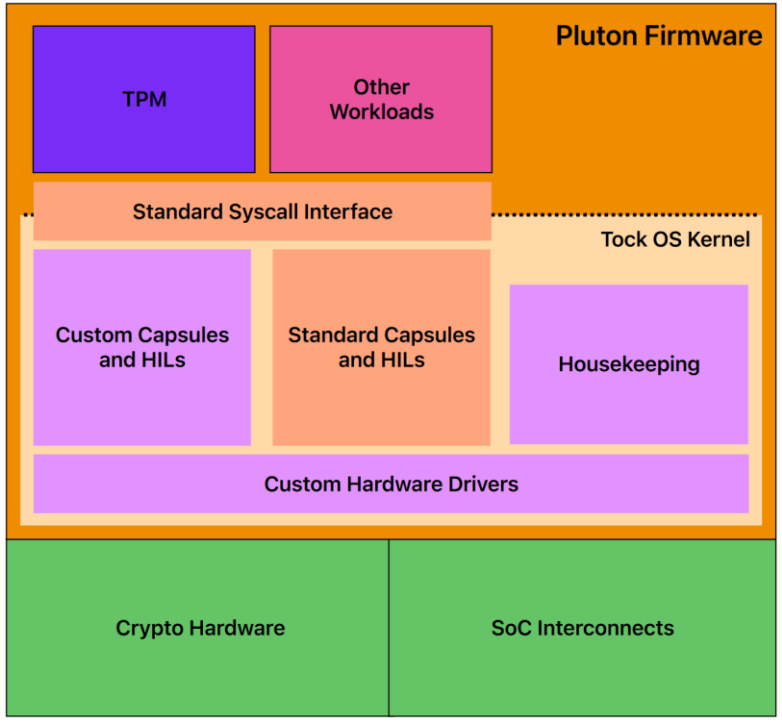
- Embedded firmware engineer for HoloLens 2, IVAS, and other related devices

About Pluton



- On-die security coprocessor
- Jointly developed by Microsoft and IHV partners
- Firmware developed and serviced by Microsoft

- Tock OS at the core of every Pluton firmware image
- Some bolted-on additions, such as crypto drivers
- Apps provide services to host OS, such as TPM and KSP

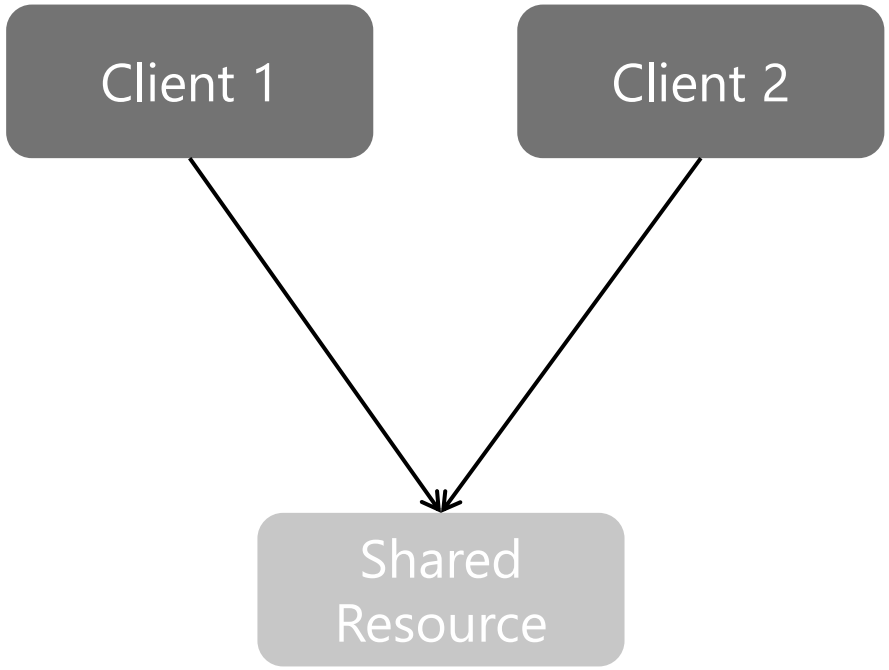


Agenda

- Introduction
- Virtualizer Pattern
- Mutex Variations
- Driver Commands

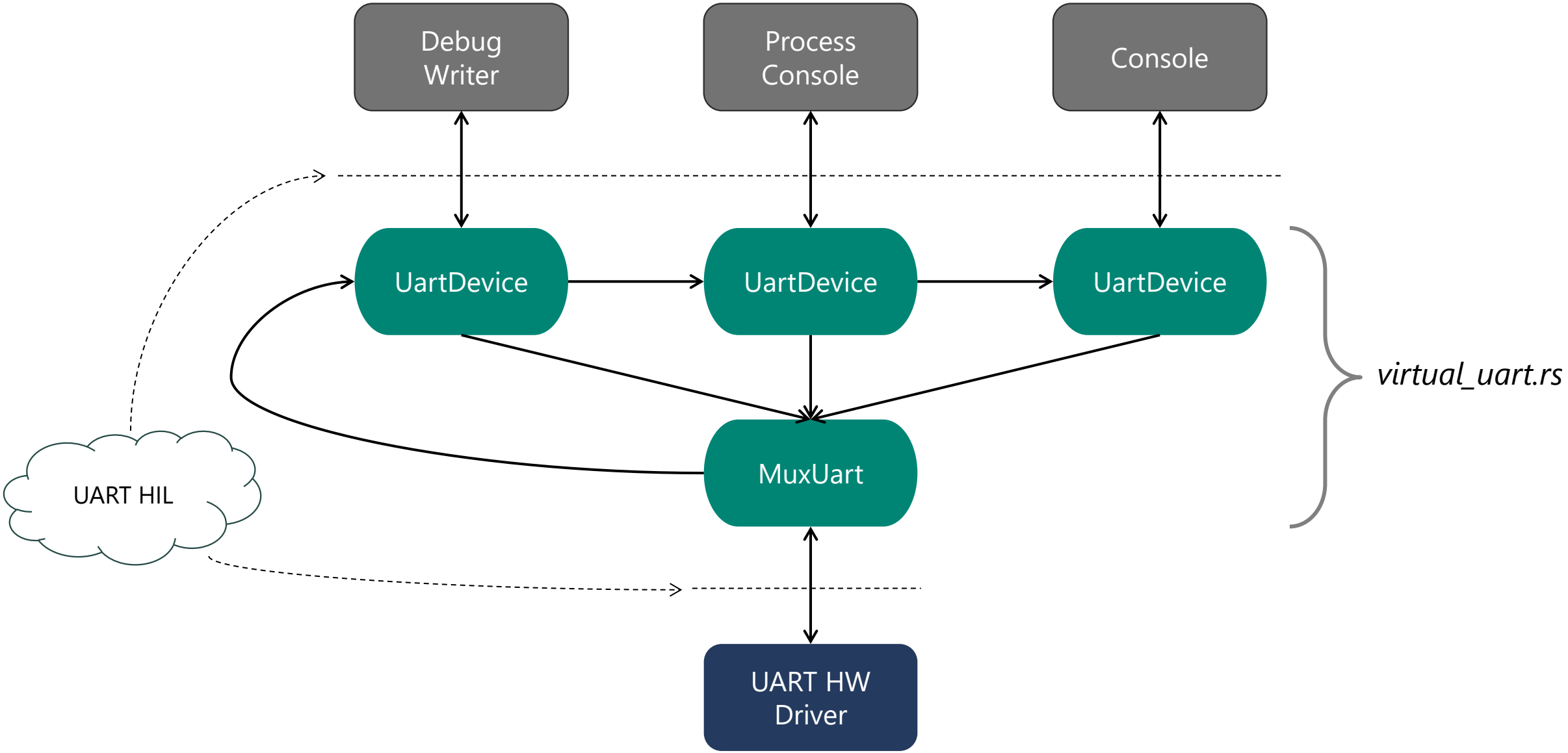
Problem Statement

- Classical CS problem
- Still relevant for Tock
  - More relevant for us?
- No BUSY or retry loops



Virtualizer Pattern

Case Study 0: UART Virtualizer



Virtualizer Pattern Analysis

Pros

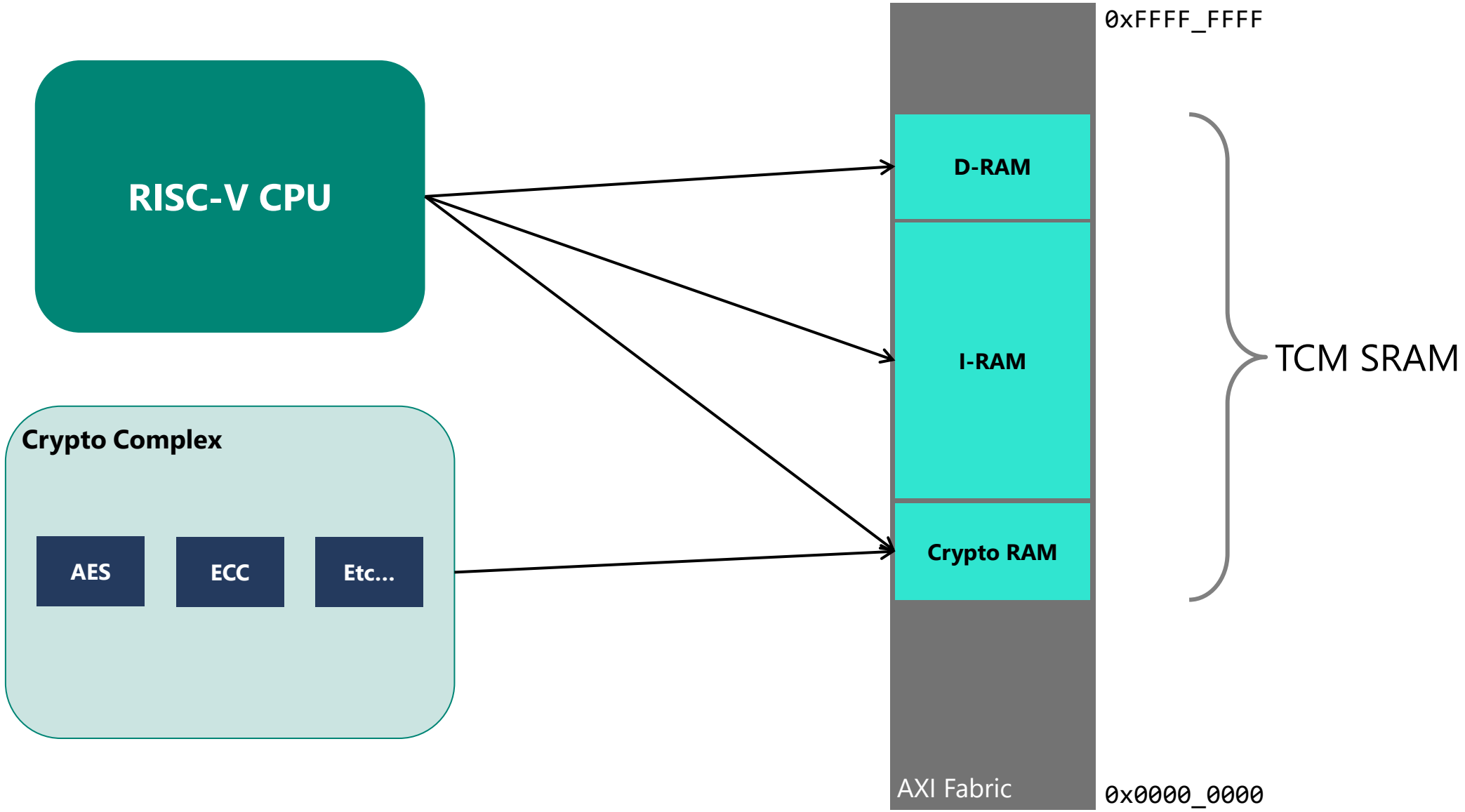
- Transparent to clients
- Tailored to specific data or usage pattern of device in question

Cons

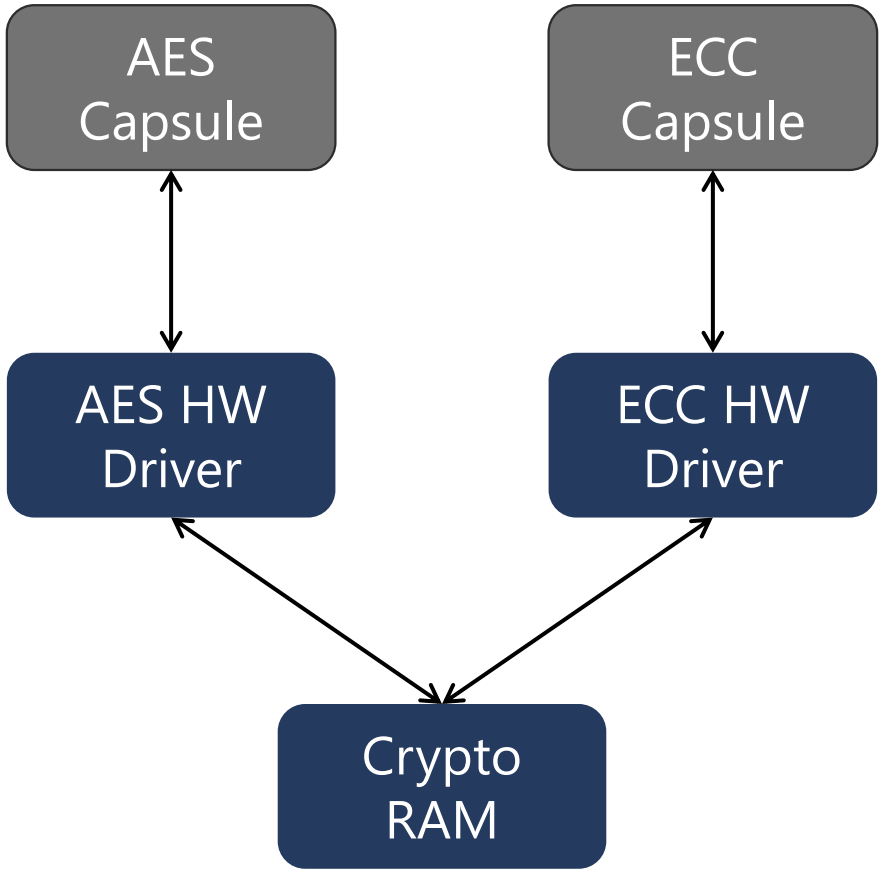
- Tedious to write and maintain
- Potentially larger code footprint

Mutex Primitives

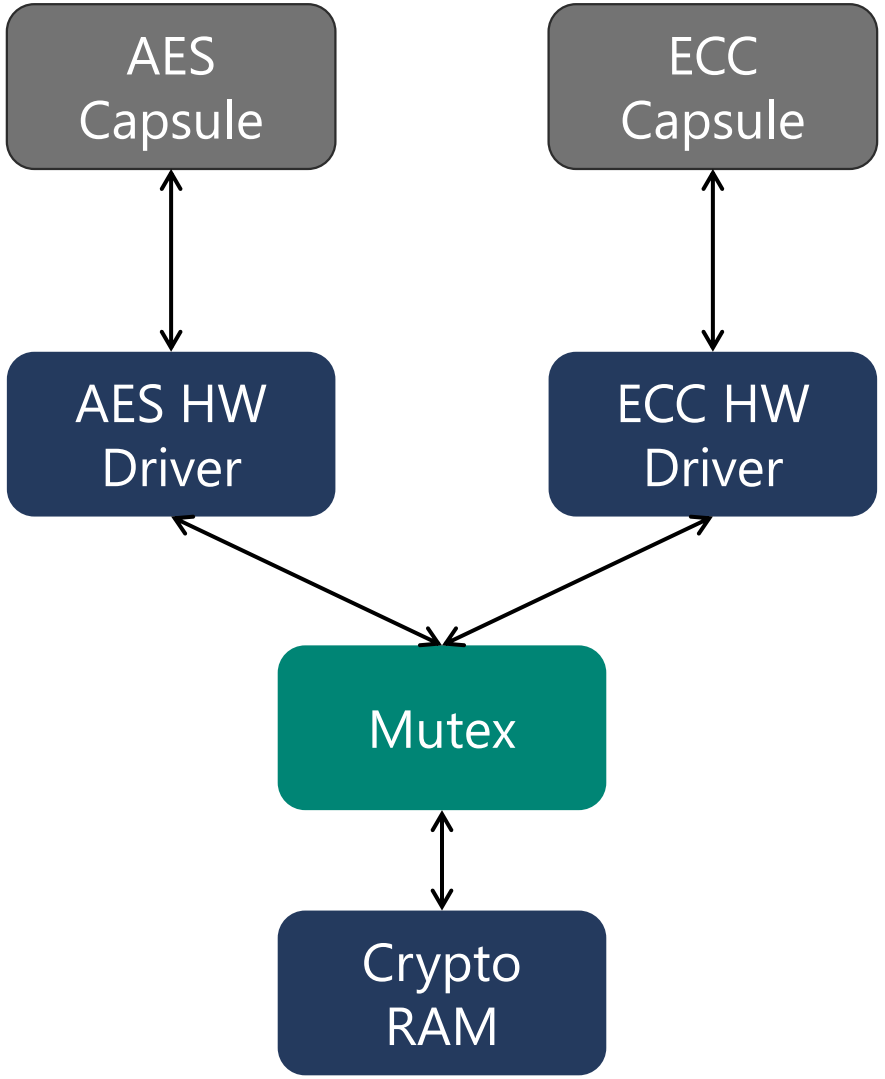
Case Study 1: Crypto RAM



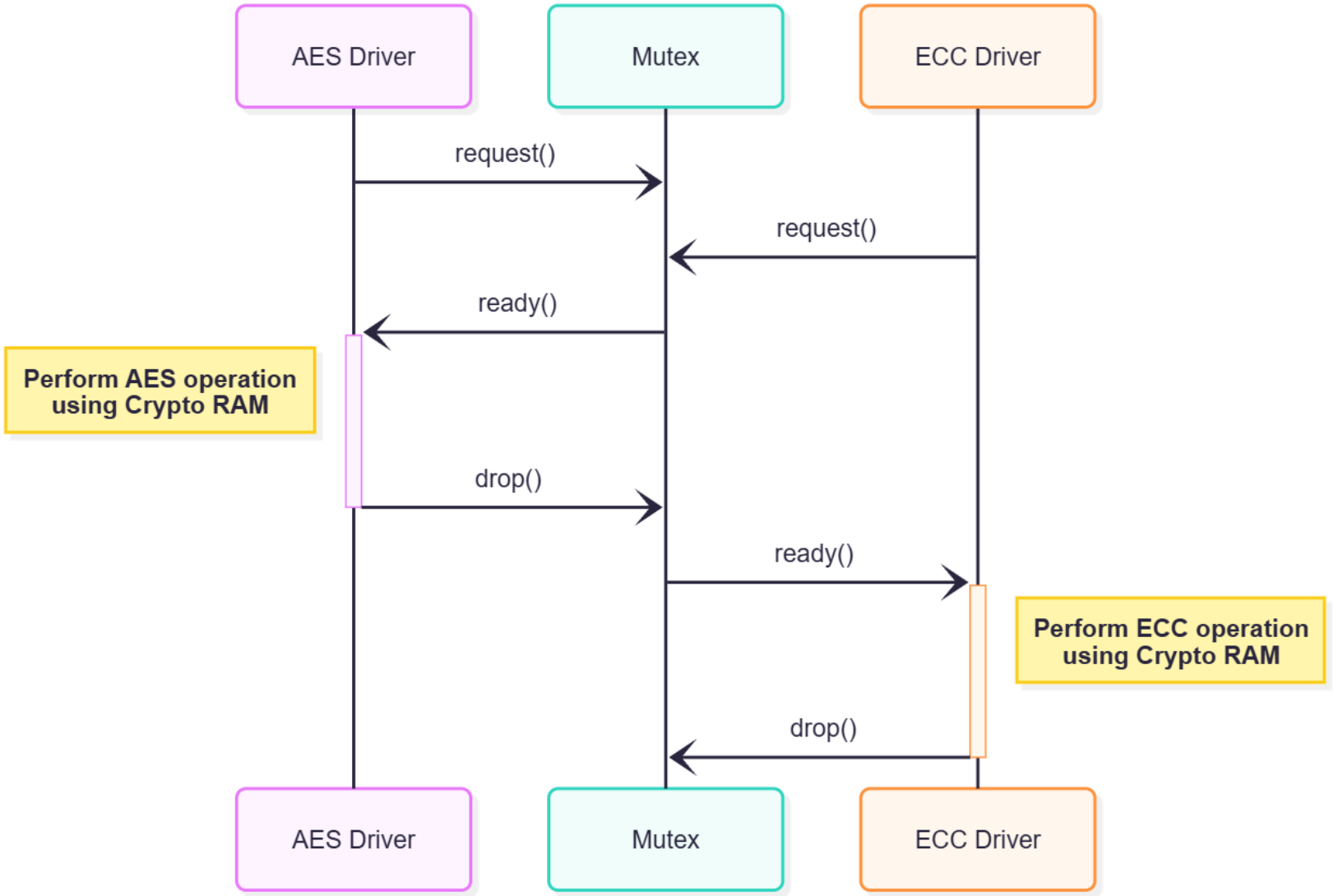
Case Study 1: Driver Architecture



Mutex<T>


AES
Capsule
ECC
Capsule
AES HW
Driver
ECC HW
Driver
Mutex
Crypto
RAM

Mutex Sequence Diagram



Mutex Usage Pseudocode

```
struct AesDriver {
    ram_mx: &Mutex<CryptoRam>,
    // ...
}

impl Aes for AesDriver {
    fn encrypt(&self, /* params */) {
        self.state.set(State::WaitForCryptoRam(/* params */));
        self.ram_mx.request(self);
    }
}

impl MutexClient<CryptoRam> for AesDriver {
    fn ready(&self, mut ram: MutexGuard<CryptoRam>) {
        match &*self.state.borrow() {
            State::WaitForCryptoRam(params) => {
                self.write_enc_params(params, &mut ram);
                self.state.set(State::Encrypting(ram));
                self.begin_enc_hw_op();
            }

            _ => panic!("unexpected mutex callback"),
        }
    }
}

impl AesDriver {
    fn handle_interrupt(&self) {
        self.state.replace(State::Idle);
        self.client.encryption_done();
    }
}
```

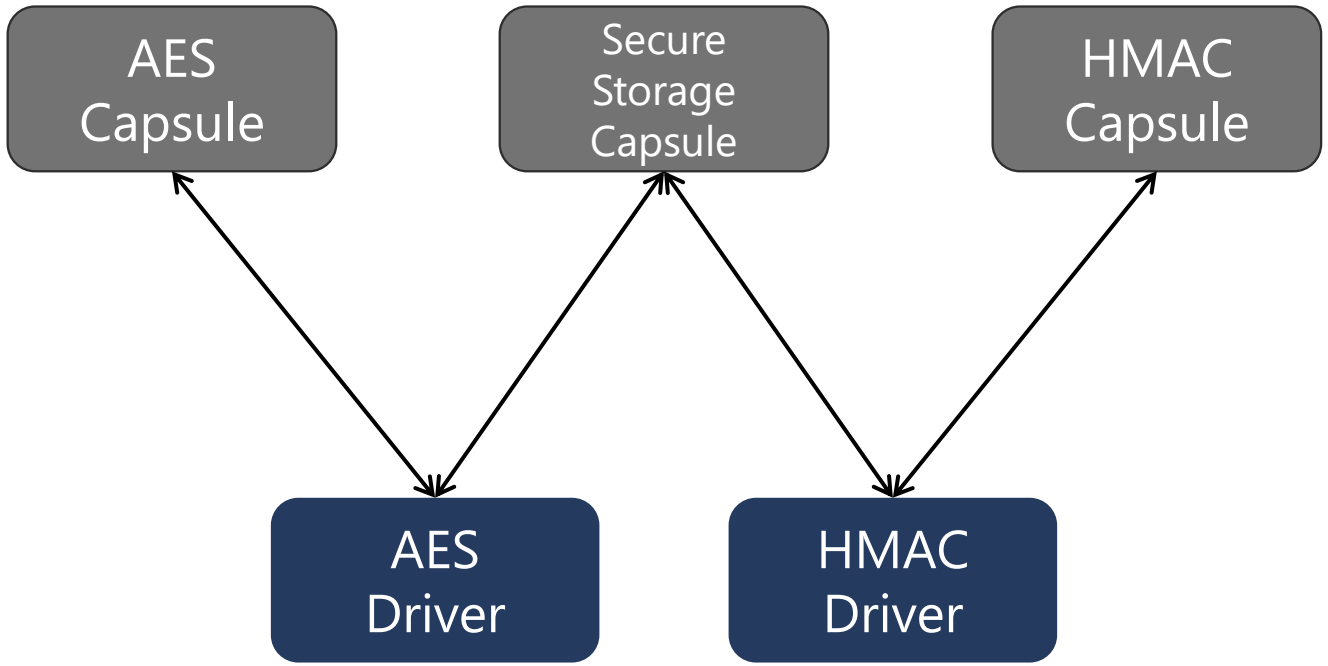
1

2

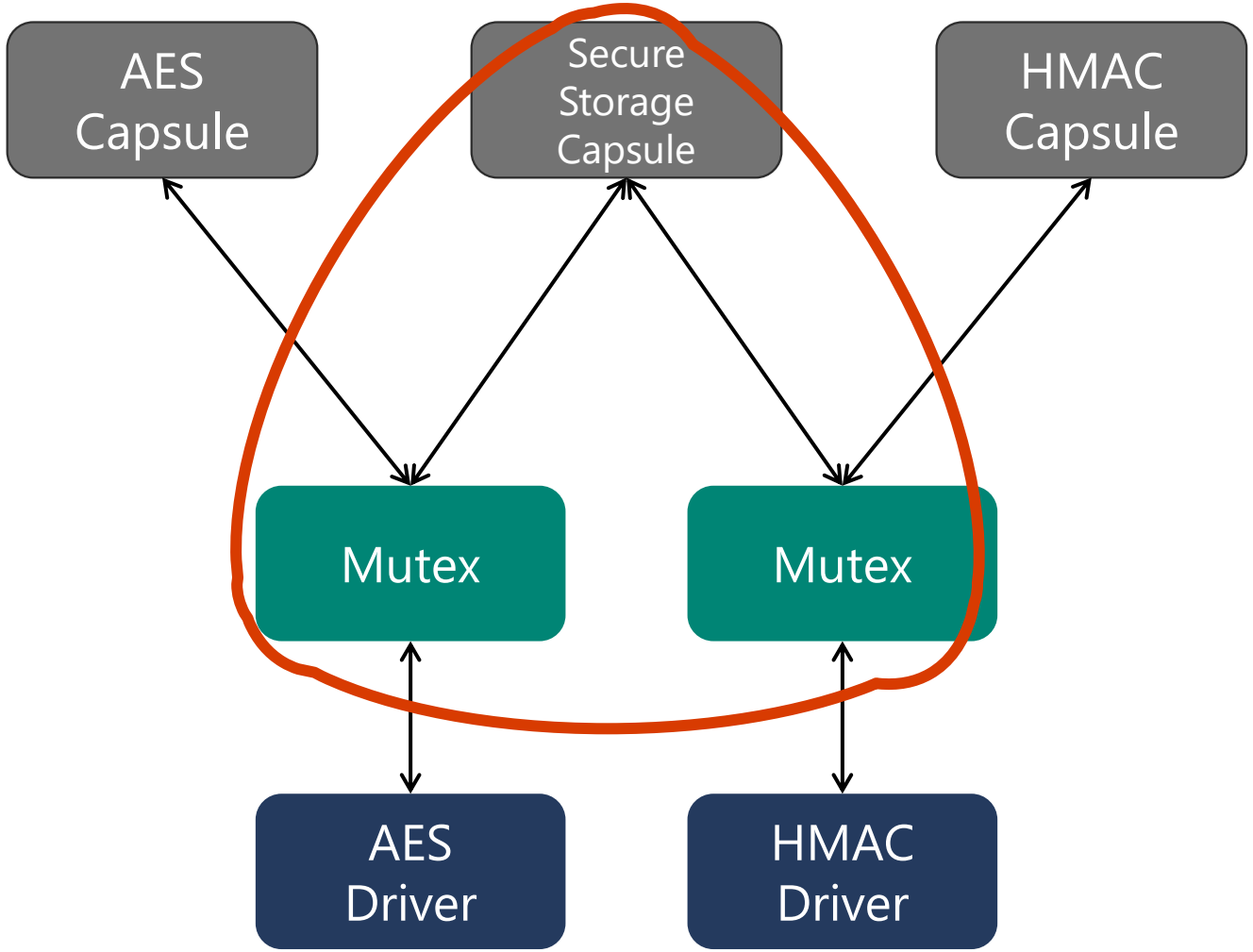
3

RAII Guard

Case Study 2: Storage Capsule



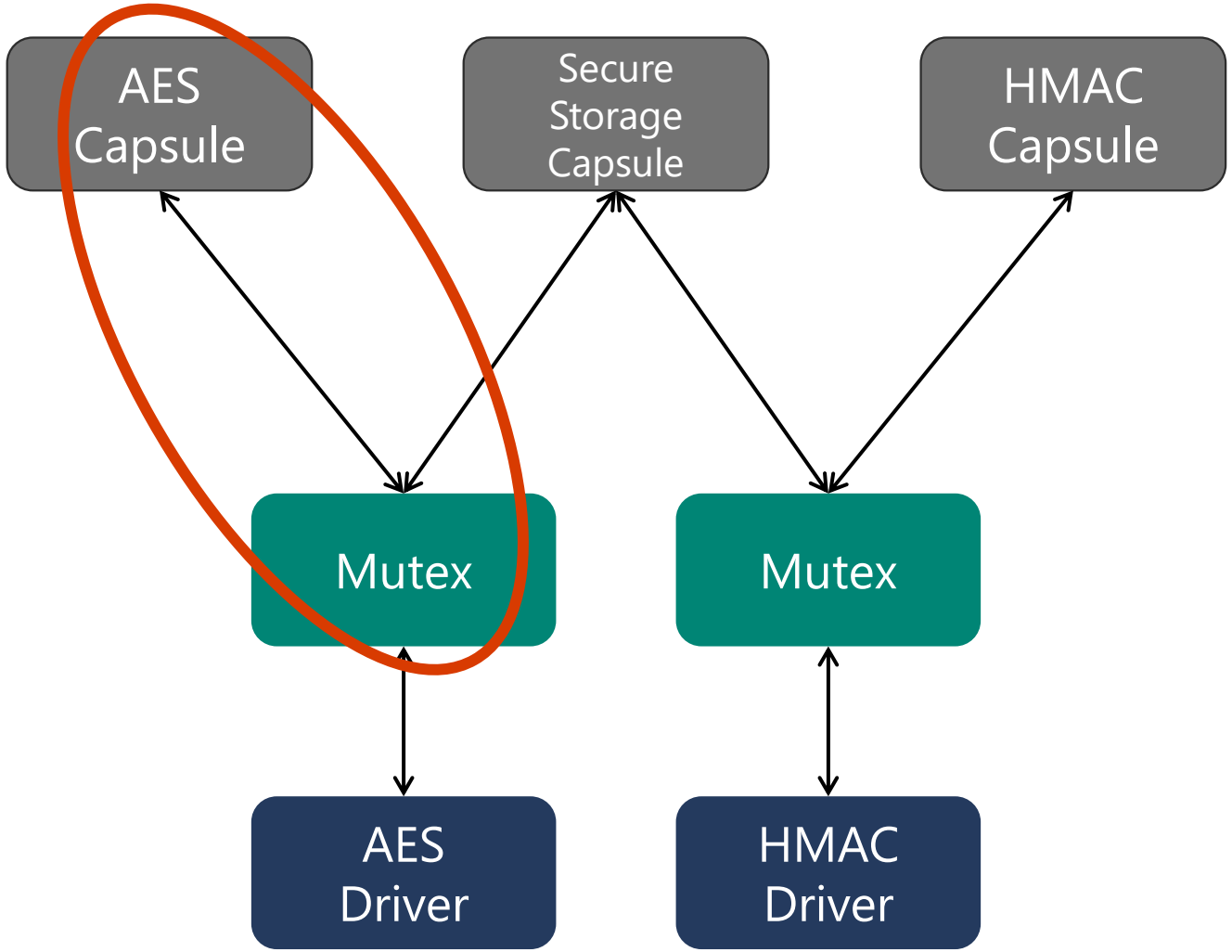
Case Study 2: Storage Capsule



Multiple Mutexes

```
struct StorageCapsule {
    aes_mx: &Mutex<AesDriver>,
    hmac_mx: &Mutex<HmacDriver>,
}

impl MutexClient<AesDriver> for StorageCapsule {
    fn ready(&self, aes: MutexGuard<AesDriver>) {
        aes.set_client(self);
        aes.encrypt(/* params */);
    }
}

impl MutexClient<HmacDriver> for StorageCapsule {
    fn ready(&self, hmac: MutexGuard<HmacDriver>) {
        hmac.set_client(self);
        hmac.create(/* params */);
    }
}
```

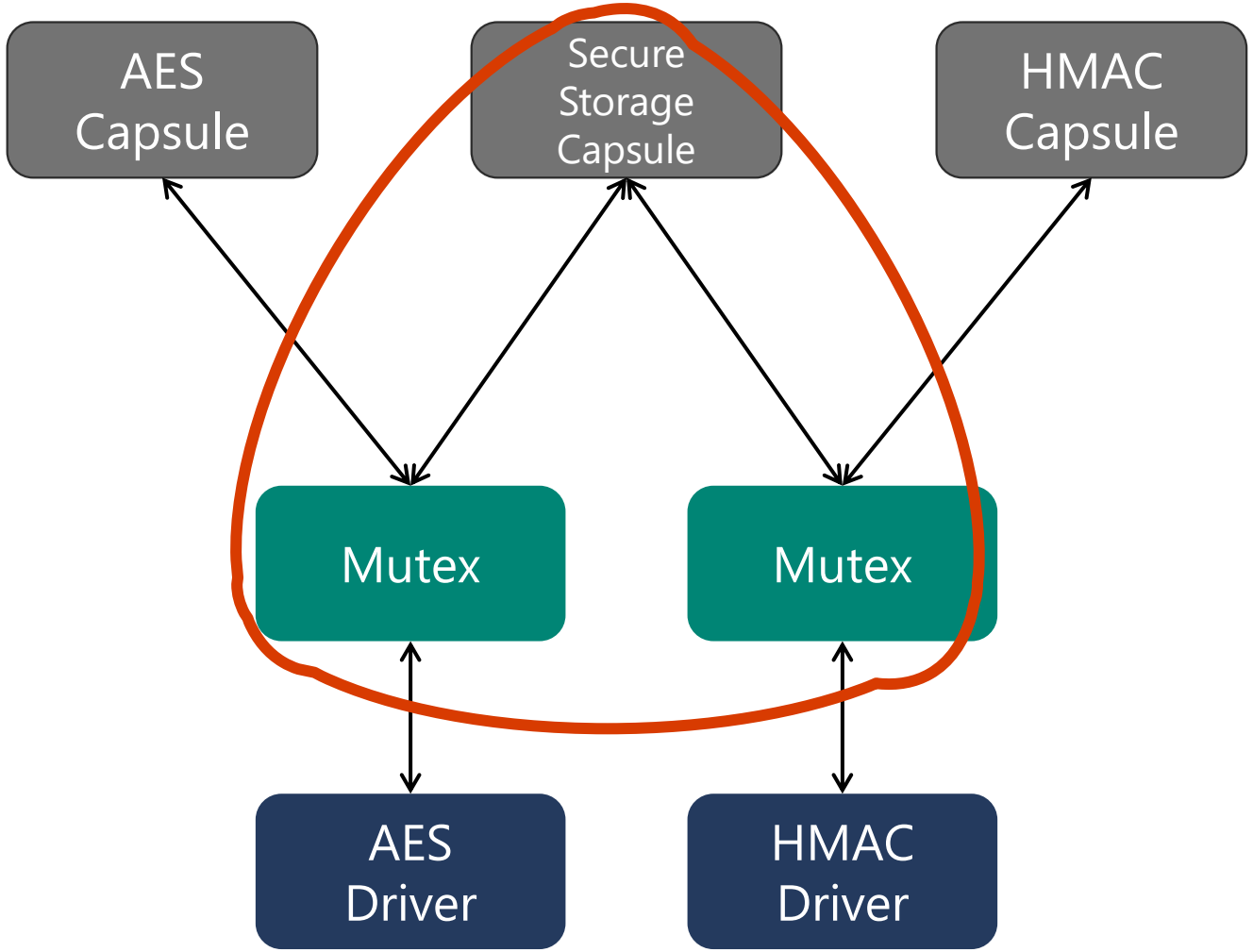
Case Study 2: Storage Capsule



Generic Mutex

```
struct AesCapsule<A: Aes> {
    aes_mx: &Mutex<A>,
}

impl<A: Aes> MutexClient<A> for AesCapsule<A> {
    fn ready(&self, aes: MutexGuard<A>) {
        aes.set_client(self);
        aes.encrypt(/* params */);
    }
}
```

Case Study 2: Storage Capsule



Multiple Generic Mutexes

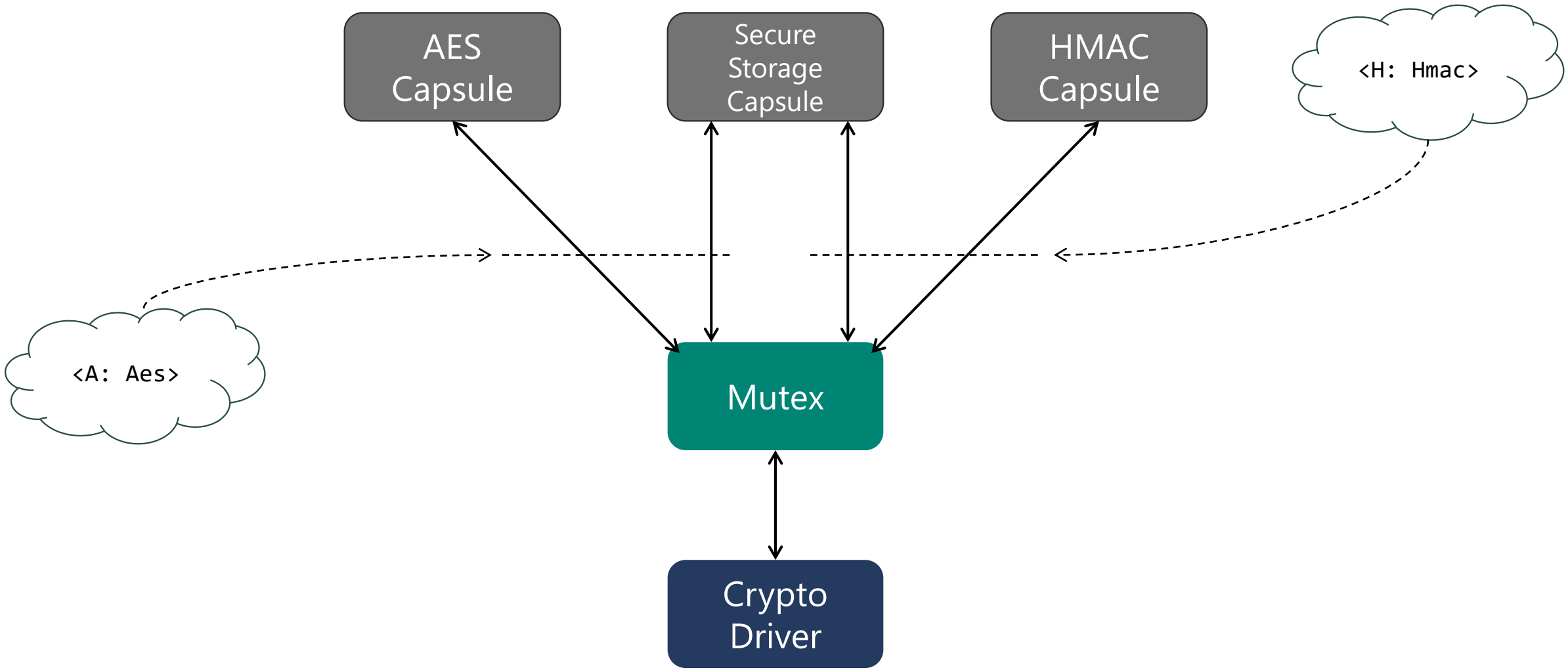
```rust
struct StorageCapsule<A: Aes, H: Hmac> {
    aes_mx: &Mutex<A>,
    hmac_mx: &Mutex<H>,
}

impl<A: Aes, H: Hmac> MutexClient<A> for StorageCapsule<A, H> {
    fn ready(&self, aes: MutexGuard<A>) {
        aes.set_client(self);
        aes.encrypt(/* params */);
    }
}

impl<A: Aes, H: Hmac> MutexClient<H> for StorageCapsule<A, H> {
    fn ready(&self, hmac: MutexGuard<H>) {
        hmac.set_client(self);
        hmac.create(/* params */);
    }
}
```

```
error[E0119]: conflicting implementations of trait `MutexClient<_>` for type `StorageCapsule<_, _>`
--> boards/mutex-test/src/foo.rs:22:1
|
14 | / impl<A: Aes, H: Hmac> MutexClient<A> for StorageCapsule<A, H>
|    |_____________________________________________- first implementation here
...
22 | / impl<A: Aes, H: Hmac> MutexClient<H> for StorageCapsule<A, H>
|    |_____________________________________________^ conflicting implementation for `StorageCapsule<_, _>`
```

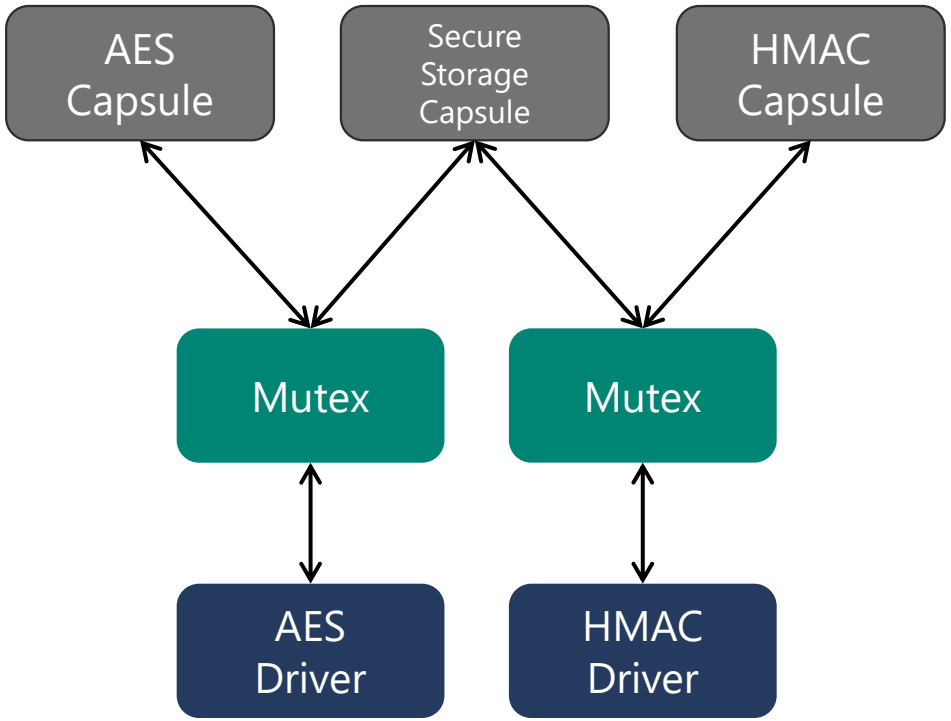
Multiple Generic Mutexes

```
struct Foo {
    // ...
}

impl Aes for Foo {
    // ...
}

impl Hmac for Foo {
    // ...
}

let s: StorageCapsule<Foo, Foo> = static_init!(/* ... */);
```

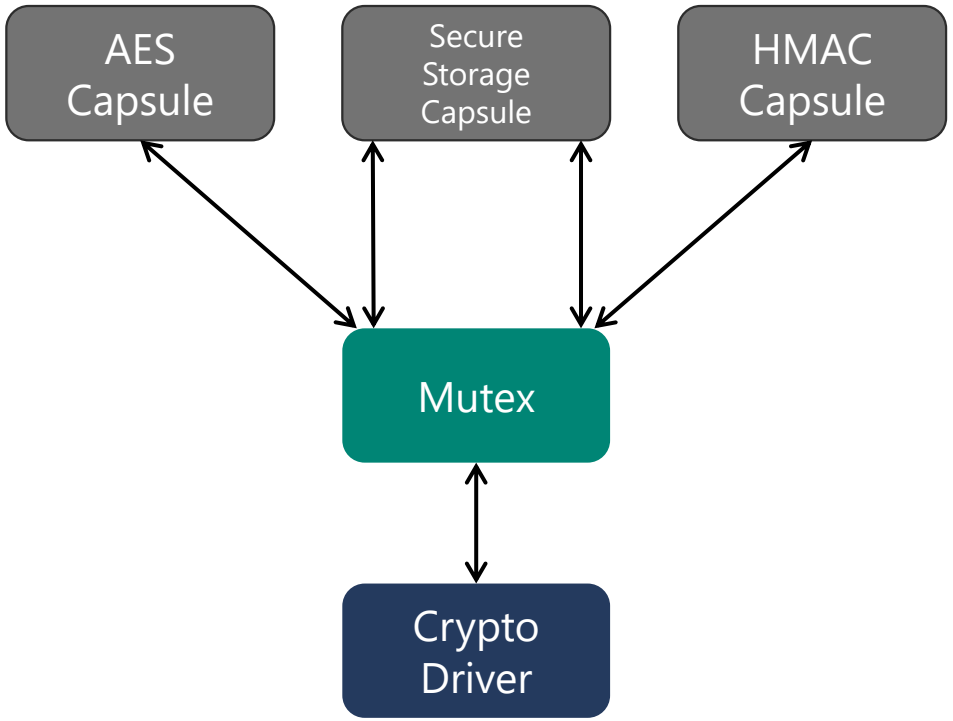
Case Study 2.1: Storage Capsule



Case Study 2.1: Storage Capsule



Platform A

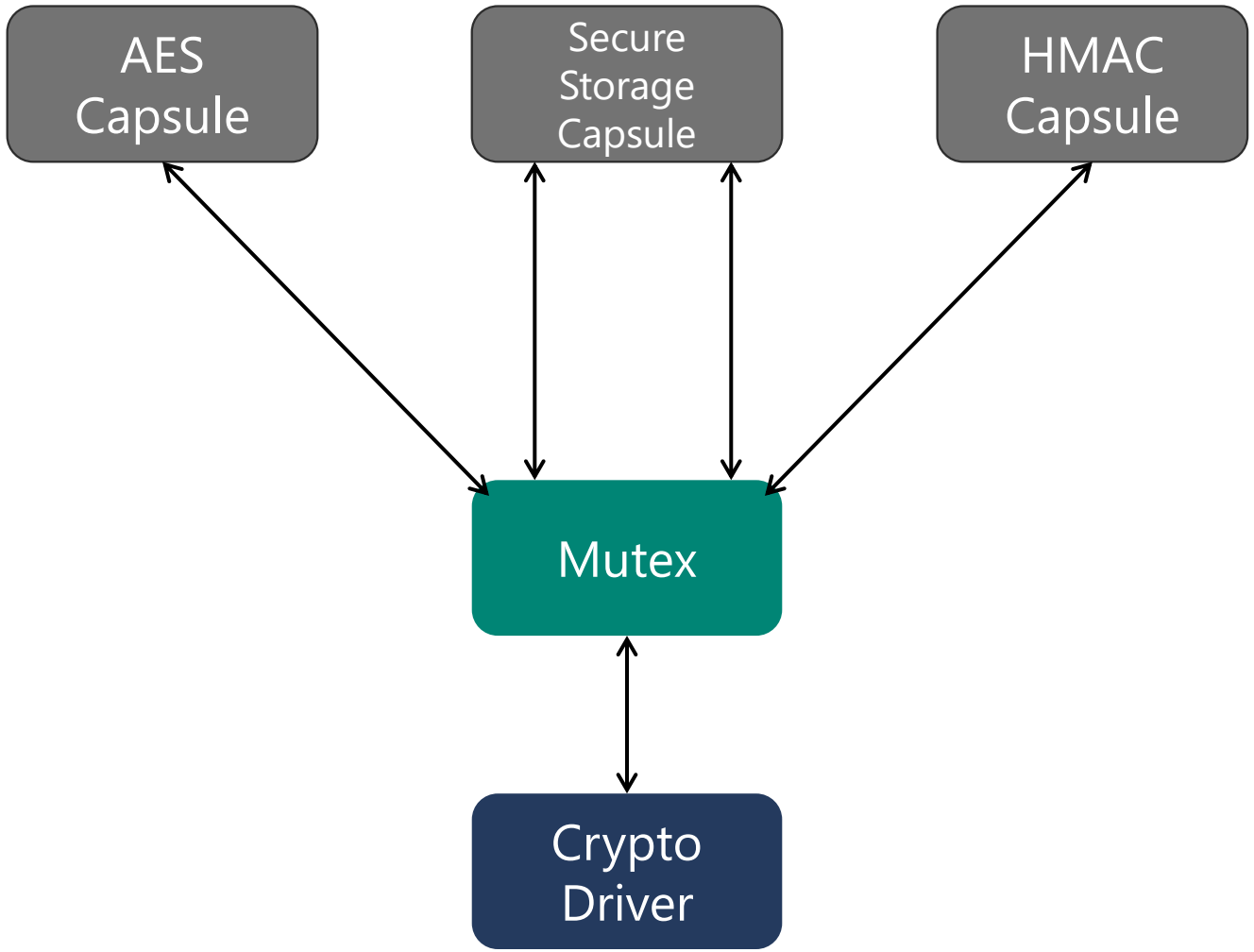


Platform B

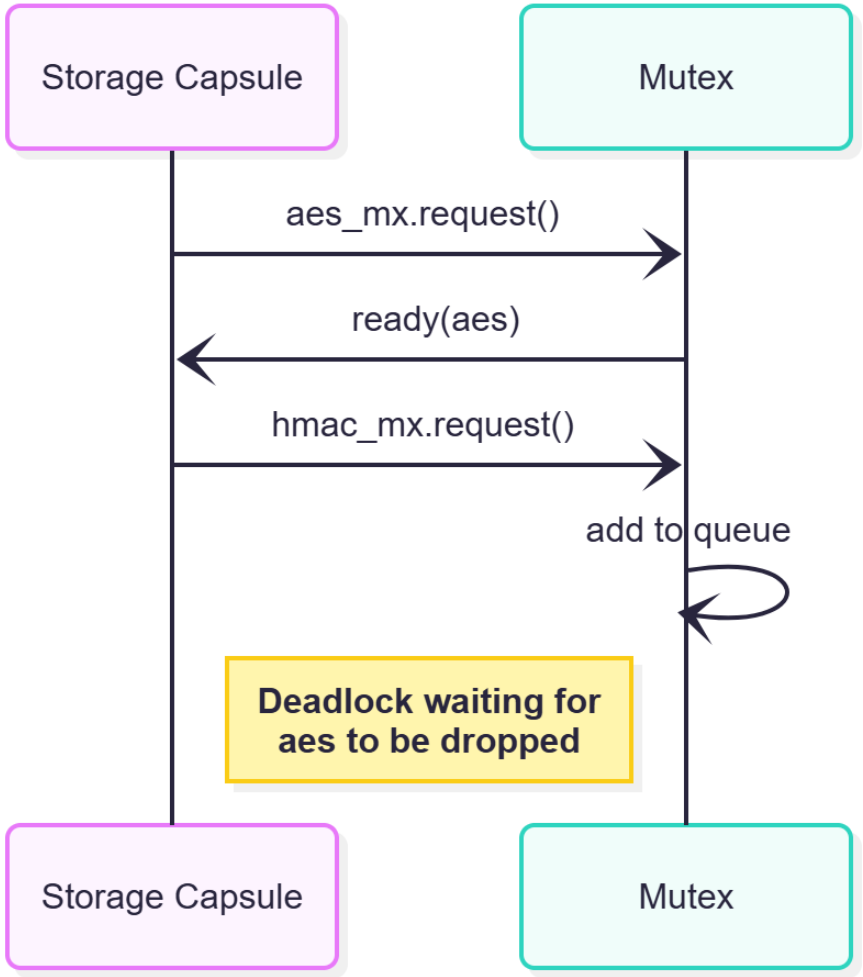
MutexGuard Downcasting

```
impl<A: Aes, H: Hmac> MutexClientAny for StorageCapsule<A, H> {
    fn ready(&self, resource: MutexGuardAny) {
        match &*self.state.borrow() {
            State::WaitForAes ⇒ {
                let Some(aes) = resource.downcast::<A>() else {
                    panic!("unexpected mutex type");
                };
                self.do_stuff_with_aes(aes);
            }

            State::WaitForHmac ⇒ {
                let Some(hmac) = resource.downcast::<H>() else {
                    panic!("unexpected mutex type");
                };
                self.do_stuff_with_hmac(hmac);
            }
        }
    }
}
```

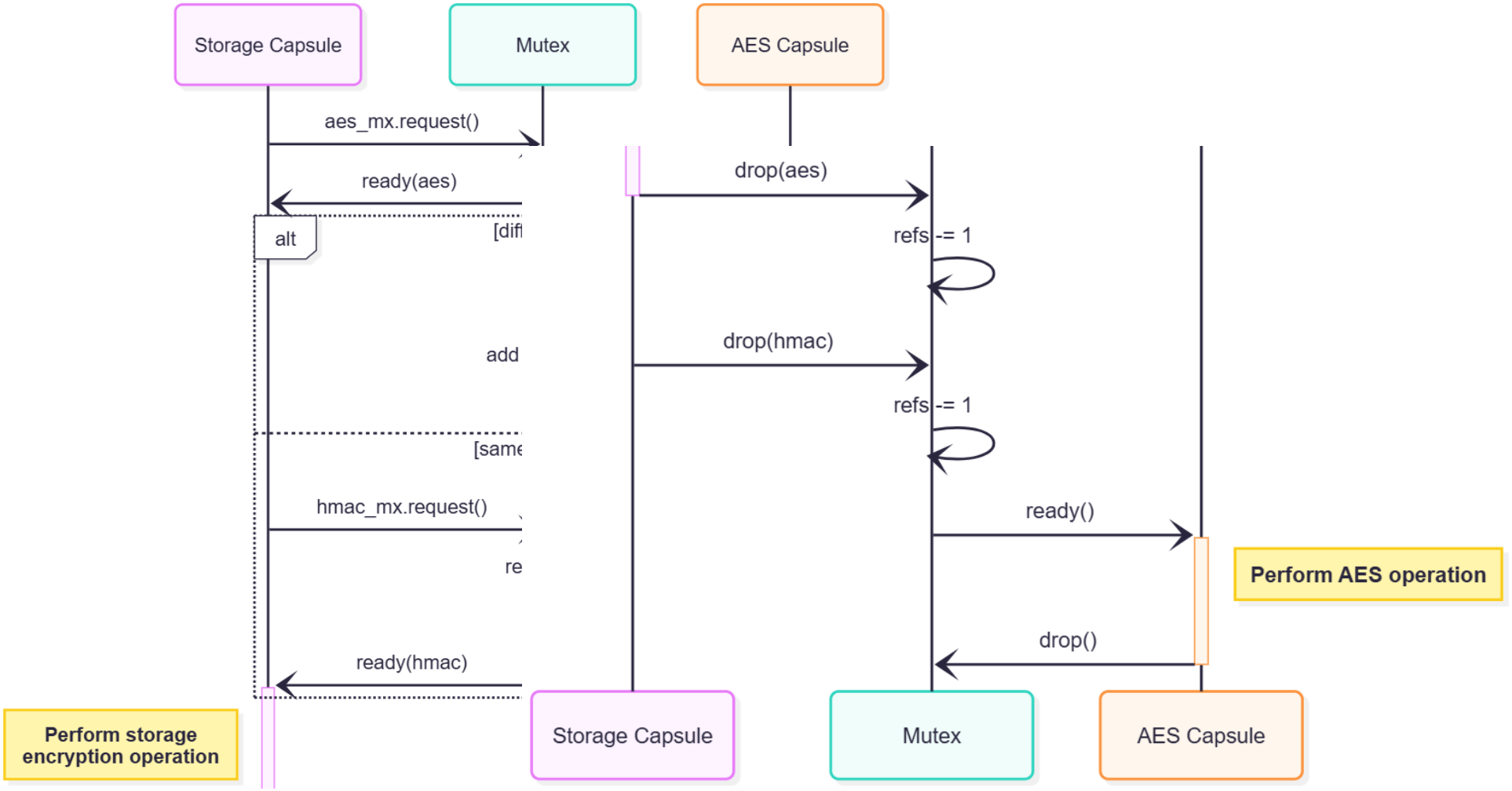
Case Study 2.1: Storage Capsule



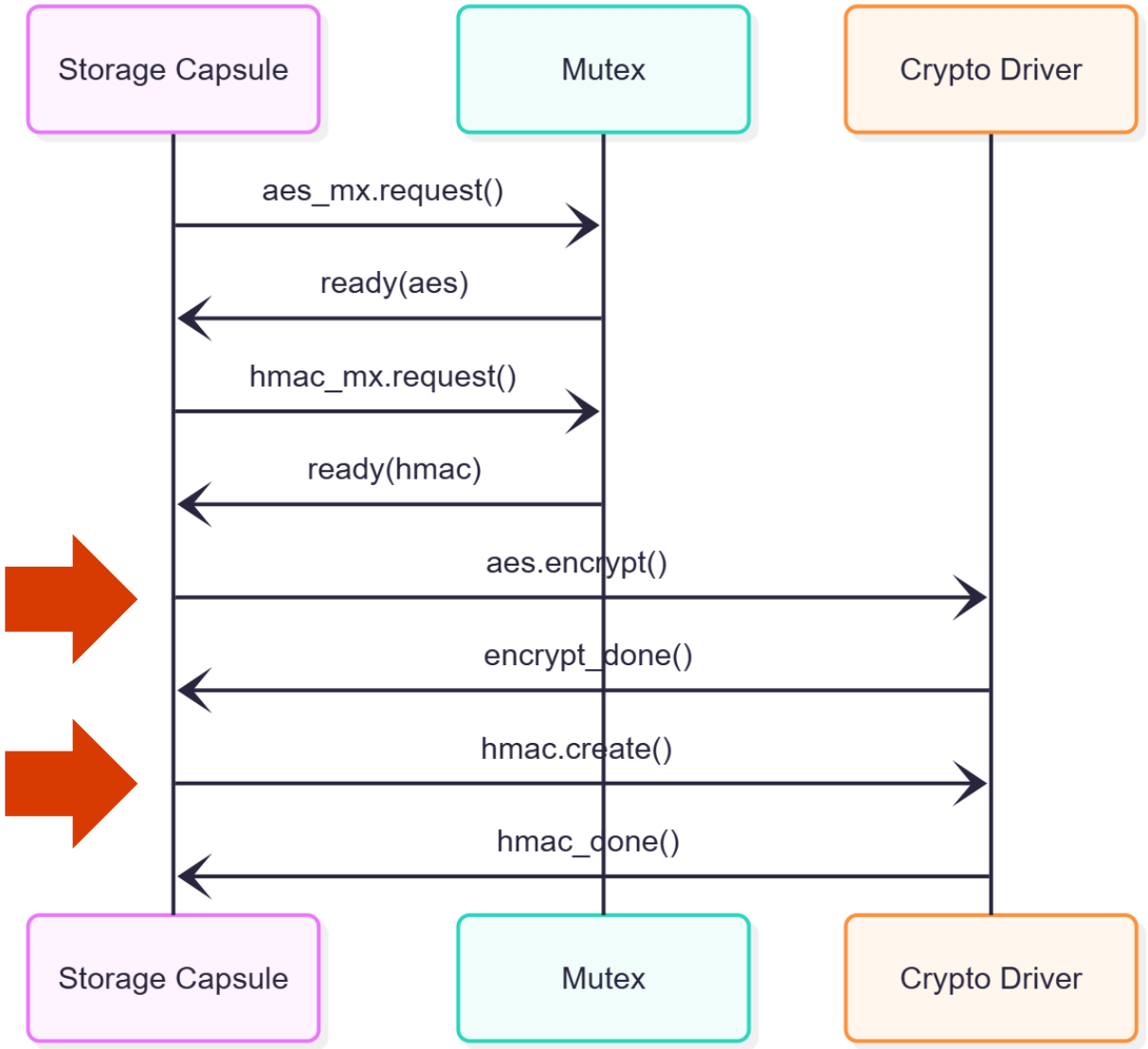
Mutex Deadlock



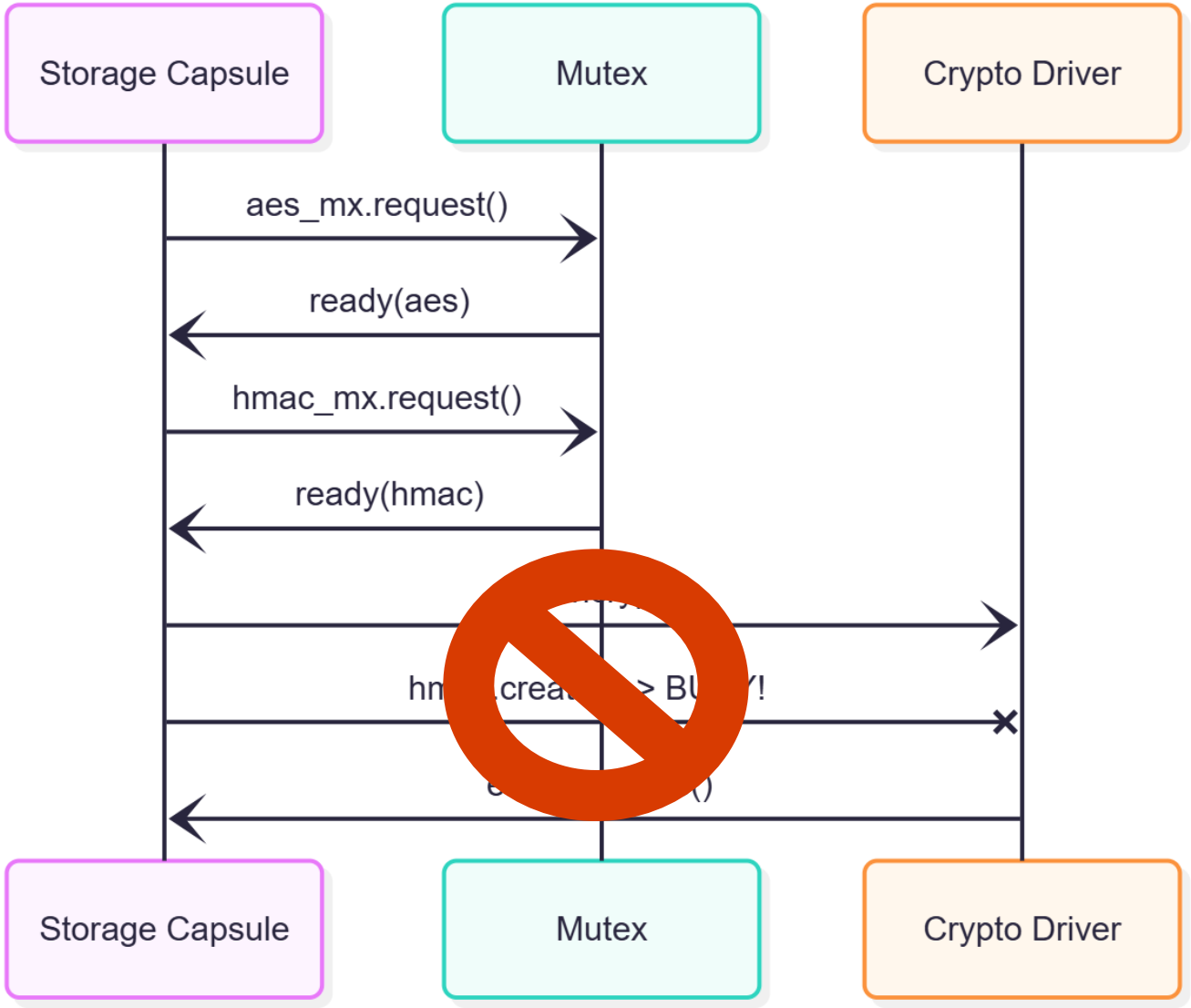
Reference Counting



Reference Counting



BUSY Hazard


Storage Capsule
Mutex
Crypto Driver
aes_mx.request()
ready(aes)
hmac_mx.request()
ready(hmac)
Storage Capsule
Mutex
Crypto Driver

Mutex Summary

Mutex<T>

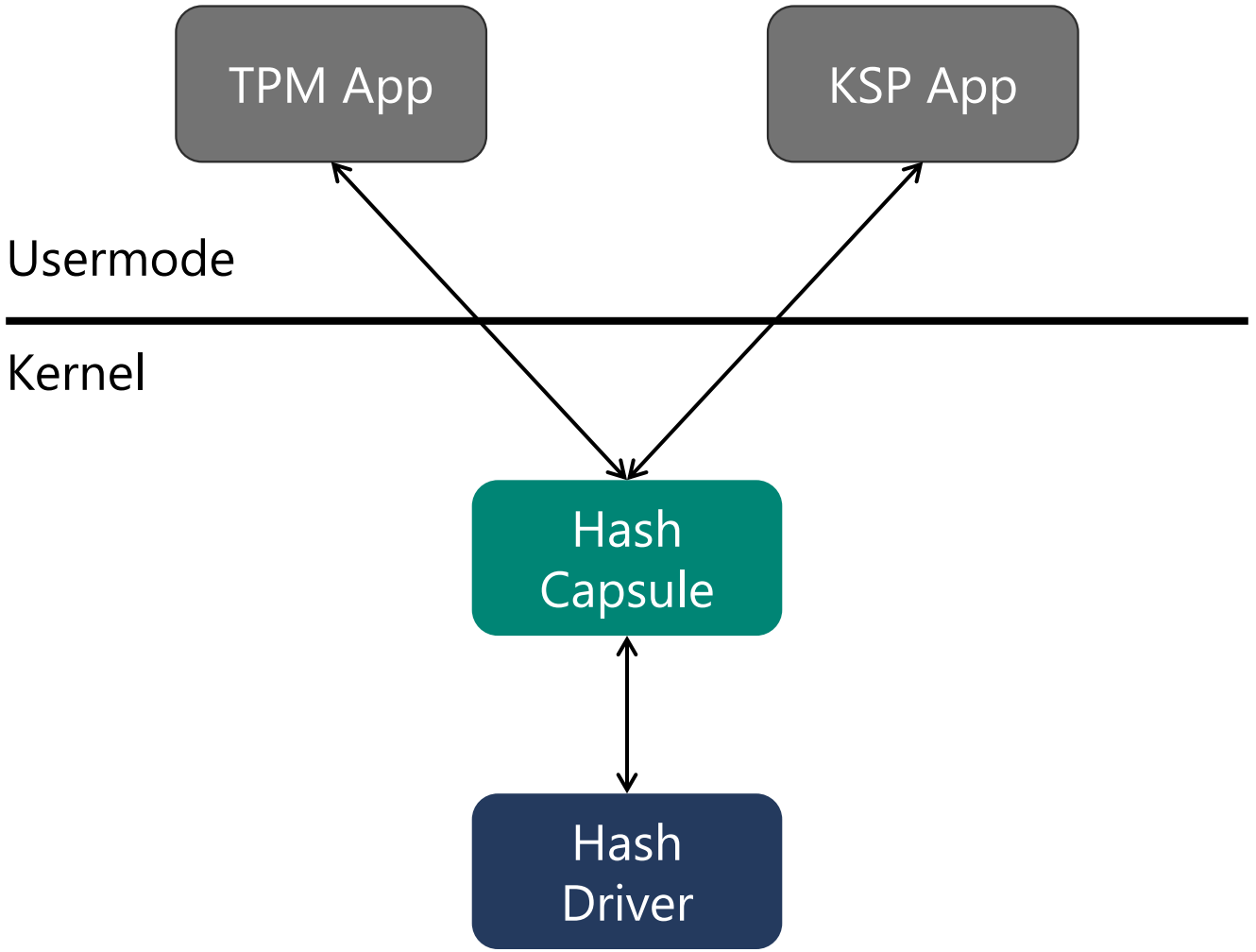
- Lightweight and simple
- Owns resource
- Allows mutation (DerefMut)

RefMutex<T>

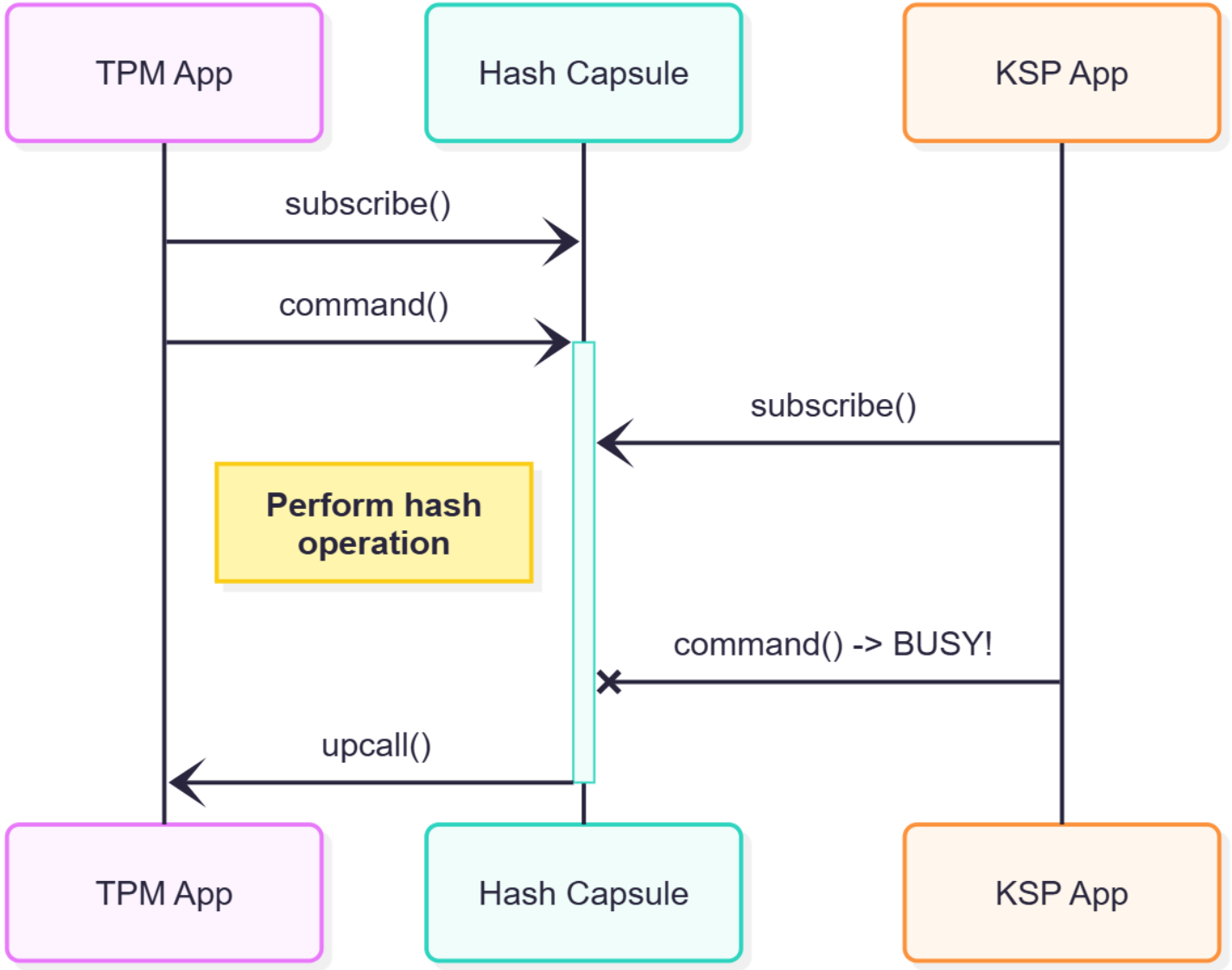
- Multiple generics (downcasting)
- Reference counting
- No mutation

Driver Command Synchronization

Case Study 3: Hash Driver Contention



Command Synchronization

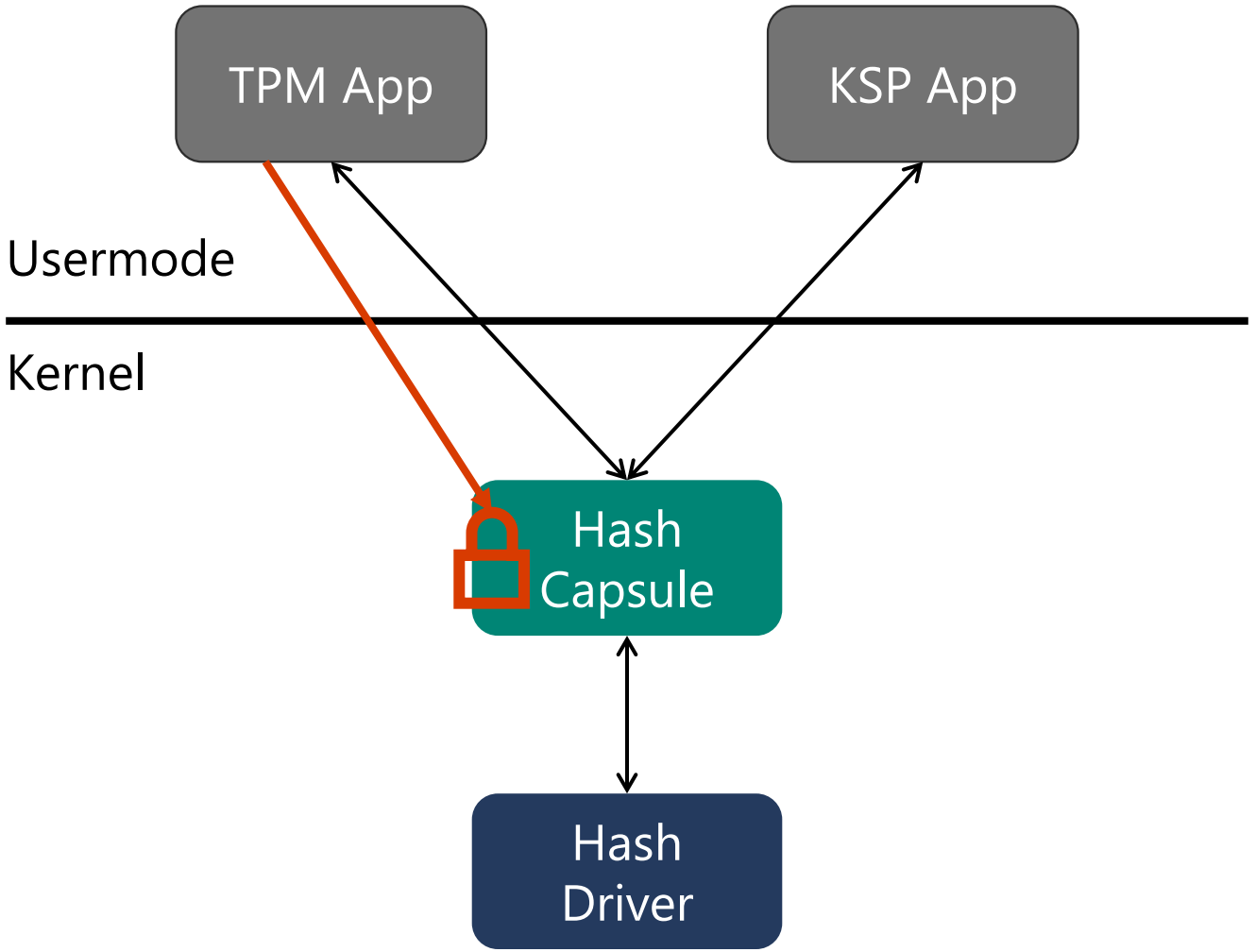


Solution Proposals

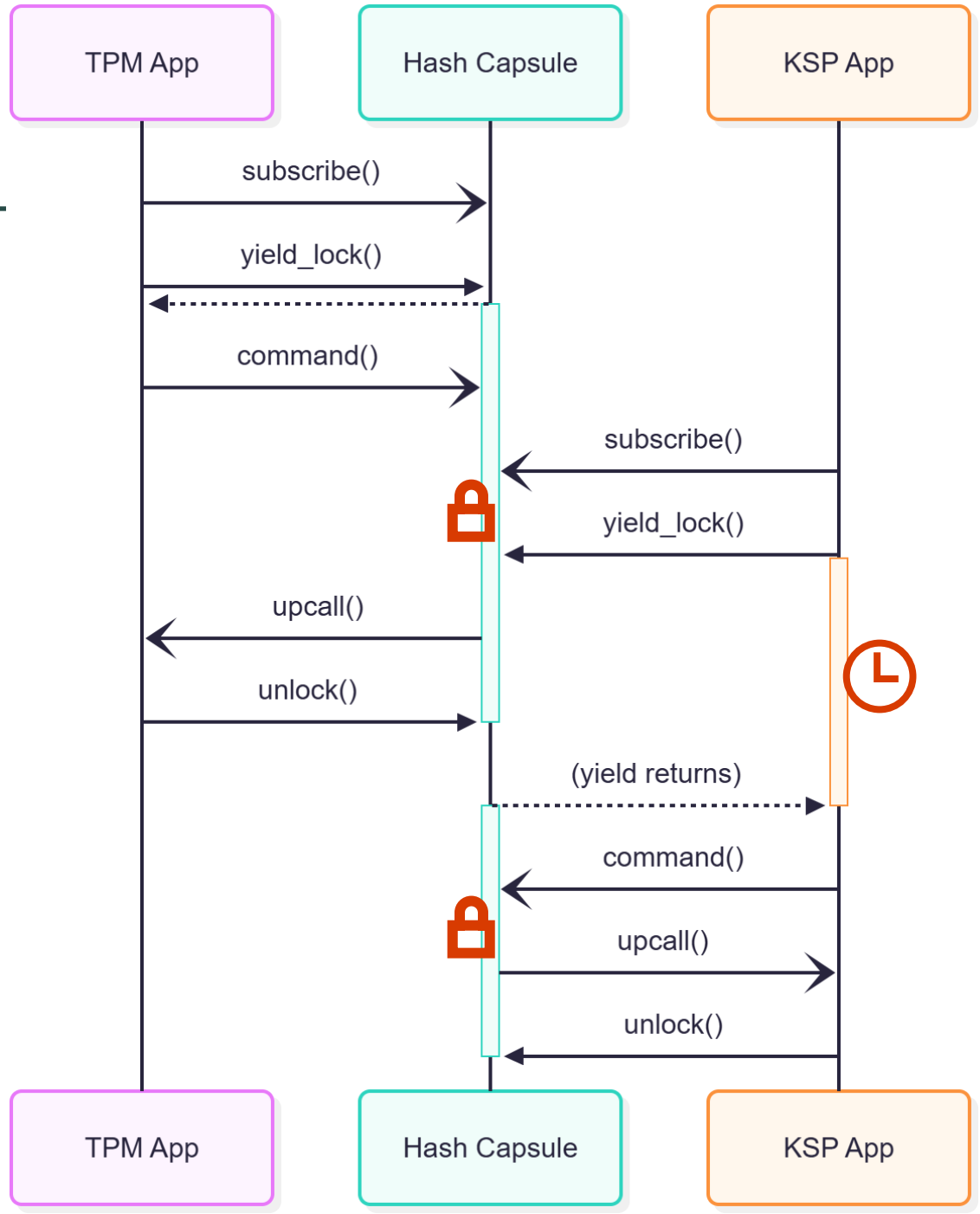
- Proposal A – Apps explicitly synchronize with each other
- Proposal B – Capsules explicitly queue requests
- Proposal C – Kernel-managed request queue

- Open Discussion!

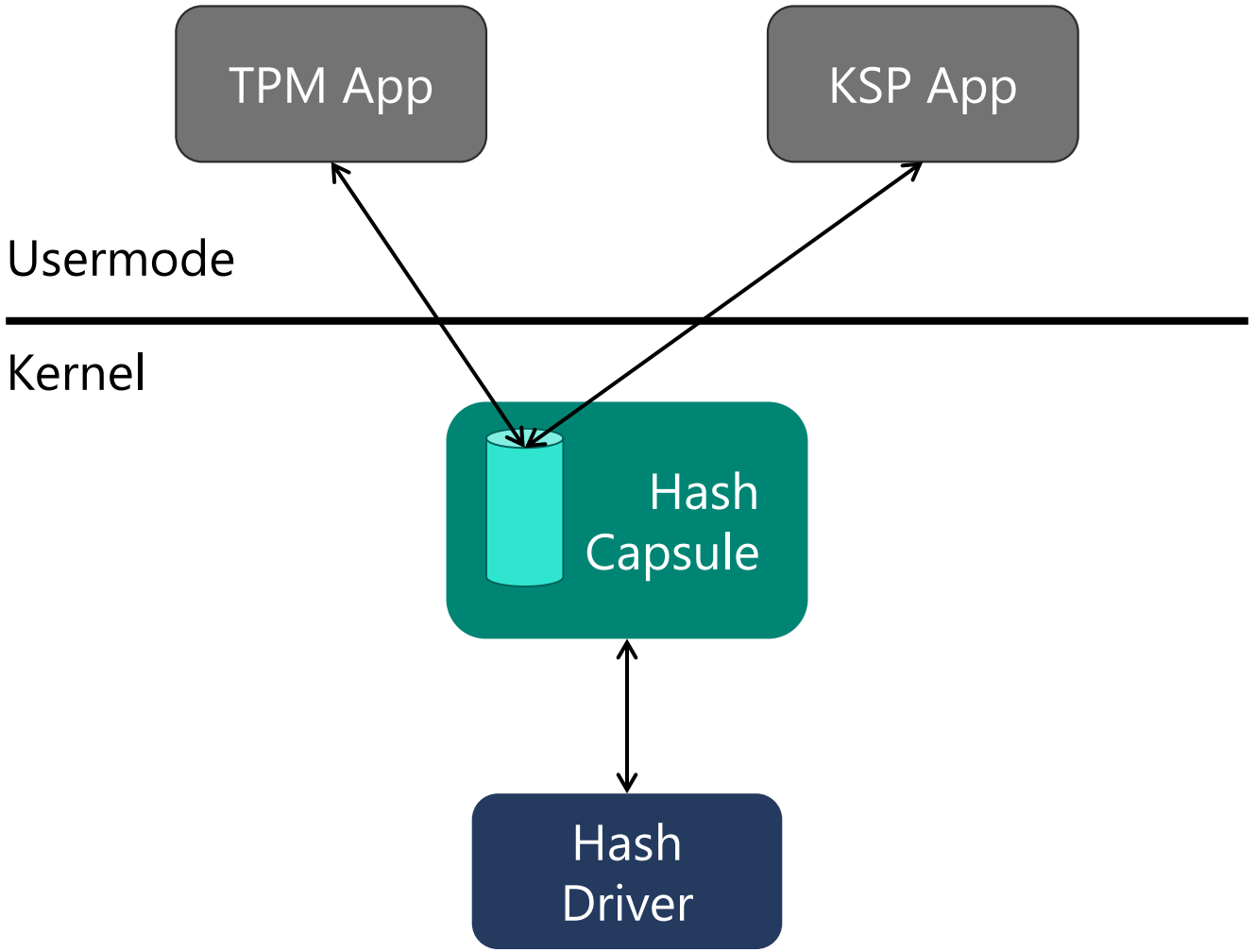
Proposal A: Lock Syscall



Proposal A: Lock Syscall



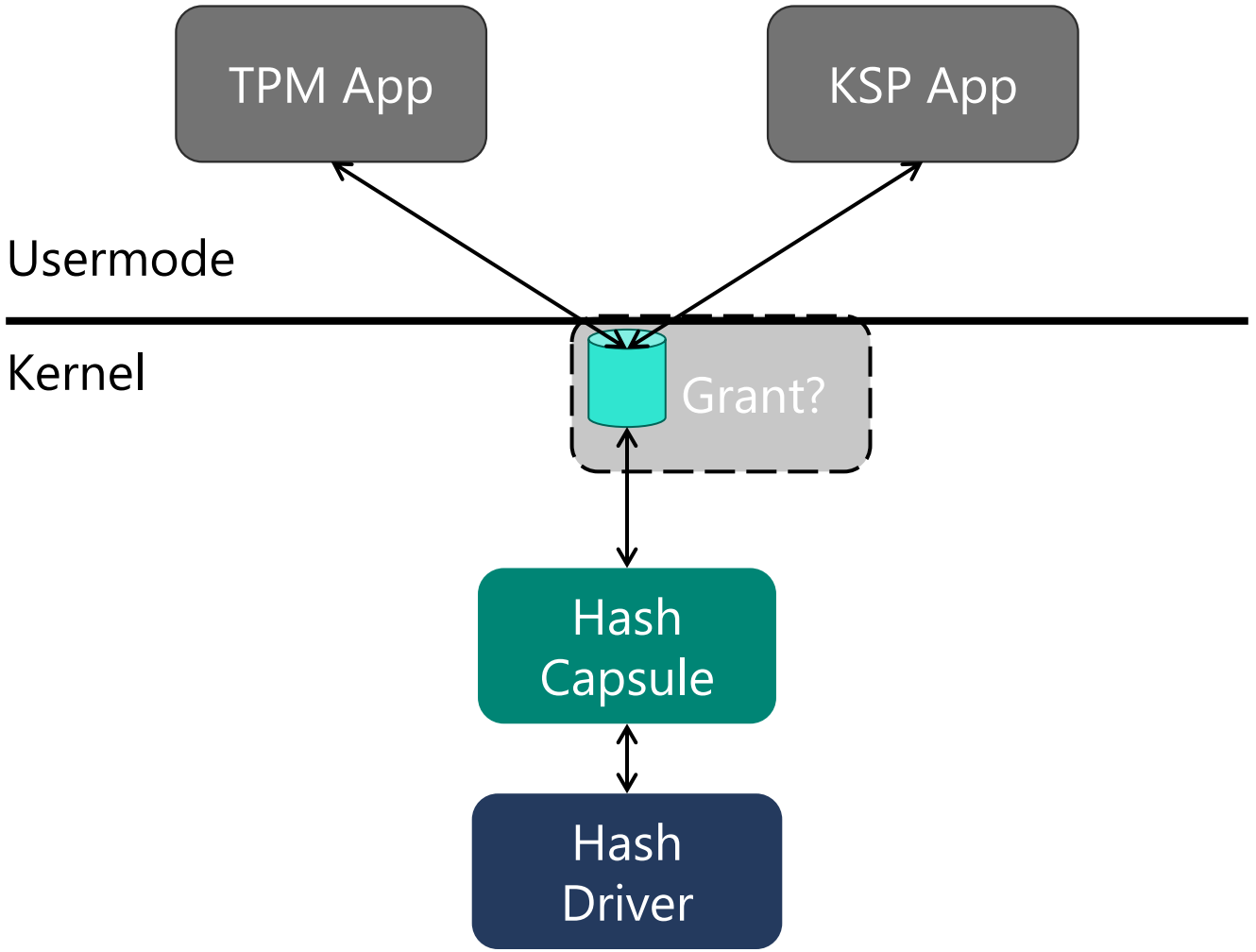
Proposal B: Capsule Queuing



Proposal B: Capsule Queuing

```
struct Command {
    pid: ProcessId,
    num: usize,
    arg1: usize,
    arg2: usize,
}

struct MyCapsule {
    cmd_queue: RefCell<RingBuffer<Command>>,
    dc: DeferredCall,
}

impl SyscallDriver for MyCapsule {
    fn command(&self, /* args */) -> CommandResult {
        let cmd_queue = self.cmd_queue.borrow_mut();

        if cmd_queue.any(|c| c.pid == pid) {
            return CommandResult::Failure(ErrorCode::BUSY);
        }

        cmd_queue.enqueue(Command {
            pid,
            num,
            arg1,
            arg2,
        });

        if !self.busy() {
            self.dc.set();
        }

        CommandResult::Success
    }
}
```

Proposal C: Kernel Queuing



Proposal C: Kernel Queuing

Second-order effects:

- Blocking syscalls
- Grant polymorphism