Concurrent Reactive Objects in Rust Secure by Construction

Marcus Lindner, Jorge Aparicio, Per Lindgren
Luleå University of Technology, Sweden; email: {marcus.lindner@,jorapa-7@student.,per.lindgren@}ltu.se

Abstract

Embedded systems of the IoT era face the software developer with requirements on a mix of resource efficiency, real-time, safety, and security properties. As of today, C/C++ programming dominates the mainstream of embedded development, which leaves ensuring system wide properties mainly at the hands of the programmer. We adopt a programming model and accompanying framework implementation that leverages on the memory model, type system, and zero-cost abstractions of the Rust language. Based on the outset of reactivity, a software developer models a system in terms of Concurrent Reactive Objects (CROs) hierarchically grouped into Concurrent Reactive Components (CRCs) with communication captured in terms of time constrained synchronous and asynchronous messages. The developer declaratively defines the system, from which a static system instance can be derived and analyzed. A system designed in the proposed CRC framework has the outstanding properties of efficient, memory safe, race-, and deadlock-free preemptive (single-core) execution with predictable real-time properties. In this paper, we further explore the Rust memory model and the CRC framework towards systems being secure by construction. In particular, we show that permissions granted can be freely delegated without any risk of leakage outside the intended set of components. Moreover, the model guarantees permissions to be authentic, i.e., neither manipulated nor faked. Finally, the model guarantees permissions to be temporal, i.e., never to outlive the granted authority. We believe and argue that these properties offer the fundamental primitives for building secure by construction applications and demonstrate its feasibility on a small case study, a wireless autonomous system based on an ARM Cortex M3 target.

1 Introduction and motivation

Besides constraints set by the environment and the target platform like available memory, CPU, and energy resources in addition to other functional and extra-functional properties of the application at hand, embedded software typically operates autonomously with requirements on safety, robustness, reliability, and security. Developers commonly design embedded systems of the IoT era by taking the outset of a reactive model implemented in C/C++ either as a bare metal interrupt driven application or through the support of some threading library. Meeting the aforementioned requirements is at a large up to the programmer with little or no support for verification. Central to correctness is the management of memory resources with problems spanning from array indexing and dangling pointers all the way to race conditions and deadlocks in the concurrent setting.

In this paper, we take the outset from prior work on Concurrent Reactive Objects (CROs) [1] with a heritage to the Timber language [2] and the Real-Time For the Masses (RTFM, [3]) set of experimental languages and tools. Whereas Timber provides a high level modeling and implementation approach offering state protection in the concurrent setting, the dynamic memory model requires automatic management which precludes the deployment to lightweight targets.

With a clear motivation, we want to provide a programming model that ensures memory safety in a concurrent setting along with a concurrency model amenable to static analysis. However, developing yet a new fully fledged language with accompanying ecosystem is questionable when taking the amount of work into consideration[1]. Instead, we seek to leverage on ongoing community efforts around programming languages and ecosystems.

Among recent developments, the Rust language stands out with a memory model, which provides compile time memory safety and monomorphization, and has a tight coupling to LLVM achieving zero-cost abstractions through link time optimization. Sidestepping the compiler is explicit (`unsafe`) and can be rejected in user code, thus allowing for fearless programming to the end of memory safety and other properties within reach of the Rust compiler. In the context of embedded development, Rust applications on bare metal targets have already been shown possible [4, 5].

In this paper, we further explore the Rust memory model and the CRC framework towards systems being secure by construction. In particular, we show the following properties.

- Granted permissions can be freely delegated without any risk of leakage outside the intended set of components. Key here is the static CRC topology, where communication paths are known at compile time, together with the Rust language borrowing semantics.

[1]An observation here is that the design of any memory safe language would need to take memory aliasing into account, a property directly given by the Rust language.

- Permissions are guaranteed to be authentic, i.e., they can neither be manipulated nor faked. Key here is the underlying module system and type scoping together with the memory safety provided by the Rust language. In effect, preventing any intentional or accidental memory corruption leads up to an unauthentic permission.
- Permissions are guaranteed to be temporal, i.e., they can never outlive the granting authority. Key here is the concept of lifetimes, which the Rust language brings and the compiler enforces.

In conclusion, we believe and argue that these properties offer the fundamental primitives for building secure by construction applications and demonstrate their feasibility on a small case study, a wireless autonomous system based on an ARM Cortex M3 target.

2 Background

The Rust memory model and the Stack Resource Policy (SRP) based scheduling approach is at heart of the proposed framework.

The ambition behind Rust is to provide a systems programming language with memory safe zero-cost abstractions. In Rust, mutability is first class, distinguishing between *immutable* (`&T`) and *mutable* (`&mut T`) references with the following invariant:

> At any instance in time each value of `T` may be *mutably* referenced once or *immutably* referenced zero or arbitrarily many times.

In Figure 1, A and B denote two concurrent execution contexts while a, b, and c are references to a shared location or resource T. The invariant applies to (1) the concurrent case with accesses from a of context A and b/c of context B and (2) the sequential case with accesses from b and c of context B. For the concurrent case (1), the invariant ensures obviously race free access while for the sequential case (2), the invariant may at first glance appear too restrictive. However, the sequential restriction allows to spot and reject at compile time memory related issues such as iterator invalidation (see Section 4.9 in [6]). Moreover, the invariants are passed to the compiler back-end (LLVM) as *no alias* attributes allowing aggressive yet safe code optimization.

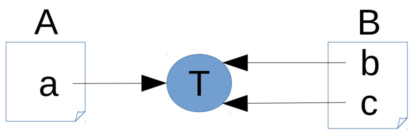

Figure 1: Illustration of the Rust memory model.

Rust implements an *affine* type system and an *ownership* model with the notion of *lifetimes*. The *borrow checker* and lifetime analysis ensures memory consistency of *safe* code.

Any access to shared mutable data ultimately boils down to explicitly stated *unsafe* code, out of reach for the Rust compiler to verify. Abstractions providing a *safe* API allow user code to access shared mutable data. Thus, given that the abstractions are sound (i.e., uphold the invariant), any program passing compilation is memory safe by construction![2]

2.0.1 *Stack Resource Policy based scheduling*

Stack Resource Policy (SRP) based scheduling offers a means to preemptive scheduling of tasks with shared resources on single-core processors [7]. The approach offers advantages in terms of deadlock-free execution, efficient memory utilization, single blocking, and so on and brings a plethora of readily available methods for static analysis, see, e.g., [7, 8].

SRP builds upon static analysis of the task set T to derive the ceiling value $\pi(r)$ for each resource r. $\pi(r) = max(p(t))$ with $t \in T_l(r)$, where $p(t)$ is the priority of task t and $T_l(r)$ is the set of tasks that (may) access the resource r. During execution, the (dynamic) system ceiling $\Pi = max(\pi(r))$ with $r \in L$, where L is the (global) set of currently held resources. A task $t \in T$ may preempt the currently running task t_e only if $p(t) > p(t_e)$, $p(t) > \Pi$, and $p(t) = max(p(t'))$ with $t' \in P$, where P is the set of pending tasks.

Targeting lightweight MCUs, we can exploit the underlying interrupt hardware to implement the system ceiling and perform static priority scheduling[3]in compliance to SRP, achieving performance on par with hand written bare metal code [3].

3 Model of computation

Component models are frequently used to capture the system topology and to bring the benefit of re-use. Our system model is declarative, defined in terms of nested Concurrent Reactive Components (CRCs) with Concurrent Reactive Objects (CROs) at the leafs. The system designer declares interaction inside the model and with its environment in terms of time constrained synchronous and asynchronous point-to-point messages, where ultimately the end points are methods of CRO leaf instances.

3.1 Execution semantics

The execution model builds on the notion of *time constrained messages* defined as

$$M : \{BL : Ti, dl : Ti, o : \&O, f : (\&O, D) \rightarrow R, d : D\},$$

where Ti is a time type, BL specifies the absolute release time, dl specifies the relative deadline, o indicates the target object, f indicates the method to execute, and d is the payload (i.e., the arguments for the receiver).

The execution of a message

$$E(m : M) \rightarrow R$$

returns with a value of type R. Messages execute concurrently under mutual exclusion on the object state (o) (similar to Ada's protected objects or Java's synchronized methods) and run-to-completion within their eligible timing window for any correctly scheduled system.

[2]Memory safety can in most cases be statically ensured. If not, a run-time monitor is injected to emit a `panic!` on a memory violation. Stack memory allocation errors (overruns) are assumed to be treated at the run-time system level.

[3]Eligible tasks with the same priority are scheduled in static order. While preserving invariants for correctness, we must take this into consideration during the response time analysis.

3.2 Timing semantics

The absolute release time BL along with the absolute deadline $DL = BL + dl$ define the eligible timing window for the execution of a message m. The execution $E(m : M)$ of a message m may emit additional synchronous messages

$$Sync(o' : \&O', f' : (\&O', D'), d' : D') \rightarrow m' : M,$$

which result in messages

$$m' = M\{BL = m.BL, dl = m.dl, o = o', f = f', d = d'\}$$

that *inherit* the sender's timing window. The synchronous execution $E(m' : M) \rightarrow r : R$ blocks the sender and returns the value r.

Similarly, the sender may emit asynchronous messages with a relative release time bl''

$$Async(bl'' : Ti, dl'' : Ti, o'' : \&O'', f'' : (\&O'', D''), \\ d'' : D'') \rightarrow m'' : M,$$

which result in messages

$$m'' = M\{BL = m.BL + bl'', dl = dl'', \\ o = o'', f = f'', d = d''\}$$

with a timing window *relative* to the sender's ($E(m : M)$) timing window. Emitting an asynchronous message m'' amounts to queuing the message for later execution. The emission of an asynchronous message returns a reference to that message, which allows the cancellation of the message as long as its execution is not yet scheduled[4].

3.3 Discussion

The CRO model resembles *actor* models in that the execution of asynchronous messages is decoupled from the sender. However, the notion of synchronous communication is usually not found in actor models, while here supported with resemblance to *monitors* and *protected* objects. Messages execute under mutual exclusion on the corresponding object (resource). This not only allows race-free execution by construction but also ensures sequential behavior of operations holding a resource. This is instrumental to control the order of side effects not only on object states but also for communication, i.e., synchronous calling of other objects and communication with the environment. Asynchronous messages are the units of concurrency with the execution semantics precisely defined by their resource dependencies, where mutual exclusion is the sole (necessary and sufficient) means to synchronization.

In this paper, we target lightweight MCUs and adopt an SRP based scheduling approach, where the asynchronous messages constitute SRP *tasks* and objects amount to (shared) SRP *resources*.

At the border of the system, we find the environment, which drives our reactive model, represented as event or message sources. Internal events and actions become *observable* only at the point where communication involves the environment. In the setting of embedded targets, the environment is typically represented by the hardware peripherals, where the interrupt handlers are our event or message sources. This can be generalized to APIs of external code and hosted environments [9], where the underlying operating system schedules our tasks on top of its thread model and the external code emits messages or events.

To facilitate re-use and to manage complexity, the model provides a hierarchical component based abstraction. The declarative definition allows us to statically analyze the topology of the system and derive a flat system instance without the need of dynamic bindings. As we show in the remainder of the paper, the CRC/CRO model can be implemented efficiently using zero-cost abstractions of the Rust language and rendering executables that perform on par with carefully designed bare-metal code.

4 LED runner example

Figure 2 depicts a CRC system, which autonomously controls the RGB values of an LED array. At the highest level, the system consists of two components, the `USART` CRO and the `LED` CRC wrapping the `STM` state machine and the `DMA` CROs. The `USART` receives and parses the serial stream and controls the `LED` component. The `DMA` CRO sends a frame of data to the LED array utilizing the DMA hardware. The `STM` CRO triggers on behalf of the periodically executing `transition` method the `on_update` method, which generates the frame content. The `on_command` method controls the behavior of the state machine, i.e., the direction and speed of the running lights. The `transition` method emits an asynchronous message with a baseline offset to postpone the release of the message, which implements the periodic behavior.

5 CRC framework

With the CRC framework, a developer specifies the system topology in terms of CROs and CRCs through `.cro` and `.crc` files, respectively. The developer provides the behavior of the CROs in form of standard Rust code, i.e., `.rs` files.

A *build script* (`build.rs`) is the Rust mechanism for code generation *before* compilation. Our framework uses a build script to analyze the system model, transform `.cro` and `.crc` files into actual Rust code, and inject it into the compilation process.

A `.cro` file for each CRO stores its specification. Listing 1 shows the definition of the `USART` CRO. The file specifies port signatures (`signature`) with Rust syntax. Each input port enumerates internal connections to output ports (`sync_ports`/`async_ports`) and peripheral dependencies (`peripherals`).

A `.crc` file for each CRC stores its specification. Listing 2 shows the main wrapping CRC of our example system. A CRC consists of CRO and other CRC instances, referred to

[4] We have not yet implemented this feature in the prototype Rust framework.

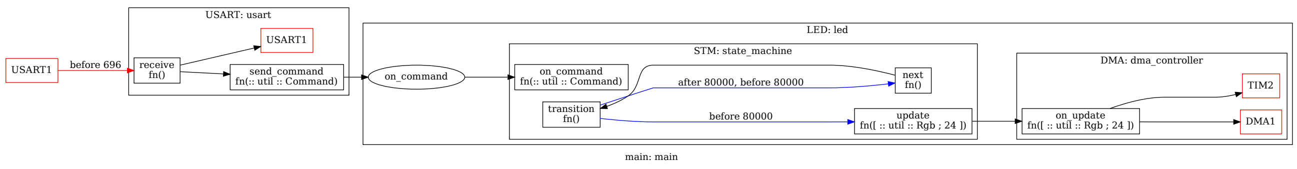
USART1
before 696
USART: usart
receive fn()
USART1
send_command fn(:: util :: Command)
on_command
LED: led
STM: state_machine
on_command fn(:: util :: Command)
transition fn()
after 80000, before 80000
next fn()
before 80000
update fn([:: util :: Rgb ; 24])
DMA: dma_controller
on_update fn([:: util :: Rgb ; 24])
TIM2
DMA1
main: main

Figure 2: The LED runner CRC system autonomously controls the RGB values of an LED array. Synchronous messages are marked black and asynchronous messages are marked red for environmental messages and blue for internal messages.

```
inputs: {
    receive: {
        signature: fn(),
        sync_ports: [send_command],
        peripherals: [USART1],
    },
},

outputs: {
    send_command: fn(::util::Command),
}
```

Listing 1: File `usart.cro` (template).

as `components`. The `.crc` file specifies incoming and outgoing connections to each component on a per-component basis. Finally, the CRC itself has input and output ports. The analysis stage of the framework derives these ports from the `connections` information in the file. If a port specifies no component, it is a CRC port. The `interrupts` field defines the interrupt sources of the CRC and according component connections. The `available` field contains a list of additional interrupt sources that may be used to dispatch async messages while `device` indicates the crate (i.e., library) holding the peripheral API.

Our framework derives a system instance by spanning the top level CRC[5].

Using our CRC framework, a user implements the CRO application logic in safe Rust code. Listing 3 partly shows the state machine implementation of our example system.

A `State` struct represents the state of the CRO and defines input ports as methods on the struct. The `new` constructor initializes the state of the CRO, which the compiler evaluates at compile time (due to the `const` context).

The exact signature of each input port varies according to its `.cro` file specification. Input port methods can have as arguments a mixture of:

1. The port input (i.e., message payloads),
2. A set of output ports both synchronous and asynchronous, and
3. A set of peripherals.

The `Ports` struct provides the output ports, which are essentially normal Rust functions (see lines 19 and 23 in Listing 3).

```
components: {
    USART: {
        template: usart,
        connections: [
            self.send_command
                        -> LED.on_command,
        ],
    },

    LED: {
        template: led,
    },

},

interrupts: {
    USART1: {
        connects_to: USART.receive,
        before: 696, // 87 us
    },

    available: [EXTI0],
},

device: stm32f103xx,
```

Listing 2: File `main.crc`.

```
1  use util::{Command, Direction, Rgb};
2  cro!(); // include auto generated code
3
4  pub struct State {
5      active: bool,
6      rgb: [Rgb; 24],
7      ...
8  }
9
10 impl State {
11     pub const fn new() -> Self {
12         State { active: false, ... }
13     }
14
15     pub fn transition(
16         &mut self,
17         port: self::transition::Ports,
18     ) {
19         (port.async.next)();
20
21         if self.active {
22             ...
23             (port.async.update)(self.rgb);
24         ...
```

Listing 3: File `state_machine.rs` (abridged).

[5] Currently, our framework allows `available` and `device` fields only in the top level CRC, but we will remove this restriction in the future to allow cross crate component re-use.

```
// root of the crate
extern crate stm32f103xx; // target device
extern crate blue_pill; // development board

crc!(); // indicates that it is a CRC system
```

Listing 4: File `main.rs`.

```
1  use stm32f103xx;
2  use blue_pill::{Channel, Pwm};
3  cro!();
4
5  pub struct State {
6      buffer: ...,
7  }
8
9  impl State {
10     ...
11
12     pub fn on_update(
13         &mut self,
14         rgb: [Rgb; 24],
15         p: self::on_update::Peripherals,
16     ) {
17         p.DMA1.claim(|dma1| {
18             p.TIM2.claim(|tim2| {
19                 let pwm = Pwm(tim2);
20                 pwm.set_duties(
21                     dma1,
22                     Channel::_1,
23                     &self.buffer
24                 ).unwrap();
25             });
26             ...
27         });
28     }
29 }
```

Listing 5: File `dma_controller.rs`.

The root of the crate (see file `main.rs` in Listing 4) lists all library dependencies and indicates with a `crc!()` macro call that this is a CRC system.

5.1 Implementation

5.1.1 Peripherals

We opted to implement access to peripherals as structs with interior mutability. Thus, mutation of the registers is possible through shared (`&T`) references. The `svd2rust` [10] tool automatically generates a register-level API for peripherals from vendor based SVD files. CROs can access peripherals as if they were resources using a `claim` method and passing a closure. Nested closures allow the access to multiple peripherals (see lines 17-25 in Listing 5). Internally, the peripheral APIs use *volatile* read/write operations.

5.1.2 Analysis

The build script collects all `.cro` and `.crc` files, parses them, and combines them into a system model sys or rejects ill-formed models (see Section 7.1).

During the analysis stage, we derive for the SRP scheduling the task priorities (i.e., interrupt priorities) from the given timing constraints and the resource ceilings from the task priorities [7,9].

5.2 Code generation

If sys is well-formed, the build script proceeds to generate Rust code required for run-time execution.

5.2.1 CROs

For each CRO, our framework generates a Rust file with the definition of the `Ports` and `Peripherals` structs. The `cro!` macro injects this file into the compilation process, which allows the Rus compiler (`rustc`) to reject user code that mismatches port signatures specified in the corresponding `.cro` file.

5.2.2 Top level CRC

Our framework generates a Rust file for the whole system that contains the full application logic. The `crc!` macro injects this file into the compilation process.

This file contains a module for each CRO instance in the system, which statically allocates a `State` struct per CRO instance. The code in the system file also optimizes connections by refining synchronous messages to function calls and asynchronous messages to enqueue operations.

Synchronous messages: For every CRO instance, our framework generates a proxy function for each input port. The signature of this proxy matches the signature entered by the user in the `.cro` file. Instantiating the `Ports` struct of a sending CRO with the proxy that matches an actual connection to a receiving CRO expresses the connection between the two CRO instances.

Asynchronous messages: An asynchronous message defers the invocation of the input port (i.e., the object method) by storing the input data and the input port function pointer in a queue. An interrupt handler executes asynchronous messages at a later time. The message struct holds a `next` field forming an *in place* linked list. In order to store differently typed objects because asynchronous messages differ in the payload field type, we enforce a static layout of all message types (`#[repr(C)]`) ensuring the `next` field to have a known offset.

6 Security

Embedded systems often access and process sensitive data. For this reason, security is an important factor that may not be disregarded when designing and developing embedded software. Ravi et al. [11] argue it is wrong to establish security by solely adding features like encryption to a system. Instead, we have to take all aspects of embedded system design into account together with existing resource constraints, e.g., performance and power limitations.

In the context of lightweight embedded systems, resource constraints come into play, thus memory and CPU efficiency

```rust
mod trusted {
    pub struct Auth {
        level: u8,
    }

    pub fn auth(k: &str) -> Option<&Auth> {
        if k == "abc" {
            Some(&Auth { level: 1 })
        } else {
            None
        }
    }
    ...
```

Listing 6: File `trusted_base.rs` (abridged).

become an issue. The emerging security enabled microcontrollers exemplify this. They range from hardware AES encryption like the ARM Cortex-M3 based stm32l162vc to more elaborate solutions like the Cortex-M4 based CEC1702 offering hardware encryption, authentication, and public key capabilities. Hardware cryptographic ciphering may offer speedup and increase energy savings by orders of magnitude over software solutions. Moreover, pre-boot authentication of system firmware offering a root of trust, firmware update authentication, authentication of system critical commands, and protection of secrets with encryption improves system integrity.

In this paper, we focus on software in security mechanisms from the outset of the *platform agnostic* CRC framework and leverage properties of the Rust language to establish security mechanisms, which are guaranteed by the Rust semantics and statically ensured by the Rust compiler. While being complementary to security mechanisms offered by the underlying hardware, we argue that a higher degree of trust and reliability can be achieved by also offering compile time guarantees to the embedded software.

6.1 Authentication and authorization

At device level, we are concerned with the permissions to access data and perform operations. The trusted base has the authority to grant such device level permissions based on a-priory knowledge or external authentication. For this presentation, we focus on device level authorization mechanisms and discuss system level authorization as future work.

The notion of *opaque* structures in the Rust language allows us to define data types that a user can neither construct, nor inspect, nor manipulate, merely pass on as parameters. This perfectly fits the need and purpose of device level authorization, where the trusted base grants permission to the user.

Listing 6 demonstrates an implementation of an authorization ticket providing a range `u8` of permission levels. Note, the `Auth` structure is public but its `level` data field is private to the module, i.e., code from other modules can hold a reference to an `Auth` structure but not create it or access the containing data. Therefore, the public `auth` function returns an authorization ticket in case the input matches the defined a-priory knowledge (here `"abc"`).

```rust
1 // user code in `safe` Rust
2 fn user1(d: &Sec<u32>, e: &Enc<u32>) {
3     let a = auth("abc").unwrap();
4     user2(&sec_add_u32(d, &e.get(a)));
5     user3(d, &e.get(a));
6     user4(d, e, a);
7 }
```

Listing 7: File `user1.rs`.

By default, `structs` in Rust do not implement the `Copy` trait, thus user code cannot duplicate the ticket but a reference thereof. Moreover, tickets are temporal with a lifetime limited to the sequential execution context of the call to the granting authority. This follows from the Rust lifetime semantics.

Listing 7 depicts user code written in “safe” Rust. The user requests authorization in line 3 and uses the given permission `a` locally in lines 4 and 5. In line 6, the user passes the permission ticket on to another user function. For brevity, the example illustrates the concept with plain Rust code but permission delegation is also possible through synchronous messages. However, asynchronous messages cannot store the received ticket due to the lifetime bound and consequently the temporal property of the authorization holds.

6.2 Secure data container

Dealing with sensitive data sets a number of restrictions and requirements regarding integrity, use, and visibility. Specifically, *integrity* restricts primitive operations on sensitive data to be limited to the trusted base, an authority limits the *use* of sensitive data, and the designers intention limits its *visibility*. Also to this end, we can ensure the desired behavior through an opaque representation of secure data.

Listing 8 demonstrates an implementation of a generic (i.e., polymorphic to the type `T`) secure data container `Sec<T>`. Only the trusted base code can instantiate and delegate this type. The function `sec_add_u32` (lines 18 to 20) exemplifies how to declare primitive operations on arbitrary instances of secure data containers in the trusted base. While the function internally uses `unsafe` code, the API is *safe*, i.e., *safe* Rust code can call the function[6]. Notice here, user code has never access to the inner *secure* data or can disclose it because the return type is also a secure data container `Sec<u32>`.

6.3 Encryption and decryption

While cryptography as such is not the focus of this work, we discuss the topic from the framework perspective and highlight outsets for efficient, reliable, and secure management of sensitive information. To this end, we leverage on the Rust language zero-cost abstractions and type system with static guarantees offered by the compiler.

[6] An alternative to `unsafe` is to use the `pub(crate)` modifier and use visibility as a fence for usage violations.

```rust
...
// opaque representation of secure data
#[derive(Debug)]
pub struct Sec<T> {
    data: T,
}

impl<T> Sec<T> {
    pub unsafe fn new(d: T) -> Self {
        Sec { data: d }
    }
    pub unsafe fn get(&self) -> &T {
        &self.data
    }
}

// safe API for operating on Sec<u32>
pub fn sec_add_u32(s1: &Sec<u32>,
                   s2: &Sec<u32>)
    -> Sec<u32> {
    unsafe { Sec::new(s1.get() + s2.get()) }
}
...
```

Listing 8: File `trusted_base.rs` (continued).

```rust
    ...
    // in place transformation
    // by a cipher closure f
    fn cipher<T, F>(s: &mut T, mut f: F)
    where
        T: Sized,
        F: FnMut(&mut u8),
    {
        let ptr = s as *mut T as *mut u8;
        for i in 0..size_of::<T>() {
            f( unsafe {
                &mut *ptr.offset(i as isize)
            });
        }
    }
    ...
```

Listing 9: File `trusted_base.rs` (continued).

6.3.1 *Cipher*

A fully fledged cryptography crate (rust-crypto = "0.2.36") is readily available providing implementations for popular ciphers (AES, RC4, and others). While strong encryption by software is likely resource consuming and may thus be out of range for light-weight targets, a microcontroller may defer the actual encryption and decryption to a capable encryption hardware if supported by the target.

To the purpose of this presentation, Listing 9 demonstrates an in place transformation of raw data. The generic function `cipher<T, F>` iterates the closure `f:F` over the byte array representation of the data `s` and transforms it. With a suitable cipher closure `f`, the function encrypts or decrypts the data.

6.3.2 *Encrypted data container*

As we have already seen in Section 6.1, authorization tickets are secure against faking and manipulation in user code. We can use this approach to delegate secure information keyed with an authorization ticket. Also here, we take the outset of an opaque type definition[7].

```rust
1       ...
2       // opaque representation of
3       // encrypted data
4       pub struct Enc<T> {
5           data: T,
6       }
7
8       impl<T> Enc<T>
9       where
10          T: Copy,
11      {
12          pub unsafe fn new(d: &T) -> Self {
13              let mut c = d.clone();
14              cipher(&mut c, |i| { *i += 1; });
15              Enc { data: c }
16          }
17
18          pub unsafe fn get_unsafe(&self)
19            -> Sec<T> {
20              let mut c = self.data.clone();
21              cipher(&mut c, |i| { *i -= 1; });
22              Sec::new(c)
23          }
24
25          pub fn get(&self, _: &Auth)
26            -> Sec<T> {
27              unsafe { self.get_unsafe() }
28          }
29      }
30  }
```

Listing 10: File `trusted_base.rs` (end).

Listing 10 demonstrates an implementation of a generic encrypted data container `Enc<T>`. The signature of the `new` constructor specifies the `unsafe` modifier, and as a consequence, solely the trusted base code can call the function and create a new encrypted data container. When calling `new`, the constructor applies the `cipher` function to a copy of the data (lines 13 and 14) and returns the encrypted data in an `Enc<T>` container (line 15). The closure `|i| { *i += 1; }` (line 14) increments each byte of the data by 1, which essentially is the classical *Caesar cipher* [12]. User code has access to a *safe* API function (`get(...)` in lines 25 to 28), which internally uses an `unsafe` function to return a `Sec<T>` secure container that holds the decrypted information.

6.4 Example

Listing 11 demonstrates the application of our proposed security system. The trusted base instantiates a secure container `Sec<u32>` (line 3) and an encrypted container `End<u32>` (line 4) Following this, it calls the user code function `user1` and passes on references to the containers (line 5). The user code is free to delegate the references but has never access to the actual content of the containers. Note also, the authorization ticket `a` is temporal with a lifetime limited to the sequential execution context of `user1`, even when delegated to `user4`.

[7]While we can indeed allow the user to read encrypted data, we do not want the user to create or manipulate encrypted data outside the control of the trusted base.

```rust
// inside trusted base
fn main() {
    let d = unsafe { Sec::new(10u32) };
    let e = unsafe { Enc::new(&32u32) };
    user1(&d, &e);
}

// user code in `safe` Rust
fn user1(d: &Sec<u32>, e: &Enc<u32>) {
    let a = auth("abc").unwrap();
    user2(&sec_add_u32(d, &e.get(a)));
    user3(d, &e.get(a));
    user4(d, e, a);
    user5(d, e);
}

fn user2(d: &Sec<u32>) {...}

fn user3(d1: &Sec<u32>, d2: &Sec<u32>) {
    let d = sec_add_u32(d1, d2));
    ...
}

fn user4(d: &Sec<u32>, e: &Enc<u32>,
  a: &Auth) {
    let d = sec_add_u32(d, &e.get(a)));
    ...
}

fn user5(d: &Sec<u32>, e: &Enc<u32>) {...}
```

Listing 11: File `example.rs`.

```rust
    ...
    impl<T> !Send for Sec<T> {}
    ...
```

Listing 12: File `trusted_base.rs`.

6.5 Discussion

Our intention here is not to provide a fully fledged security framework but rather to demonstrate that security by construction is indeed feasible with our approach. The reader may notice that unwrapping encrypted information stores the decrypted data in plain form. This is perfectly secure from the perspective of the embedded software as the plain data is still wrapped in a secure container `Sec<T>` and thus not directly exposed to the user code. However, side channel attacks may exploit plain (decrypted) data that is stored in persistent memory.

With the proposed design, we allow persistent storage of decrypted data beyond the lifetime of the authorization ticket. I.e., a CRC component may store a `Sec<T>` container in its state when using a delegated authentication ticket to unwrap data from an encrypted container. If we want to ensure that this cannot happen, we need to apply only a small change to the trusted base (see Listing 12).

By default, Rust structs are `Send`, but we may override the default implementation and explicitly declare `Sec<T>` **not** to be `Send`. CRO states require `Send` and consequently the Rust compiler rejects all attempts to store a `Sec<T>` (e.g., `e.get(a))` at compile time. The same applies to asynchronous messages, and thus when using this approach, decrypted data cannot live longer than the authorization ticket.

Looking further at Listing 8, we find that the `sec_add_u32` function operates on the `Sec<T>` type and requires encrypted data to be decrypted before passing on. With trait objects in Rust, we could implement `sec_add_u32` for any type that allows access to `T` with an additional `Auth` parameter. The advantage is that the decryption only takes place at the instant of the function execution and limits the exposure to side channel attacks. However, a drawback is the increased complexity and the impeded ability for the compiler to generate zero-cost abstractions, because the Rust compiler introduces dynamic dispatch only for trait objects.

Another possible extension is to associate each CRC component with an authorization level. This allows us to statically differentiate between partitions of the system at design time and give a base authorization that can be temporarily raised. Moreover, we may associate each `Sec<T>`/`Enc<T>` with an `Auth` level providing precise control over the data access. Note, secure software implementations do not require any of these extensions, they just provide additional means to manage granularity.

In the setting of mixed critical systems, our framework allows design time analysis of security aspects. The topology of the system statically defines the delegation of authorization, and thus our framework effectively mitigates the need for run-time monitoring of security breaches. In effect, we can fearlessly introduce untrusted code for low critical subsystems with jeopardizing neither system safety nor security.

6.6 Comparison to C/C++

We exploit the borrow semantics, the lifetime semantics, and the possibility to prohibit the execution of unsafe user code in Rust programs to establish a statically verifiable security architecture. Following our approach, the Rust compiler ensures in a system built on such a proposed trusted base that no stealing (borrow semantics) or faking (no unsafe user code) of authorization can occur as well as an authorization has a guaranteed temporal validity with well-defined life span (lifetime semantics).

When it comes to the system level languages C/C++, there is no concept like borrow semantics. Memory can freely be aliased because the compiler is not rejecting multiple references to the same memory location. In effect, it is impossible for the compiler to statically deduce a lifetime for a memory location and thus eventually drop the reference and free the memory. In contrast to Rust, where we utilize the lifetime semantics for a guaranteed temporal validity of authorization tokens, this is not possible in plain C/C++. On the other hand, the C++ Standard Library includes *smart pointers* with the special pointer type `unique_ptr`. It essentially provides the same functionality as the Rust ownership model and supports the RAII (Resource Acquisition Is Initialization, [13]) programming principle. Such pointers indicate unique ownership of the memory they reference to and the memory is automatically freed when the pointer goes out of scope. However, the big difference to Rust is the validation point. While Rust incorporates the ownership model into the language, it can be statically verified during compilation. On violation, smart pointers in C++ cause a run-time error.

C/C++ does not support the segmentation of code into safe and unsafe partitions. We utilize this functionality of Rust to assure at compile time that no user code is able to generate, copy, or store authorization tokens. In C/C++, the same assurance can only be achieved by either applying static code analysis (e.g., formal methods) or verifying the authenticity of all authorization tokens by the trusted base on each usage during run-time. But the run-time token verification generates computational overhead and requires to carry additional information along with an authorization token, e.g., a private key signature of the token from the trusted base.

7 Memory safety of the Rust CRC framework

The pillar of the Rust memory model is **avoiding mutable aliasing** (referred to as the *invariant* in the following). As we provide a safe API, user code does not contain any `unsafe` fragments, and hence the `rustc` compiler grants memory safety. CRO connections, which the build script generates with `unsafe` code, are outside the knowledge of the compiler. Consequently, we have to ensure that the `unsafe` fragments preserve the invariant.

7.1 Synchronous messages

Each method receives a `&mut self`, a mutable reference to its state. Any synchronous message chain, for which an object o appears more than once, generates a mutable alias to the state of o and hence the build script has to reject it at compile time.

7.2 Asynchronous messages

The current implementation statically allocates a single element buffer for each asynchronous connection per CRO instance. A static mutable variable, which is hidden from the user, passes the message payload by value. A data race may occur if the sender (writer) preempts the dispatcher (reader). We handle this case by *panicking* the sender. An alternative option is to use an SRP resource for the buffer that ensures race free access[8].

7.3 Peripheral access

Let us assume a system with two objects A and B, which have access to the same peripheral P and a connection between output port op of object A and input port ip of object B. Let us further assume the method associated to the input port ip of object B claims the peripheral P. If in this system a method of object A claims the peripheral P and sends a synchronous message within this claim block through the output port op to the input port ip, we end up aliasing the reference to the register block of P. This is, however, **not** a problem because a `claim` returns an immutable reference (`&T`) to the register block, which upholds the invariant.

[8]However, this does not prevent a message payload to be overwritten before it has been dispatched. Therefore, further system wide timing analysis and potentially larger buffers are required.

7.4 Leaking of references

Passing data by reference in Rust is memory safe by construction. The borrow checker, one of the `rustc` compiler passes, is in charge of rejecting the use of invalid references at compile time It does this by tracking the *lifetime* of each memory location. In Rust, lifetime refers to the lexical scope for which access to a memory location is valid. The special lifetime identifier `'static` indicates in Rust that the memory location is valid for the entire program.

In our CRC framework, it is possible to pass data by reference in a synchronous message but not in an asynchronous message. The compiler can trace the lifetime of data across synchronous messages because they run in the same execution context. On the other hand, asynchronous messages run in different execution contexts. Semantically, a reference passed in an asynchronous message has to be valid for the span of both execution contexts. This cannot be verified at compile time and thus the compiler rejects it.

7.4.1 Leaking of peripherals

Peripherals provide a `claim` interface, which grants access to the peripheral register block only within the closure passed to it. The borrow checker does not allow references to escape from the closure.

7.4.2 Leaking through static variables

The Rust compiler prohibits to pass references between objects outside the message passing mechanism of the CRC framework. Such an operation requires to store the reference in a global (i.e., `static mut`) variable and load it from there. The compiler rejects this because static variables con only store *values* with `'static` lifetime and *references to values* with `'static` lifetime. E.g., the compiler rejects to store a reference to a stack allocated variable in a static variable. Apart from the lifetime problem, it is also `unsafe` to read, write, or modify `static mut` variables because the accesses to them are not synchronized. In conclusion, our CRC framework upholds the Rust memory invariants if we reject systems with synchronous message cycles (see Section 7.1) and ensure race-free execution with SRP [9].

8 Demonstration and performance analysis

For the design and measurements in this section, we used a Cortex-M3 microcontroller on a *Blue Pill* development board [14] running at 8 MHz and with zero memory wait states. Figure 2 depicts the example system implementation utilizing our proposed framework and Figure 3 illustrates the toolchain of our CRC framework.

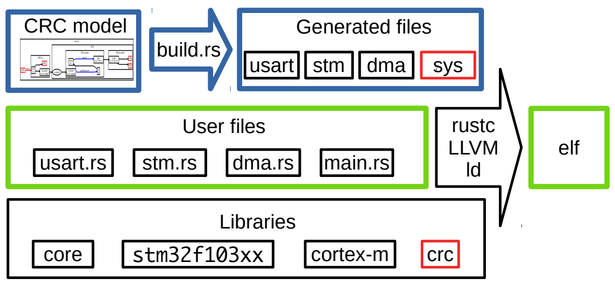

Figure 3: The `build.rs` build script is at heart of our toolchain. It analyzes the CRC/CRO model and generates the port binding proxies and system configuration, i.e., the statically allocated state and message memory. The user files that implement the CRO application logic contain no `unsafe` code. Among dependencies, the `crc` library contains the hardware related resource protection and scheduling primitives. `rustc` and `LLVM` compile all files and libraries for the `binutils-ld` linker to build a monolithic `elf` binary.

```rust
pub unsafe fn claim<R, F>(
    nvic_prio_bits: u8,
    ceiling: u8, f: F) -> R
where
    F: FnOnce() -> R,
{
    let max_priority = 1 << nvic_prio_bits;
    let old = basepri::read();
    let hw  = (max_priority - ceiling)
                    << (8 - nvic_prio_bits);
    basepri_max::write(hw); // sys ceiling
    let r = f();
    basepri::write(old);
    r
}
```

Listing 13: Resource protection with `claim`.

8.1 Characterization of overhead

In order to characterize the overhead, we performed a set of clock cycle accurate measurements with a 100% repeatability between runs. For all measurements, we compiled the code in `-release` mode.

The `claim` interface, as depicted in Listing 13, has an overhead of 4 clock cycles (call to return). We also observed this overhead when invoking object methods because the system applies the same `claim` mechanism to prevent data races on the object state. Our implementation of `claim` closely follows [3] and enforces compiler barriers around the critical section.

Synchronous messages are plain function calls, which allow to inline the code. In many cases, `rustc` opts to inline and eliminate the overhead of a function call. It also enables further optimization because it gives the compiler more local information about the behavior of the program.

Enqueuing an asynchronous message takes 20 clock cycles plus the time required to copy the message payload from the stack into a statically allocated buffer.

Dispatching asynchronous messages has a per message overhead of 26 clock cycles plus the time required to copy the message payload from a statically allocated buffer back into the stack.

The interrupt latency (11 clock cycles) plus the proxy overhead claiming the target object and entering the user code (3 clock cycles) determines the external event latency. It amounts on an 8 MHz MCU to $1.75us$.

The model offers a plethora of methods for response time analysis, taking into consideration preemption and blocking [7] as well as offsets [8]. Scheduling and resource protection overhead is $O(1)$, i.e., free of run-time dependencies. Hence, further scheduling analysis can utilize the characterizations as direct input.

8.1.1 *Example system*

We designed our example system with reactivity in mind. The environment and the application at hand set the timing constraints, which we specified in cycles as depicted in Figure 2.

The USART operates at $115.2kbps$, which is roughly $87us$ or 696 cycles to serve an arriving byte. For simplicity, we assume a single buffer.

The LED array consists of 24 daisy chained WS2812B-LEDs. In order to update each LED with a unique RGB value, the DMA peripheral sends a non-return-to-zero bit stream and latches the output on the end of the frame by holding the data line low for at least $50us$. The DMA operates at $400kHz$, which results in a transfer time of $1.5ms$. This is on the safe side at half of the maximum specified operation rate.

The design ensures that blocking will not be an issue, because the state of the STM is completely decoupled from the state of the DMA[9]. Alternatively, we could use an asynchronous message between the USART and the STM in the LED component to achieve the same effect of decoupling. When we see our LED application as a freestanding and re-usable component, there is no restriction on how to implement it, both synchronous and asynchronous calls work equally well.

Looking at the STM CRO, we set the interarrival time of `transition` events in the system to $10ms$, i.e., a frequency of $100Hz$. The number of preemptions during a $10ms$ period is roughly 115. I.e., the `transition` suffers $115 * C(receive)$ in the worst case. We measured a worst case execution time of 299 cycles for *receive*, amounting to a total of $4.3ms$.

The response time for a task is $r = C + P + B$, where C is the execution time, P is the preemption time (interference), and B is the blocking time. For the `transition`, we derive $r = 0.098ms + 4.3ms + 0$, which is a worst case estimation well under the required $10ms$ or 80000 cycles[10]. For this presentation, we conclude the response times of `receive` and `on_command` to be clearly within their timing requirements and skip the precise analysis.

We measured a CPU utilization of 13.25% at the maximum animation speed (100 frames per second).

[9]The $1.5ms$ transfer period blocks the DMA, but there is only $87us$ in between two USART events. Hence, a synchronous (blocking) approach is not sufficient.

[10]Computing the actual busy period of `transition` and taking the USART parsing logic into account allows to derive a less pessimistic estimation. Not all character inputs yield the worst case behavior.

```
text   data    bss    dec    hex  filename
3974    196    620   4790   12b6  crc-test
```

Listing 14: `arm-none-eabi-size`

8.2 Memory usage

The system compiled in `-release` mode shows a $4kB$ Flash memory footprint and less than $1kB$ of RAM usage. Listing 14 displays the actual sizes.

The DMA buffer requires 601 bytes to store the non-return-to-zero bit encoding including a postamble of 25 zeros to latch the data to the WS2812B LED array. In the STM CRO we store the RGB values of each individual LED ($24 * 3 = 72$ bytes) and send it with an async message buffer. In total, this amounts to 745 bytes. The remaining allocated RAM memory of 67 bytes holds additional CRO states (USART, STM, DMA) and message structure overhead.

We conclude the abstraction to be memory efficient and zero-cost in comparison to a handwritten implementation.

9 Conclusions and future work

In this paper, we present a Rust based component model for concurrent programming along with a framework for analysis and code generation that produces efficient, memory safe, race- and deadlock-free executables for single-core Stack Resource Policy (SRP) based scheduling. As the main contribution, we show that the CRC model allows a secure by construction design of embedded software, covering authentication for operations as well as abstractions for safe and secure data containers. For the underlying CRC framework, we discuss soundness in regard to the Rust memory model and SRP invariants.

Other contributions include key design decisions for the ecosystem under development, a feasibility demonstration on an ARM Cortex-M3 target, and the characterization of run-time overhead for resource protection and scheduling primitives.

For the prototype, we manually carried out the timing analysis and timer queue generation. Current and future work includes the analysis of arbitrary timing offsets to determine safe (yet tight) bounds for the number of outstanding asynchronous messages and the synthesis of queuing and timer primitives.

Based on recent advances of the RustBelt formal model [15], we project a formalization and mechanized proof of correctness.

References

[1] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black, "Reactive objects," in *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pp. 155–158, 2002.

[2] The Timber Language, webpage. http://www.timber-lang.org, last accessed 2017-09-16.

[3] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming api and static priority srp kernel primitives," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pp. 110–113, IEEE, 2013.

[4] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, (New York, NY, USA), pp. 1:1–1:7, ACM, 2017.

[5] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "Gpu programming in rust: Implementing high-level abstractions in a systems-level language," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 315–324, May 2013.

[6] "The Rust Programming Language" book, webpage. https://doc.rust-lang.org/book/, last accessed 2018-02-05.

[7] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, pp. 67–99, Apr. 1991.

[8] J. Mäki-turja and M. Nolin, "Tighter response-times for tasks with offsets," in *In Proc. of the 10 th International conference on Real-Time Computing Systems and Applications (RTCSA'04*, 2004.

[9] P. Lindgren, M. Lindner, E. Fresk, D. Pereira, and L. Pinho, "RTFM-core: Language and Implementation," 2014. Embedded Systems Week, New Delhi, India.

[10] svd2rust, webpage. https://github.com/japaric/svd2rust, last accessed 2017-09-16.

[11] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 461–491, Aug. 2004.

[12] D. Kahn, *The Codebreaker: The Story of Secret Writing*. Macmillam, 1976.

[13] S. Meyers, *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[14] "Blue Pill" compact STM32F103 board, webpage. https://wiki.stm32duino.com/index.php?title=Blue_Pill, last accessed 2018-09-07.

[15] "RustBelt: Logical Foundations for the Future of Safe Systems Programming", webpage. http://plv.mpi-sws.org/rustbelt, last accessed 2017-09-16.