# Implementing an FF-A Secure Partition Manager in Rust

Bálint Dobszay & Imre Géza Kis
2024-05-16
Arm

Linaro Connect
MADRID 2024 | MAY 14-17 2024

# Introduction

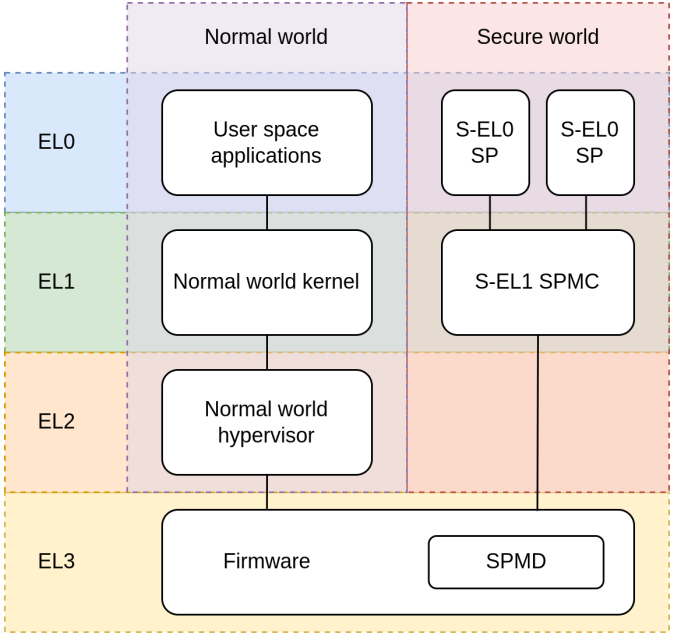
- Motivation
  - Learning Rust as a hobby vs. writing secure firmware as day job
  - How about firmware development in Rust?
  - The S-EL1 SPMC is a well-defined component by the FF-A specification
  - First goal: running Trusted Services Secure Partitions (SPs)
- Timeline
  - 2022-2023 – Development
  - 2023. Dec. – Published prototype on TrustedFirmware.org
  - Current state: gathering feedback
- Future maintenance TBD

# About Rust language

- The safety features of Rust makes it ideal for security critical environments
- Large portion of the vulnerabilities are caused by memory safety issues [1]
- Compile time checks, no garbage collector, performance similar to C [2]
- Cargo: standard build system & package manager
- LLVM based compiler (AArch64 has tier 1 support)
- Many major companies started to adopt Rust for new projects

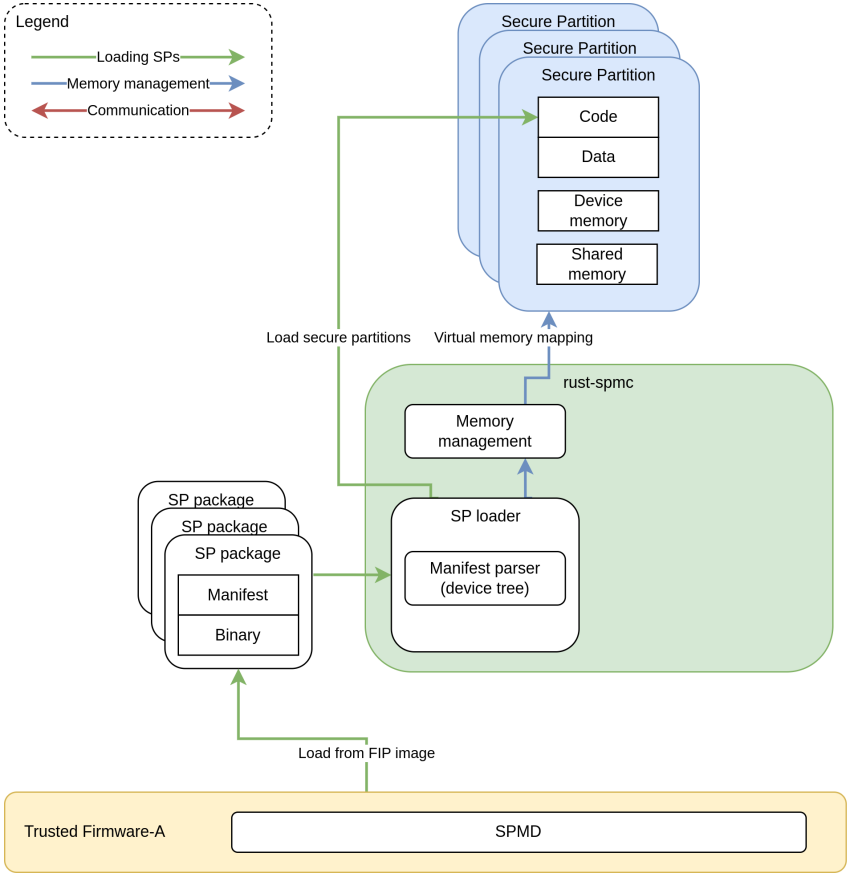
# Firmware Framework for Arm A-profile (FF-A)

- Defines the software architecture of firmware components
- Standardized communication protocol
  - Register ABI
  - Memory sharing primitives
  - Component discovery
- Offers isolation of components using Arm architectural features
- Secure Partition Manager (SPM)
  - Isolation of the Secure Partitions
  - Communication between Secure Partitions and Normal World components



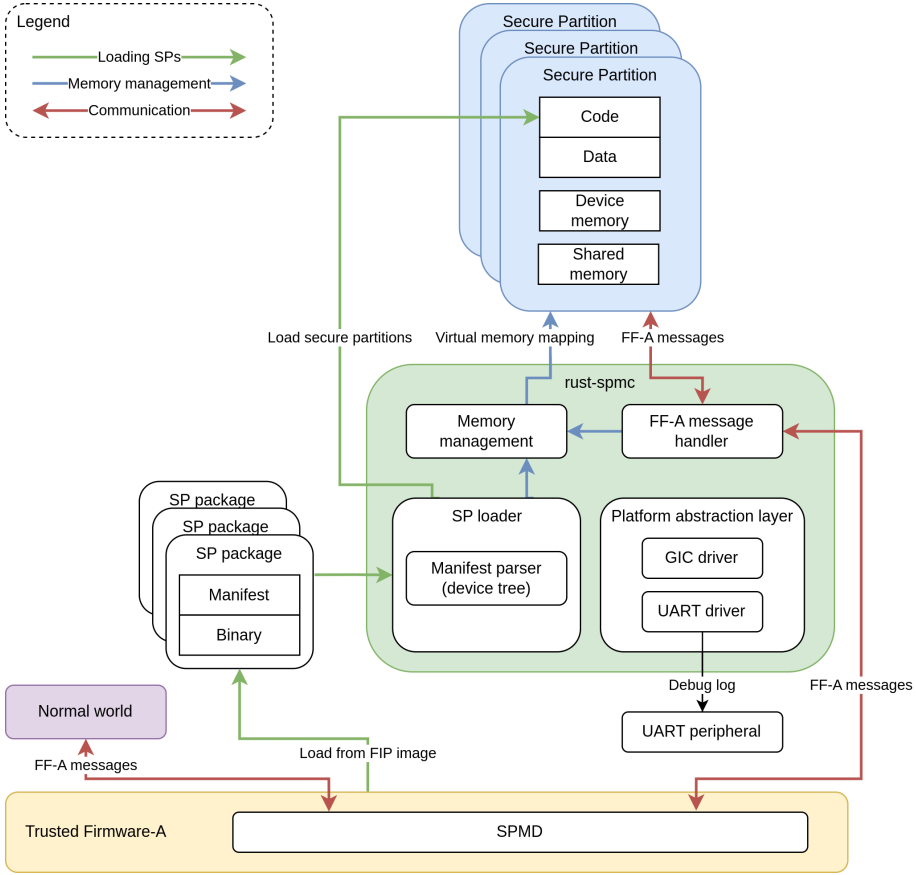
# SPMC structure (boot)

- Minimal low level startup code
- **Loading Secure Partitions**
  - Configuring based on their manifest
- **Isolation: Configure virtual memory mapping**
- Communication: Parse and forward FF-A messages
- Platform abstraction layer
  - UART driver, interrupt controller driver
- Multi-core operation
  - Run multiple SPs on different cores
  - Thread safety



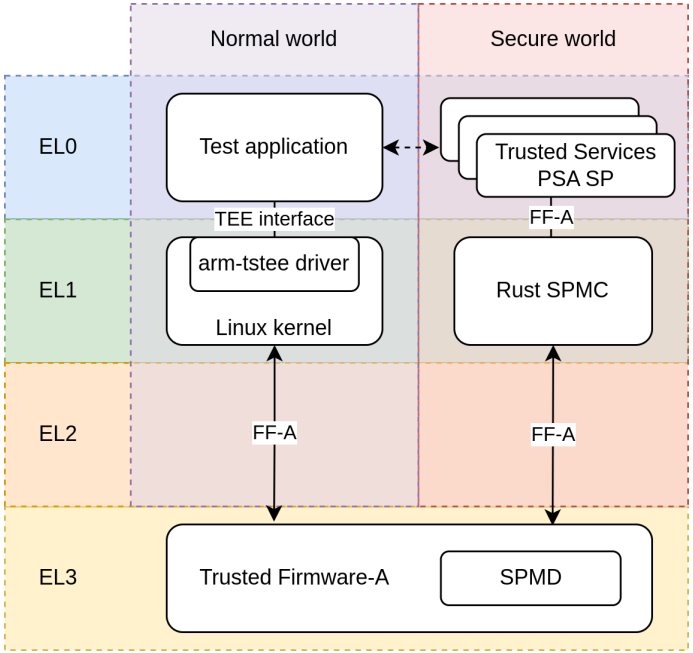
# SPMC structure

- Minimal low level startup code
- Loading Secure Partitions
  - Configuring based on their manifest
- Isolation: Configure virtual memory mapping
- **Communication: Parse and forward FF-A messages**
- **Platform abstraction layer**
  - UART driver, interrupt controller driver
- Multi-core operation
  - Run multiple SPs on different cores
  - Thread safety



# SPMC example integration

- End-to-end testing
- Arm FVP Base RevC platform
- Trusted Firmware-A
  - Implements the SPMD component
- Trusted Services S-EL0 SPs
  - PSA services reference implementation
- Normal world components
  - Linux (FF-A driver, TS-TEE driver)
  - Test applications from Trusted Services (includes PSA ACS)



# Conclusion (pros)

- **Mandatory boundary/null checks**
- Lifetime and ownership
- Efficient built-in collections
- Traits, structured code, compile time resolving
- Thread safe secondary core init
- Integrating assembly code is easy
- Cargo build system

```rust
let mut resource: Option<Resource> = None;

// This would cause a panic:
// called `Option::unwrap()` on a `None` value
// resource.unwrap().action();

resource = Some(Resource::new(5));

if let Some(res) = &resource {
    res.action();
}

resource.unwrap().action();
```

# Conclusion (pros)

- Mandatory boundary/null checks
- **Lifetime and ownership**
- Efficient built-in collections
- Traits, structured code, compile time resolving
- Thread safe secondary core init
- Integrating assembly code is easy
- Cargo build system

```rust
impl<'a> MappedBuffer<'a> {
    pub fn new(addr: usize, len: usize) -> Self {
      memory_map(addr, len);
      Self { buffer: [...] }
    }
    pub fn get_buffer(&self) -> &[u8] {
        self.buffer
    }
}

impl<'a> Drop for MappedBuffer<'a> {
   fn drop(&mut self) { memory_unmap([...]); }
}

fn example() {
   let buffer: &[u8];
   {
       let mapped_buffer = MappedBuffer::new([...]);
       buffer = mapped_buffer.get_buffer();
       // mapped_buffer is dropped here
   }
   println!("Buffer value: {:?}", buffer);

   // error[E0597]: `mapped_buffer` does not live long enough
}
```

# Conclusion (pros)

- Mandatory boundary/null checks
- Lifetime and ownership
- **Efficient built-in collections**
- Traits, structured code, compile time resolving
- Thread safe secondary core init
- Integrating assembly code is easy
- Cargo build system

- Collections
  - Sequences: Vec, VecDeque, LinkedList
  - Maps: BTreeMap
  - Sets: BTreeSet
  - Misc: BinaryHeap
- Iterators
  - Finding, filtering items
- Available for no_std

# Conclusion (pros)

- Mandatory boundary/null checks
- Lifetime and ownership
- Efficient built-in collections
- **Traits, structured code, compile time resolving**
- Thread safe secondary core init
- Integrating assembly code is easy
- Cargo build system

- Implementing interfaces
- Type safety

```
pub trait PlatformInterface {
    type Context: ContextInterface;
    type NormalWorld: NormalWorldInterface;
    const CORE_COUNT: usize;

    fn init_log();
    fn init_heap();
    fn create_page_pool() -> PagePool;
    fn create_kernel_space(page_pool: PagePool) ->
    KernelSpace;
    fn init_interrupts();
    fn init_core_interrupts();
    fn get_current_el() -> ExceptionLevel;
}
```

# Conclusion (pros)

- Mandatory boundary/null checks
- Lifetime and ownership
- Efficient built-in collections
- Traits, structured code, compile time resolving
- **Thread safe secondary core init**
- Integrating assembly code is easy
- Cargo build system

- Built-in Send and Sync trait
- The compiler prevents sending or sharing non-thread-safe objects between threads
  - Example: wrapping object into Mutex makes it safe

```
let spmc = Arc::new(Spmc::new([...]).unwrap());
spmc.init().unwrap();

for i in 1..Platform::CORE_COUNT {
  let local_spmc = spmc.clone();
  let local_kernel_space = kernel_space.clone();
  set_sec_core_entry(i, move |core_index: usize| {
        local_kernel_space.activate();
        Platform::init_core_interrupts();
        local_spmc.main_loop();
  })
}

spmc.main_loop();
```

# Conclusion (pros)

- Mandatory boundary/null checks
- Lifetime and ownership
- Efficient built-in collections
- Traits, structured code, compile time resolving
- Thread safe secondary core init
- **Integrating assembly code is easy**
- Cargo build system

- No need for manual assembler configuration

```
core::arch::global_asm!(include_str!("startup.S"));
core::arch::asm!(
    "msr ttbr0_el1, {0}
     isb",
     in(reg) ttbr_value)
```

# Conclusion (pros)

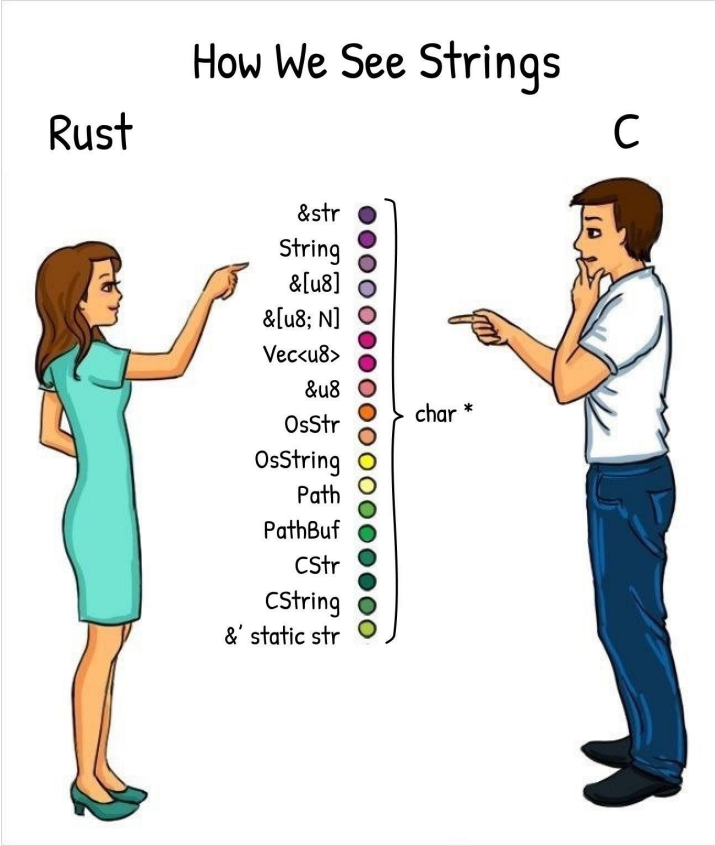
- Mandatory boundary/null checks
- Lifetime and ownership
- Efficient built-in collections
- Traits, structured code, compile time resolving
- Thread safe secondary core init
- Integrating assembly code is easy
- **Cargo build system**

- Standard but not mandatory
- Encourages code reuse
- Easy cross compilation

`--target aarch64-unknown-none-softfloat`

# Conclusion (cons)

- Lifetimes vs hardware
  - Storing Rust allocated resources in hardware defined structures
  - Requires manual lifetime handling
- Fight against the compiler
  - Propagating explicit object lifetime
- Heavy language syntax and standard features

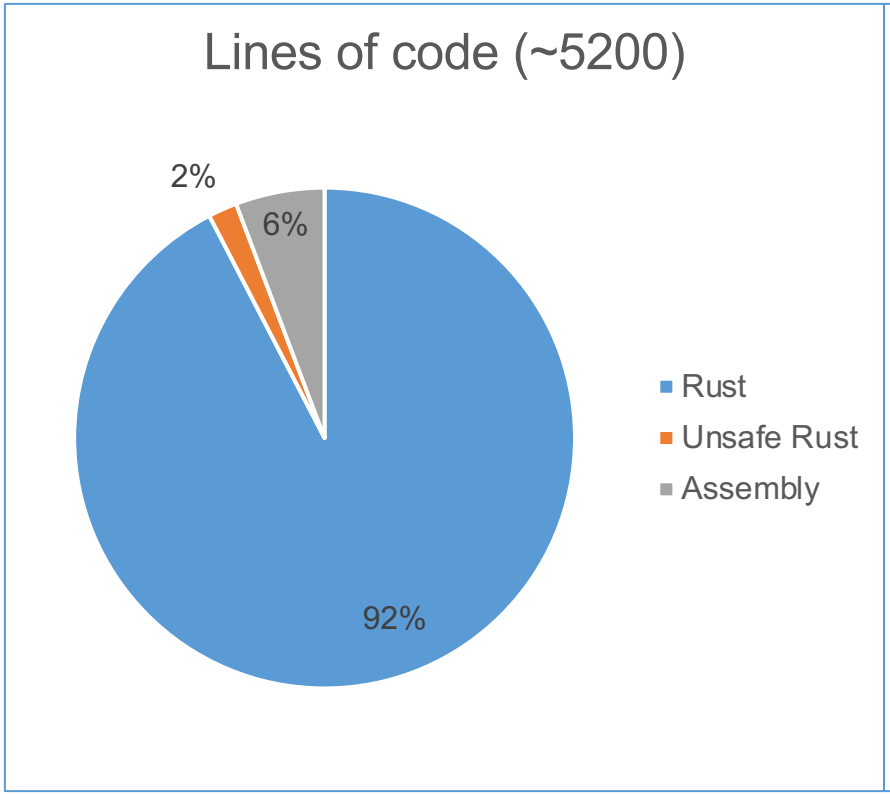


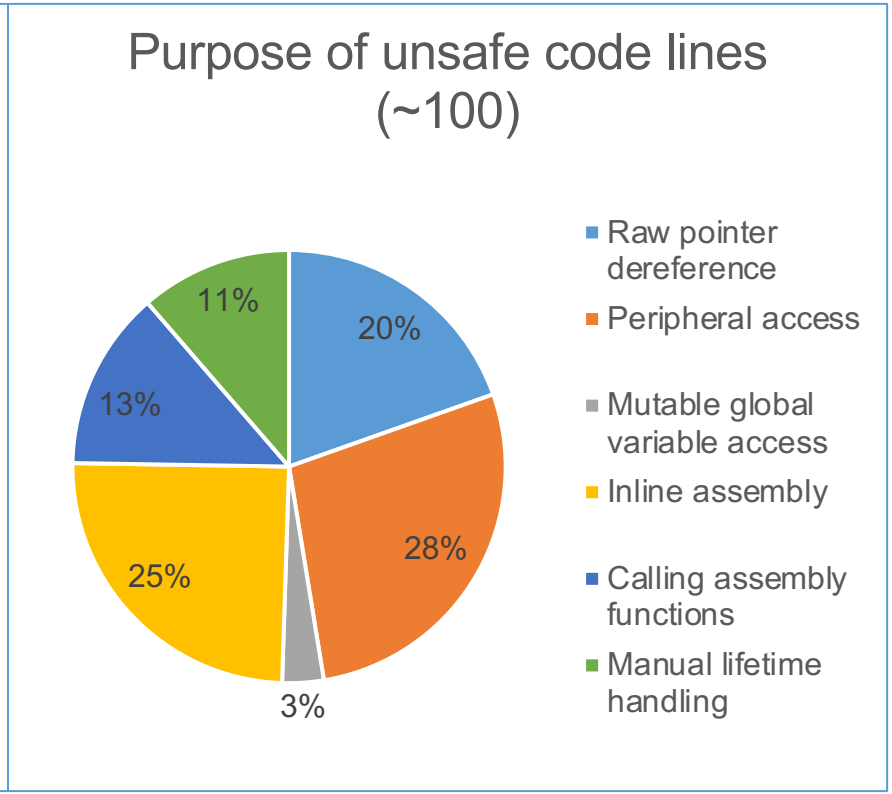

Source: unknown (probably Reddit?)

# Unsafe

- Disable some of the compiler checks
- Contained and minimalized
- Makes review effort more focused
- General advice
  - Do not use
  - Use existing crate which wraps the required feature in a safe way
- In firmware it’s inevitable

# Unsafe


Lines of code (~5200)
2%
6%
92%
Rust
Unsafe Rust
Assembly


Purpose of unsafe code lines (~100)
11%
20%
13%
28%
25%
3%
Raw pointer dereference
Peripheral access
Mutable global variable access
Inline assembly
Calling assembly functions
Manual lifetime handling

# Summary

- Experimental proof-of-concept project
  - FF-A feature parity for S-EL1 SPMC
  - Running S-EL0 Secure Partitions
- Rust has many benefits
- Future plans
  - Hardening, testing
  - Implement full FF-A feature set
  - Investigate deployment to S-EL2

# Resources

- rust-spmc git repository (code, documentation, build & test instructions)
- Arm Firmware Framework for Arm A-profile
- Rust language
- Trustedfirmware.org mailing list
- Trustedfirmware.org Discord

Linaro Connect
MADRID 2024 | MAY 12-17 2024

# Thank you