# Encapsulated Functions:

## Fortifying Rust's FFI in Embedded Systems


Leon Schuermann, *Princeton University*
Arun Thomas, *zeroRISC Inc.*
Amit Levy, *Princeton University*

PRINCETON UNIVERSITY
zeroRISC


## Overview

**Encapsulated Functions** is a framework for safely invoking *untrusted* code in a memory-safe system with minimal overhead. Encapsulated Functions combines

➡ hardware-based memory protection mechanisms present in modern microcontrollers with

➡ a set of safe type-abstractions

to facilitate safe interactions with untrusted and unmodified third-party libraries.

## Motivation

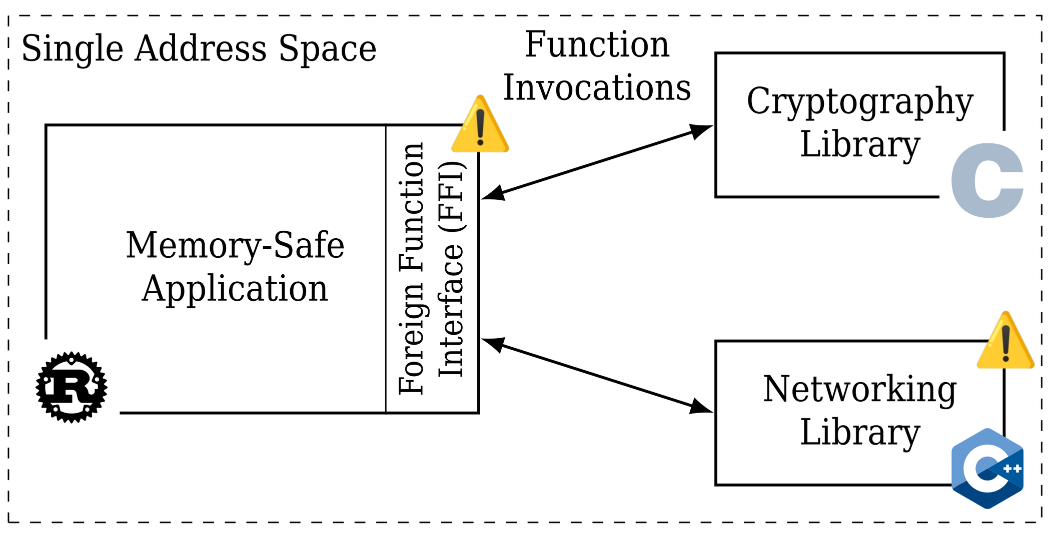
Rust is suitable to replace C or C++ for safety- and security-critical embedded systems. It provides memory safety without compromising runtime compute & memory overhead.

Still, it is often infeasible to rewrite such critical software in Rust:

➡ Extensive certification requirements mean that rewrites incur vast development efforts and costly re-certification.

➡ Rust is missing required infrastructure, e.g. certified compilers / standard libraries.

We can expect a *gradual* transition to memory-safe software:




**Foreign code endangers the system's safety**:

➡ A buggy library can arbitrarily modify the safe language's memory and thus violate its safety requirements.

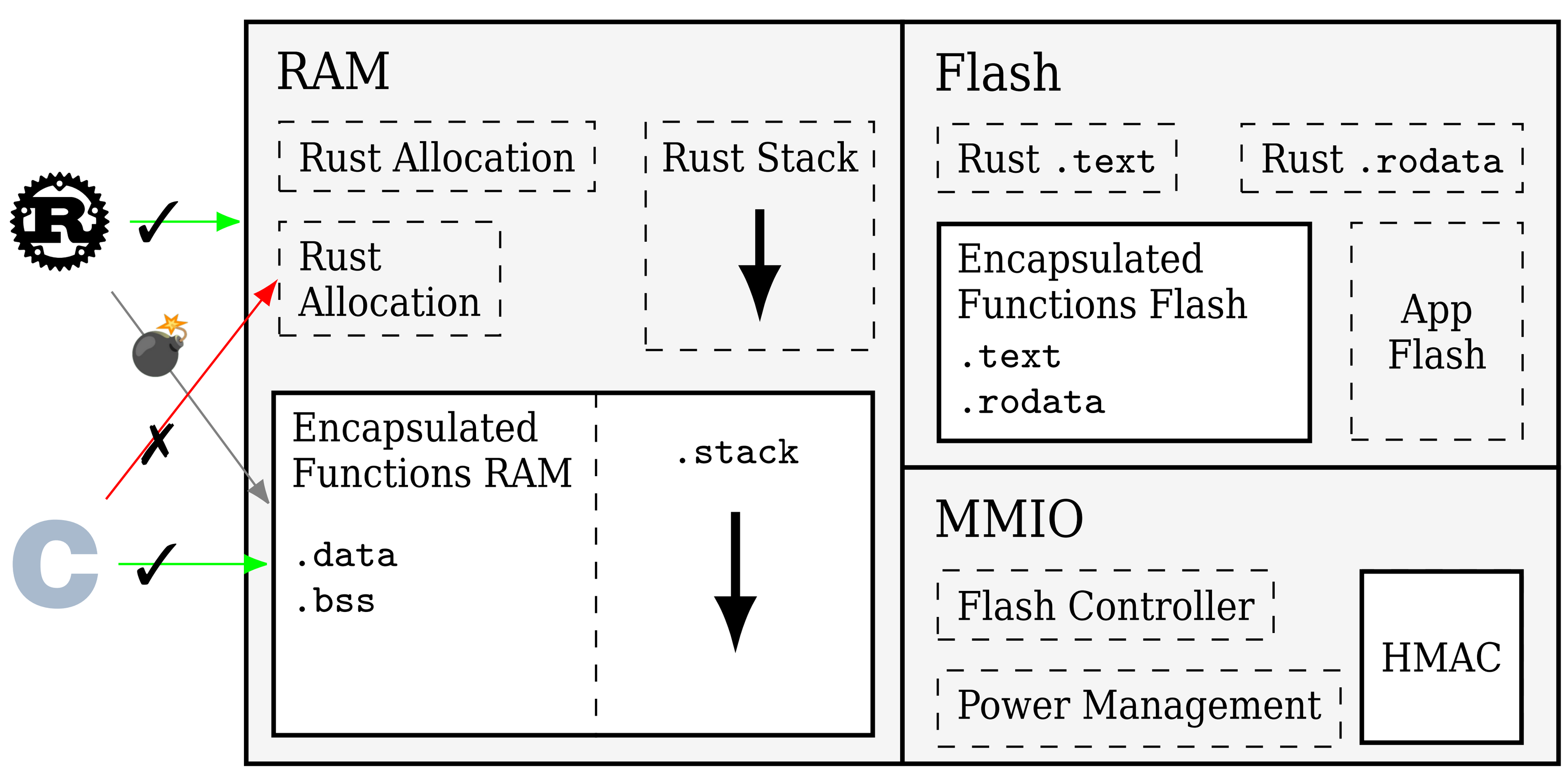
➡ Interactions between languages can cause unsoundness due to differing cross-language semantics (e.g. *valid values*).

## Lightweight Context Switches

➡ Use memory protection mechanisms of microcontrollers to isolate untrusted code (e.g. ARM Cortex-M MPU, RISC-V PMP).

➡ Coarse-grained protection regions: RAM, Flash, MMIO periph.

➡ Lightweight Context Switches optimize over executing regular processes and maintain synchronous function call semantics.




➡ Accesses into foreign memory can still violate Rust's soundness!

## Safe Type-Abstractions

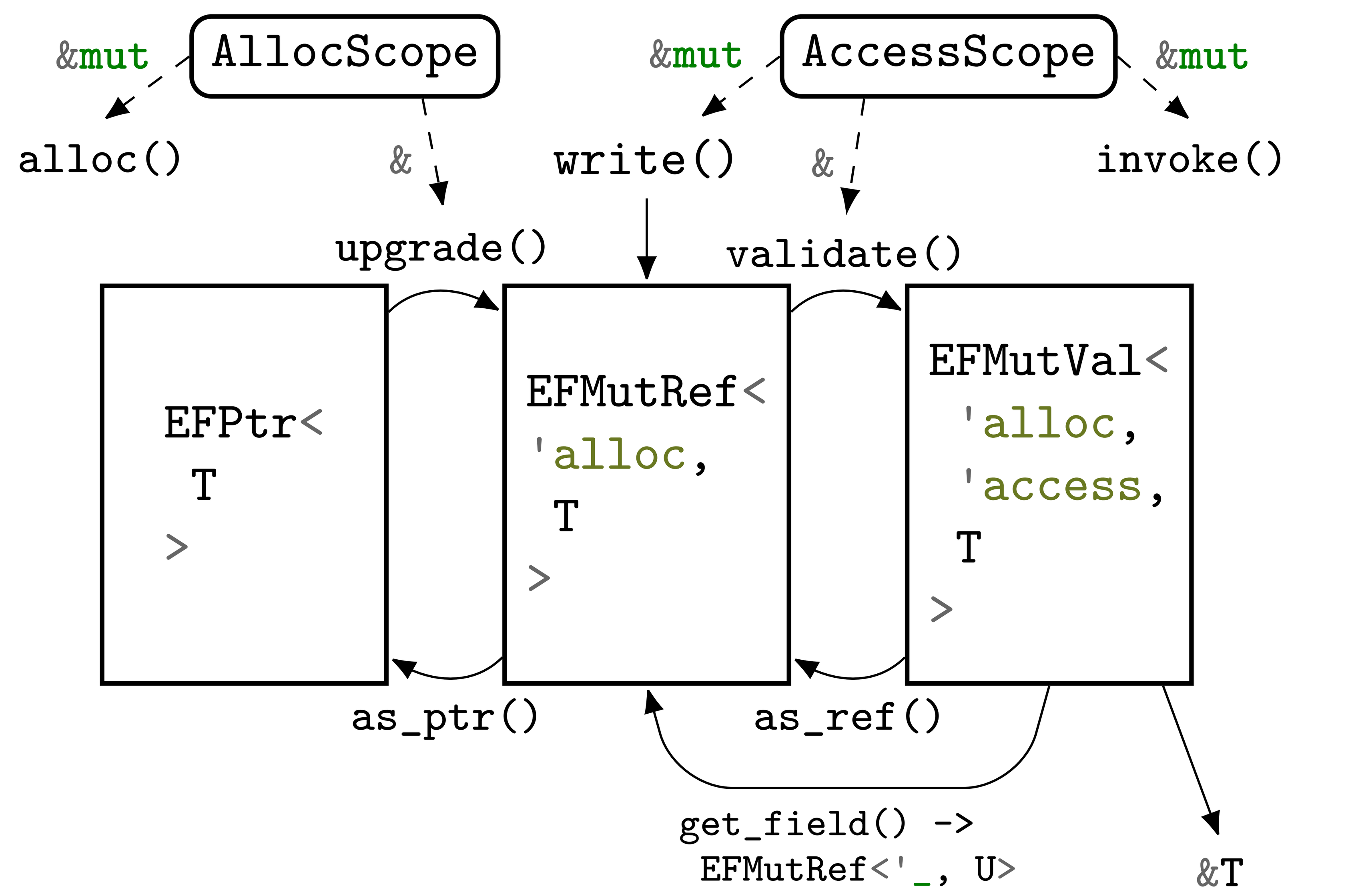
Isolating foreign / untrusted code is not sufficient—differing cross-language semantics can break safety guarantees in subtle ways:

| Mutable Aliasing | Null Pointers | Valid Values |
|---|---|---|
| *is disallowed in Rust* | *must not be coverted to references* | *different across languages (e.g. bool)* |

Introduce a set of type abstractions that:

➡ **Eliminate hazardous cross-language invariant violations.**

➡ Use the *typestate*-paradigm to represent *validation states* of references, tied into memory allocation & lightweight context switch mechanisms through *Rust lifetimes*.




➡ `EFPtr`: Raw pointer type, safe to pass across FFI boundaries

➡ `EFMutRef`: Reference type validated to be well-aligned & wholly contained in mutably accessible foreign memory
*Bound to an allocation scope, forced out of scope on allocation changes.*

➡ `EFMutVal`: Reference type validated to adhere to the EFMutRef restrictions & contain a valid instance of type T
*Bound to an access scope, forced out of scope on writes / invocations.*

## Case Study

We integrate Encapsulated Functions into the Tock embedded OS, written in Rust. We use it to integrate the OpenTitan *CryptoLib*, a C-based library providing hardened implementations of cryptographic algorithms and hardware drivers.

opentitan
Tock

➡ Overhead of *Lightweight Context Switches*:

| RISC-V PMP Pre-configured | Lightweight Context Switch | Tock Process Context Switch | |
|---|---|---|---|
| ✓ | 120 instr. | 530 instr. | 23% |
| ✗ | 360 instr. | 770 instr. | 47% |

➡ Integrates into standard Tock kernel HMAC interfaces

➡ Direct access to hardware peripherals (MMIO HMAC core)

➡ Works alognside regular Tock processes, shares RISC-V PMP

## Future Work

➡ Extensive expressiveness and performance evaluation

➡ Port to other Oses, architectures, memory protection schemes

➡ Support multi-threaded execution

➡ Automatic generation of bindings for `EF*` types