RUST-Encoded Stream Ciphers on a RISC-V Parallel Ultra-Low-Power Processor

Francesco Barchi, Giacomo Pasini, Emanuele Parisi, Giuseppe Tagliavini,
Andrea Bartolini, Andrea Acquaviva

PARMA-DITAM 23

Outilne

- **Introduction**
- Background
- Method
- Results

Introduction

Cyber-Physical Systems (CPS) challenges have become increasingly evident in these last years.

Growing complexity of these devices

Increasingly interconnected

Able to act and manipulate the surrounding reality

This was possible through the contribution of new powerful and energy efficient:

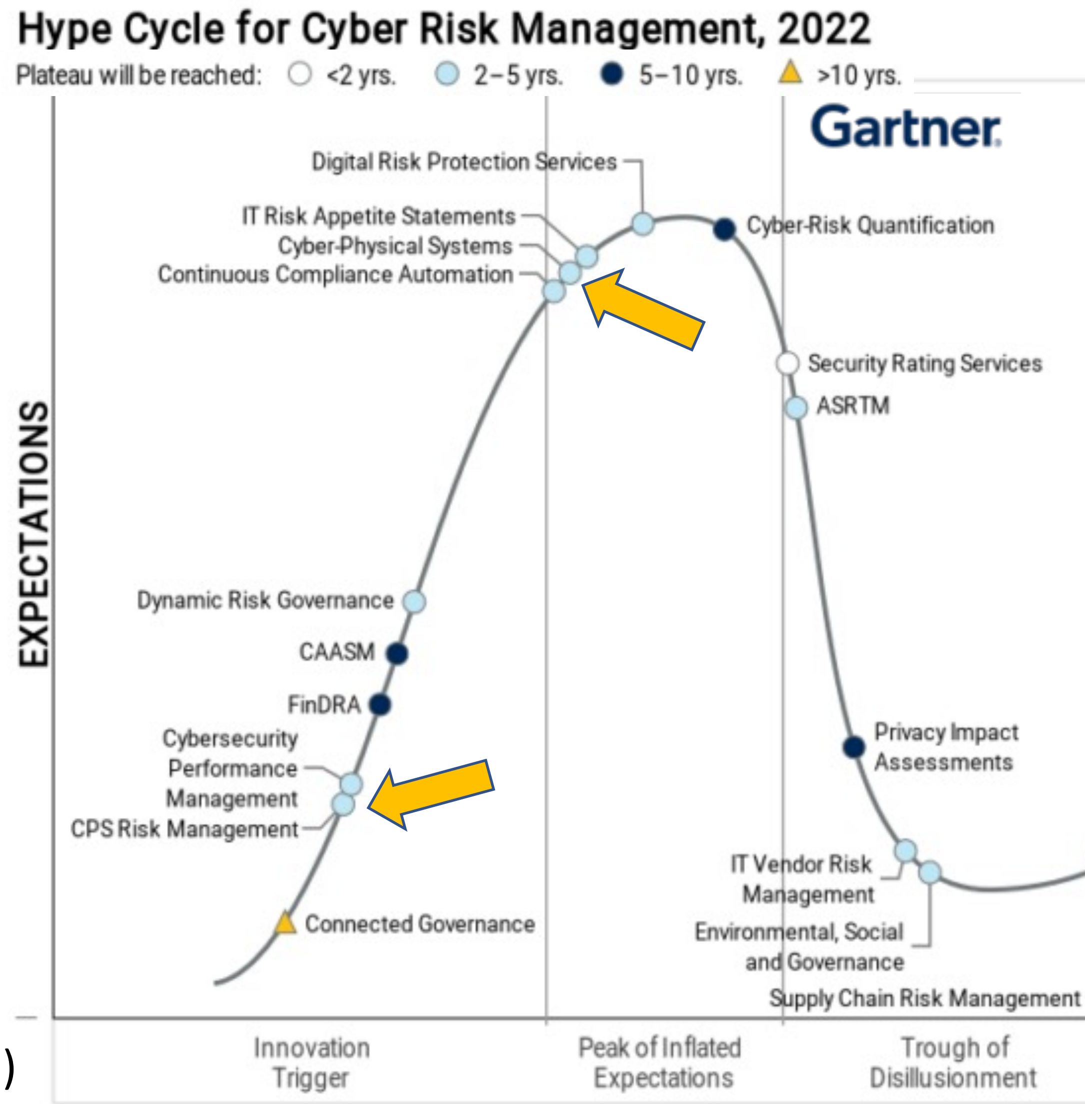
- Systems of Chips (SoC)
- Communication Wireless technologies (e.g., NB-IoT, 5G)

Introduction

Gartner® Hype Cycle™ for Cyber Risk Management, 2022:

- **CPSs** are reaching the peak of inflated expectations.
  - Katell Thielemann describes CPS as «Engineered systems that orchestrate sensing, computation, control, networking and analytics to interact with the physical world.»
  - Inflated expectations: « … phase of overenthusiasm and unrealistic projections … well-publicized activity by technology leaders results in some successes ….»

- **CPS risk management** is an innovation trigger; it started its ascent among the potentially relevant topics for the next five years.



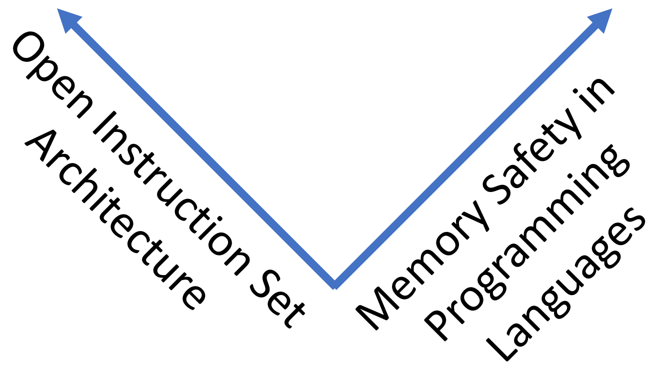
Source: Gartner (July 2022)

Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:

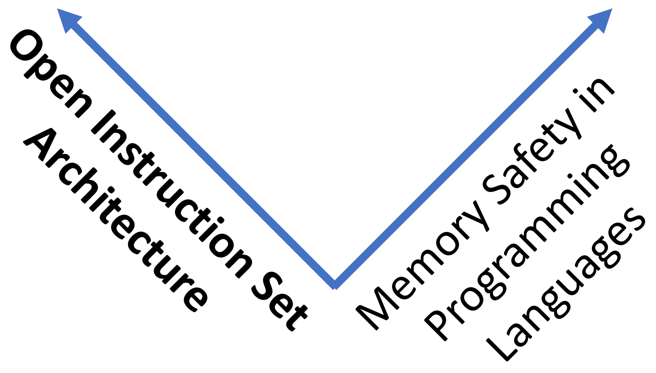
Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:



Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:

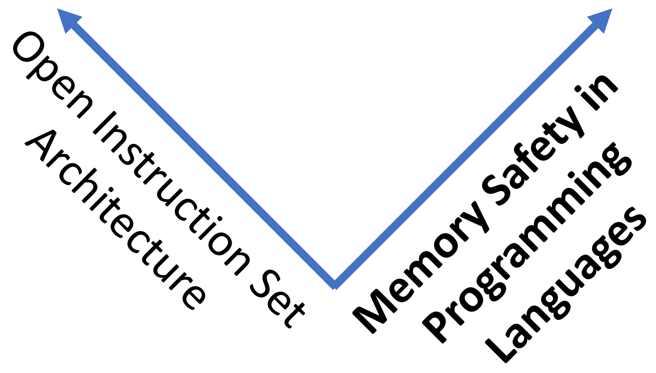


An open ISA (Instruction Set Architecture) contributes to the development of more security in system-on-chips:

- It allows for greater **transparency** and **collaboration** in the development and review process.
- The design and implementation of the ISA can be examined and audited by a larger community, making it more likely that any vulnerabilities or weaknesses will be discovered and addressed.
- Open source allows for implementing security features and protocols that may not be present in proprietary ISAs.

Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:

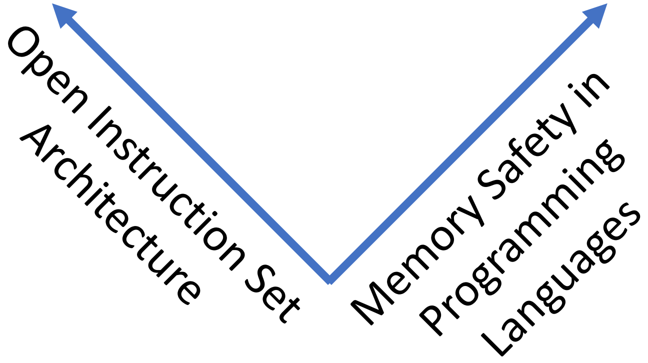


Memory safety in a programming language contributes to the development of more security in CPS, preventing common programming errors that can lead to security vulnerabilities.

- **Buffer overflow prevention**: a common source of security vulnerabilities. Bounds checking ensures that data is written only to the memory allocated.
- **Pointer safety**: preventing common programming errors such as null pointer dereferences and use-after-free bugs.
- **Memory leak prevention**: Memory safety features ensure that memory is properly allocated and freed, preventing memory leaks and ensuring that the system does not run out of memory.

Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:

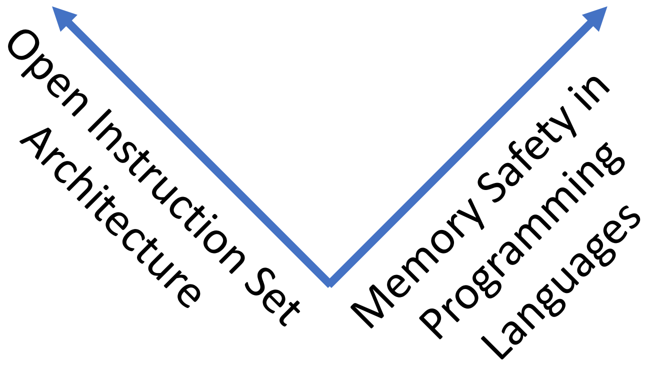


RISC-V is an open-source ISA:

- Transparency and collaboration in the development and review process
- Customisable: allows for implementing security features and protocols that may not be present in proprietary ISAs.

Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:



RISC-V is an open-source ISA:

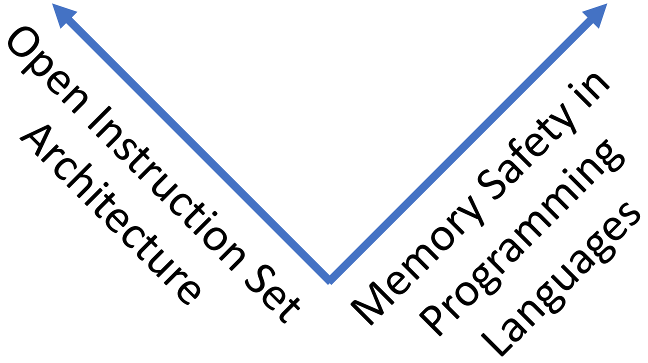
- Transparency and collaboration in the development and review process
- Customisable: allows for implementing security features and protocols that may not be present in proprietary ISAs.

RUST was designed with security in mind.
It catches the majority of memory mistakes at compile time.

- Avoid undefined behaviour
- Avoid memory corruption

Introduction

The risk management of CPSs is a complex topic.
Among other things, it relies on the software and hardware security employed in developing the system. In our vision, we have two major fields of research that contribute orthogonally to improving security:



RustCrypto/stream-ciphers

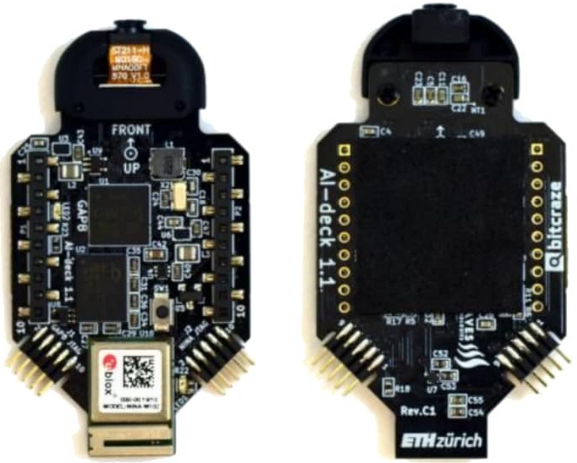
In this work, we face the interoperability challenge of compiling and executing **RUST-encoded software** in an existing **RISC-V platform**: GreenWaves' **GAP8**. This SoC is a parallel ultra-low-power (PULP) system composed of a cluster in a chip.

Introduction

We developed a framework to integrate the RUST library into an existing software/hardware ecosystem. A use-case scenario is encrypted video surveillance in micro-UAV

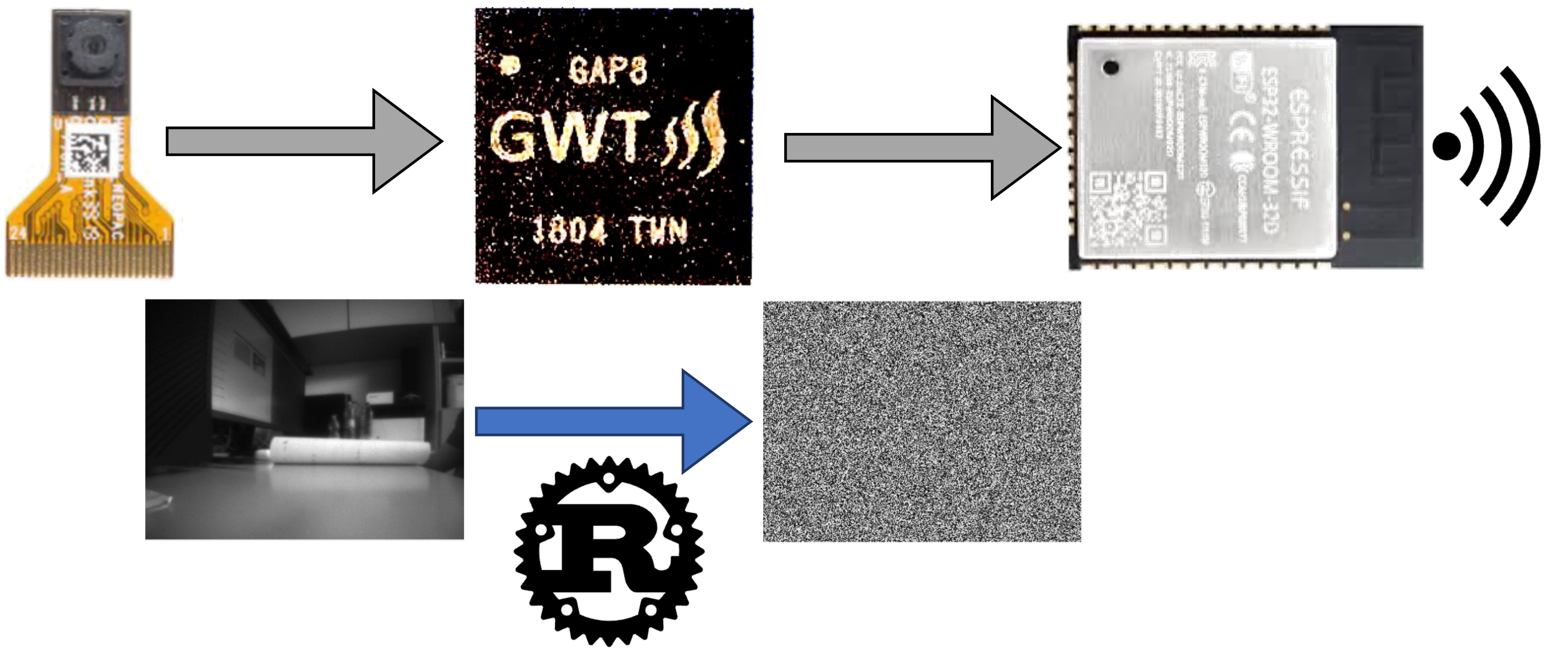
CPS
Crazyflie

AI-Deck



Example application
Encrypted video surveillance in micro-UAV

RISC-V SoC

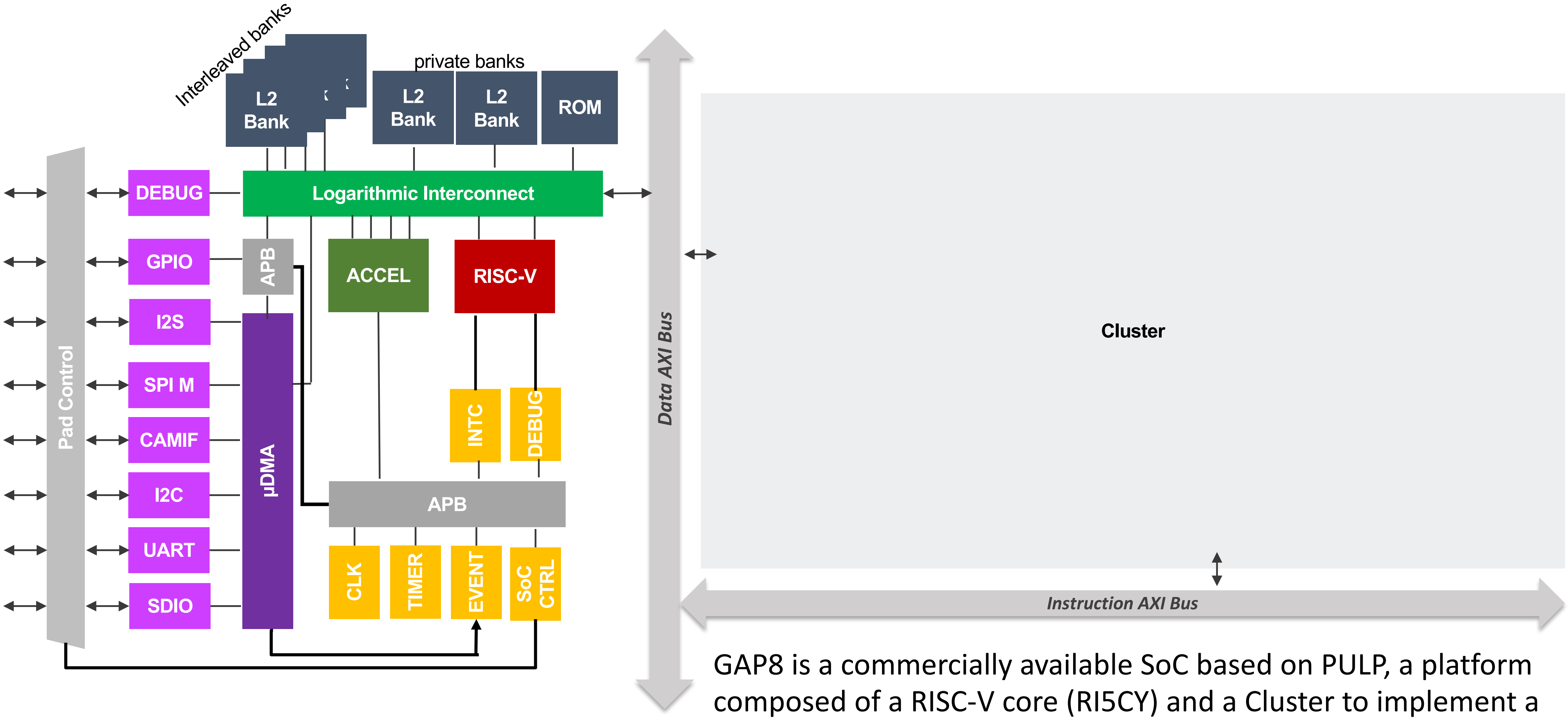


Encrypt video stream using a RUST library

Outilne

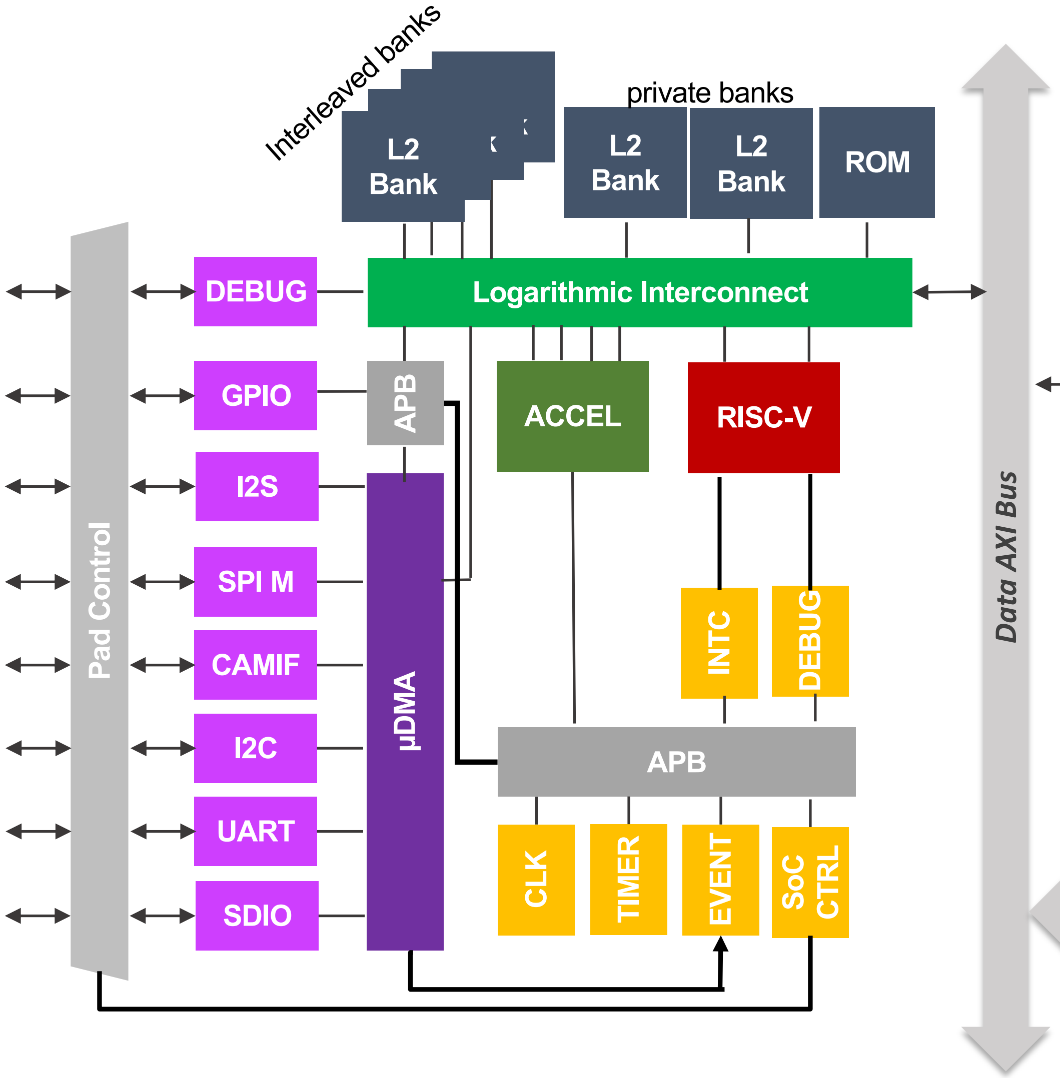
- Introduction
- **Background**
  - **Hardware: PULP and GAP8**
  - Language: RUST and RUST for embedded devices
  - Software: Stream ciphers and Chacha20
- Method
- Results

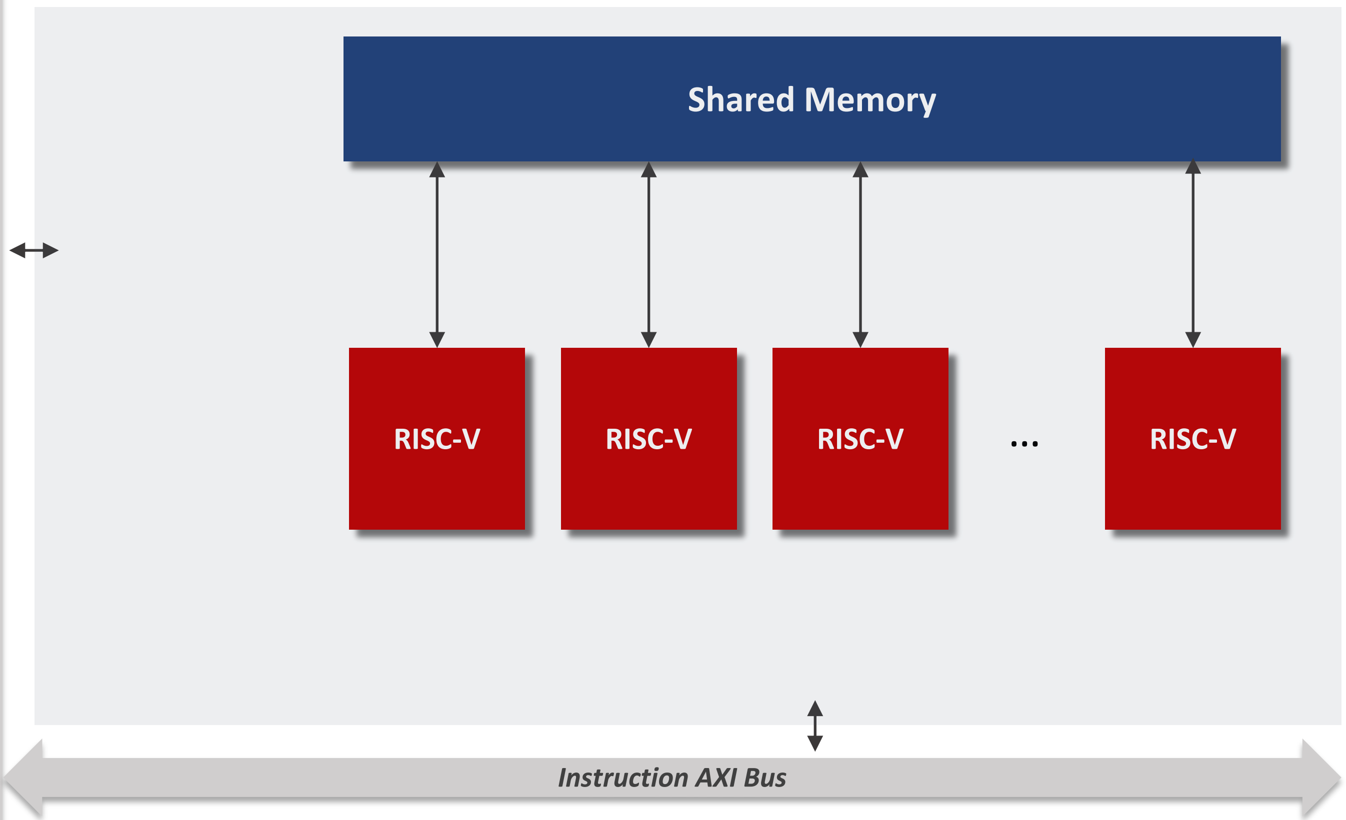
PULP and GAP8



GAP8 is a commercially available SoC based on PULP, a platform composed of a RISC-V core (RI5CY) and a Cluster to implement a **Shared-Memory** parallel programming model.

PULP and GAP8

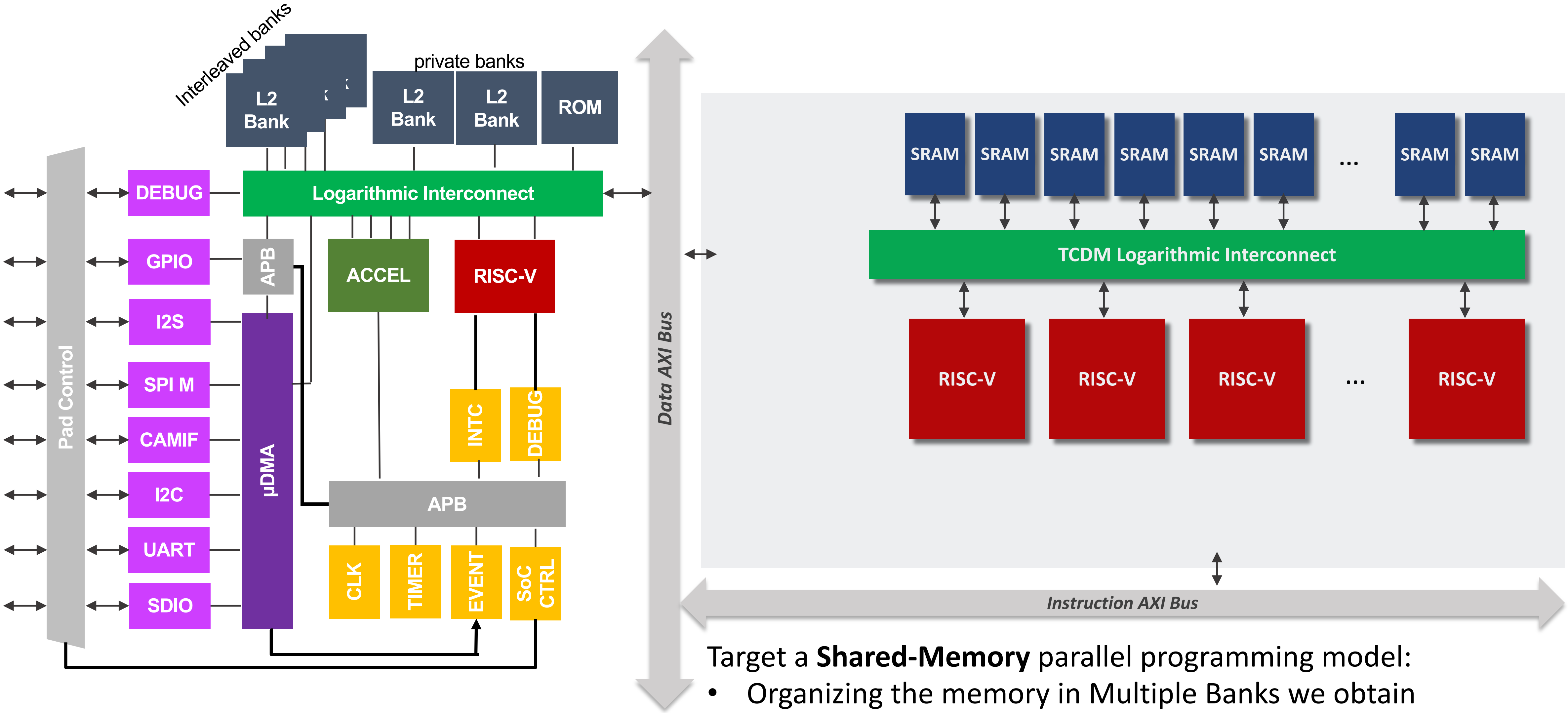




Target a **Shared-Memory** parallel programming model:

- From 4 to 16 additional cores sharing directly a **Shared Memory** (a *Tightly Coupled Data Mem* (*TCDM*) *or L1*)

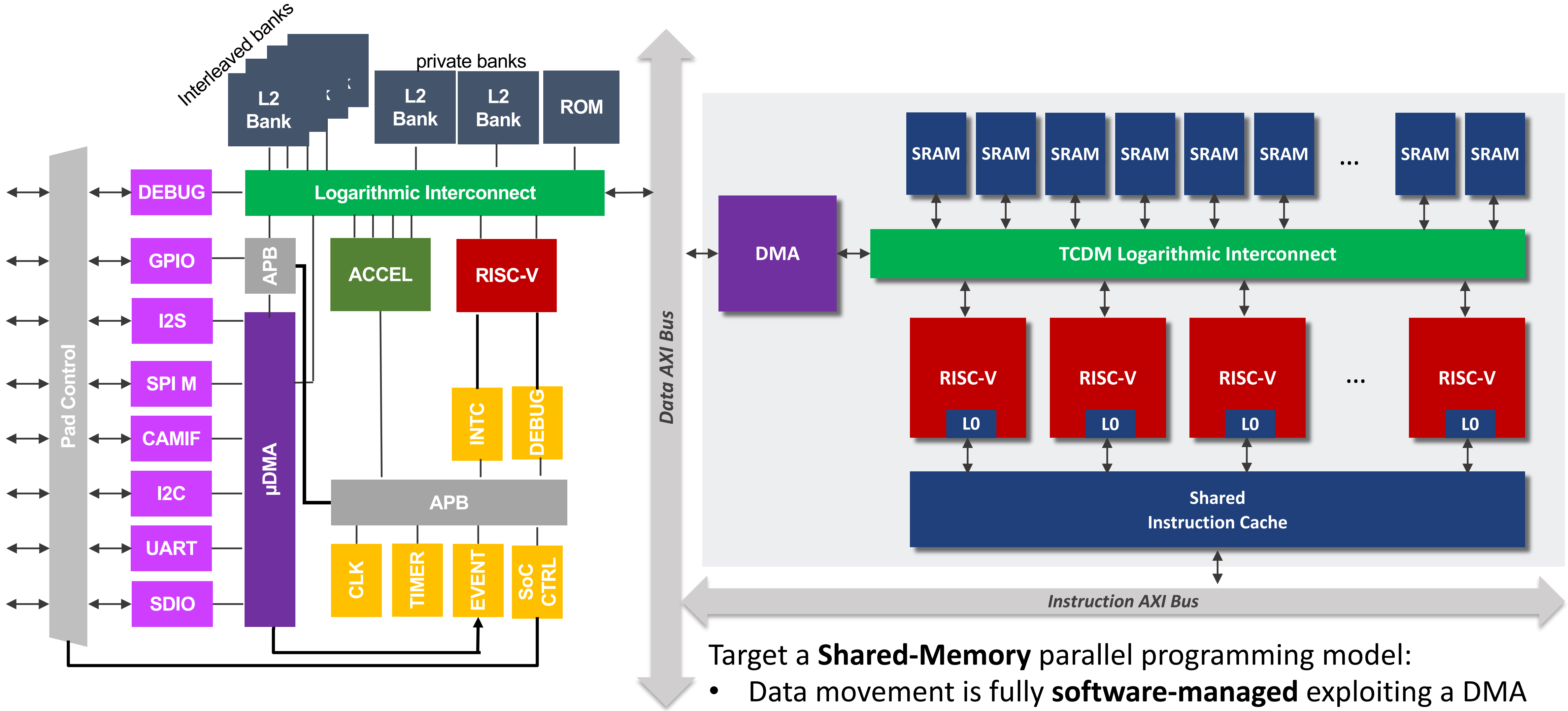
PULP and GAP8



Target a **Shared-Memory** parallel programming model:

- Organizing the memory in Multiple Banks we obtain concurrent access

PULP and GAP8

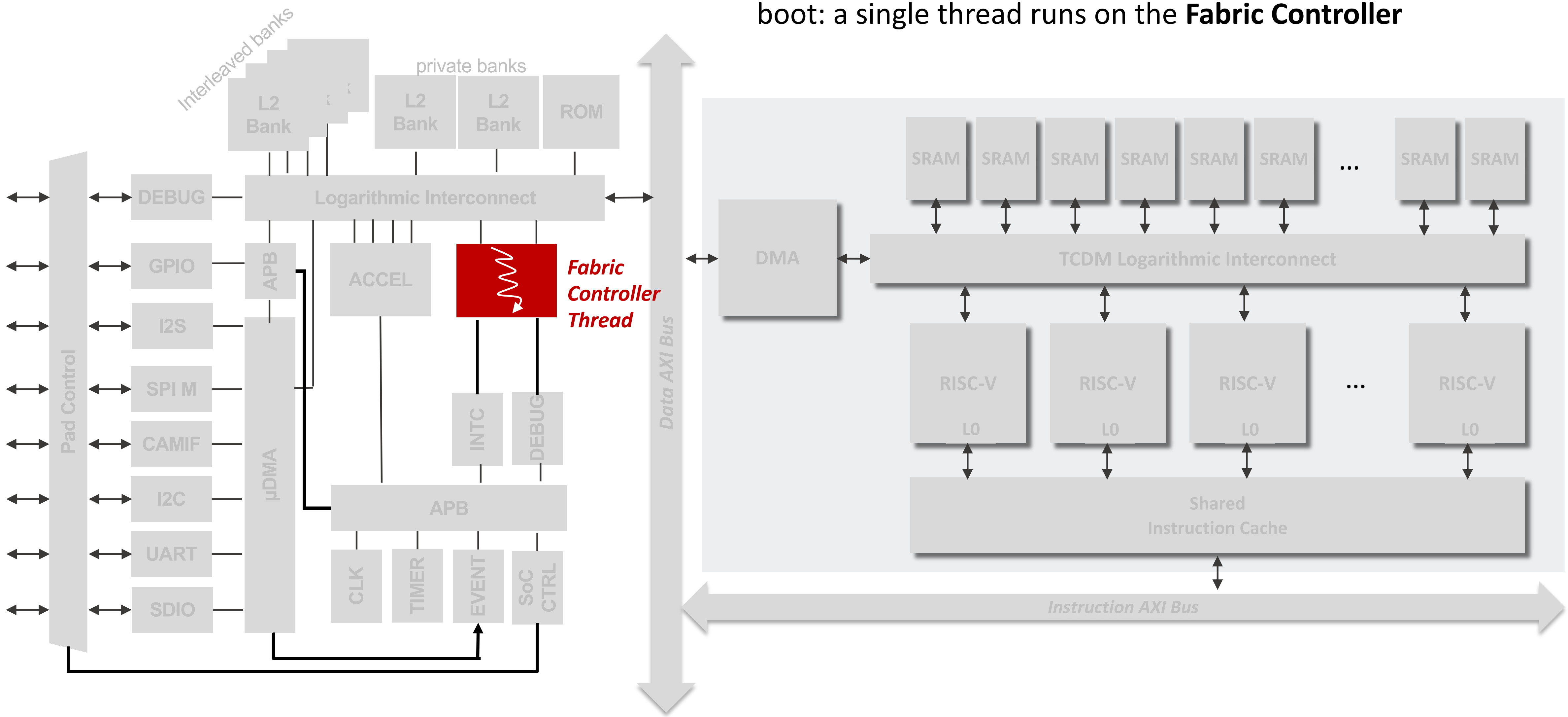


Target a **Shared-Memory** parallel programming model:

- Data movement is fully **software-managed** exploiting a DMA

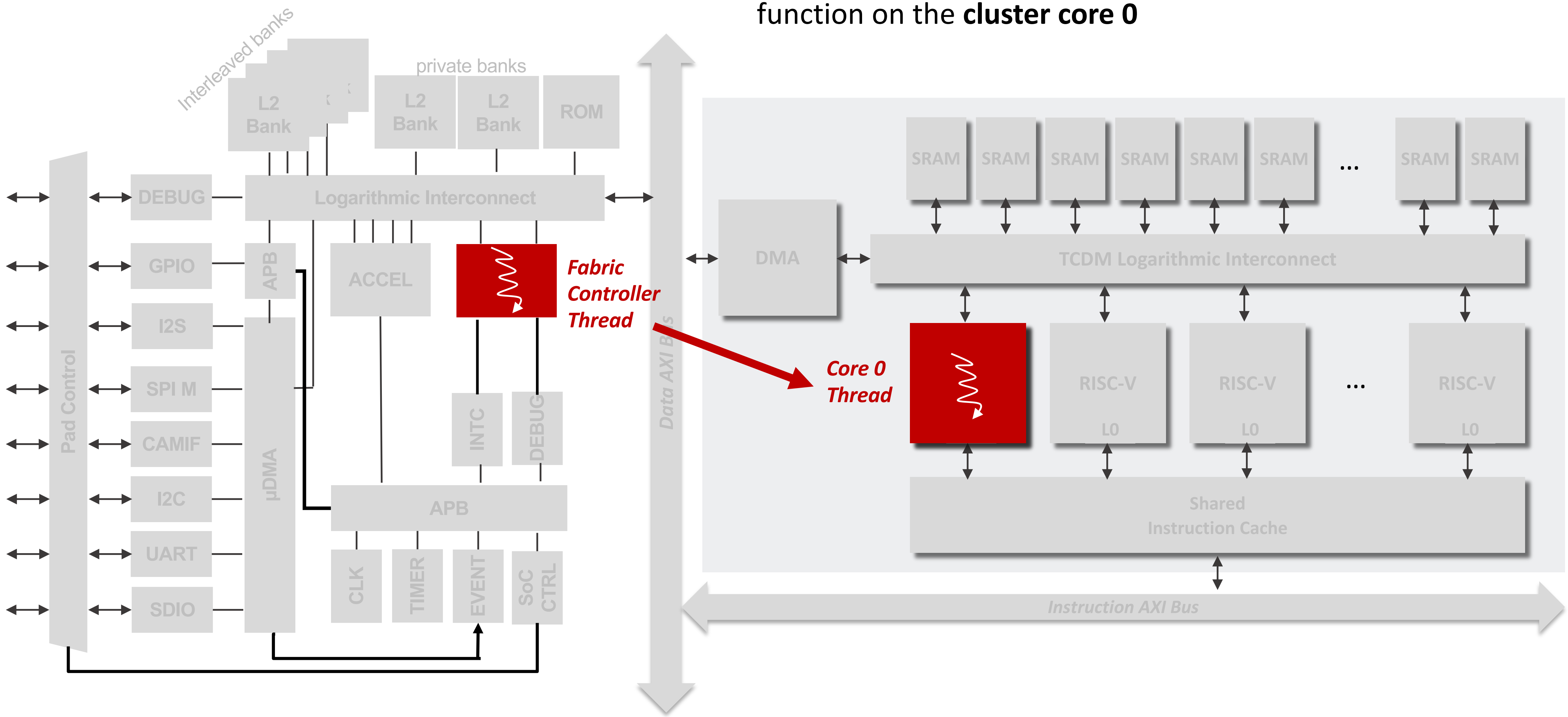
PULP and GAP8

By default the cluster is **inactive** and **clock-gated** at boot: a single thread runs on the **Fabric Controller**



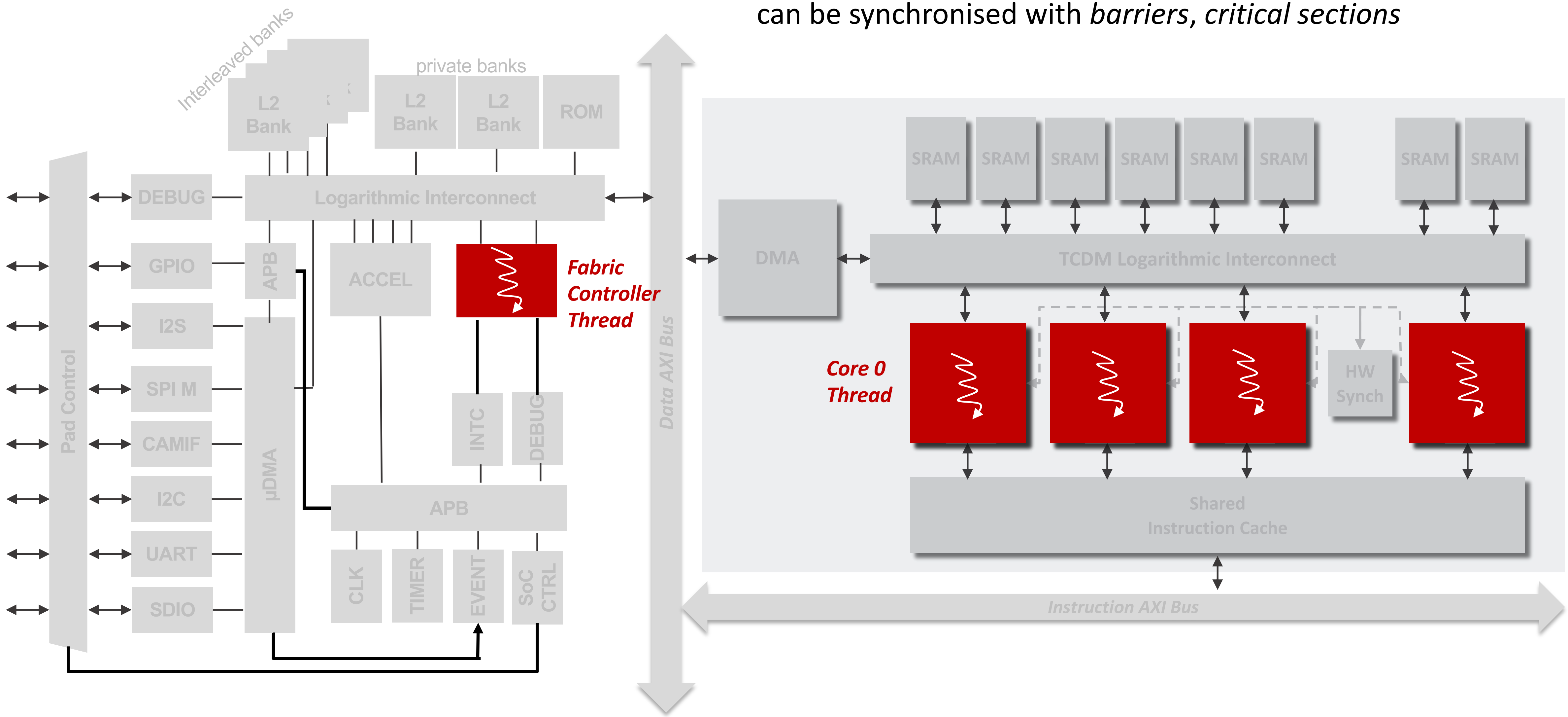
PULP and GAP8

Step1: **Mount** the cluster (enabling the clock) and **call** a function on the **cluster core 0**



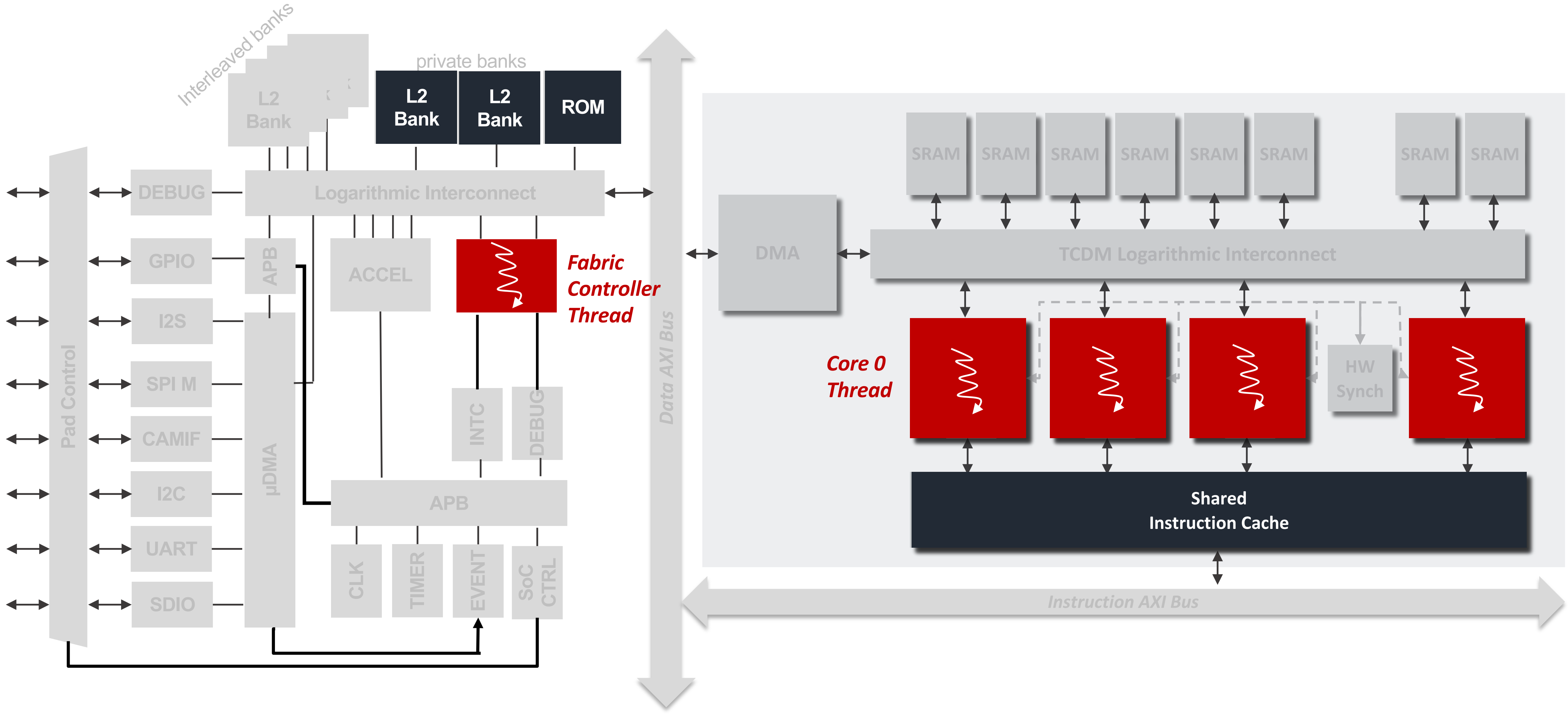
PULP and GAP8

Step2: **Fork** an **execution team** on multiple cluster cores; they can be synchronised with *barriers*, *critical sections*



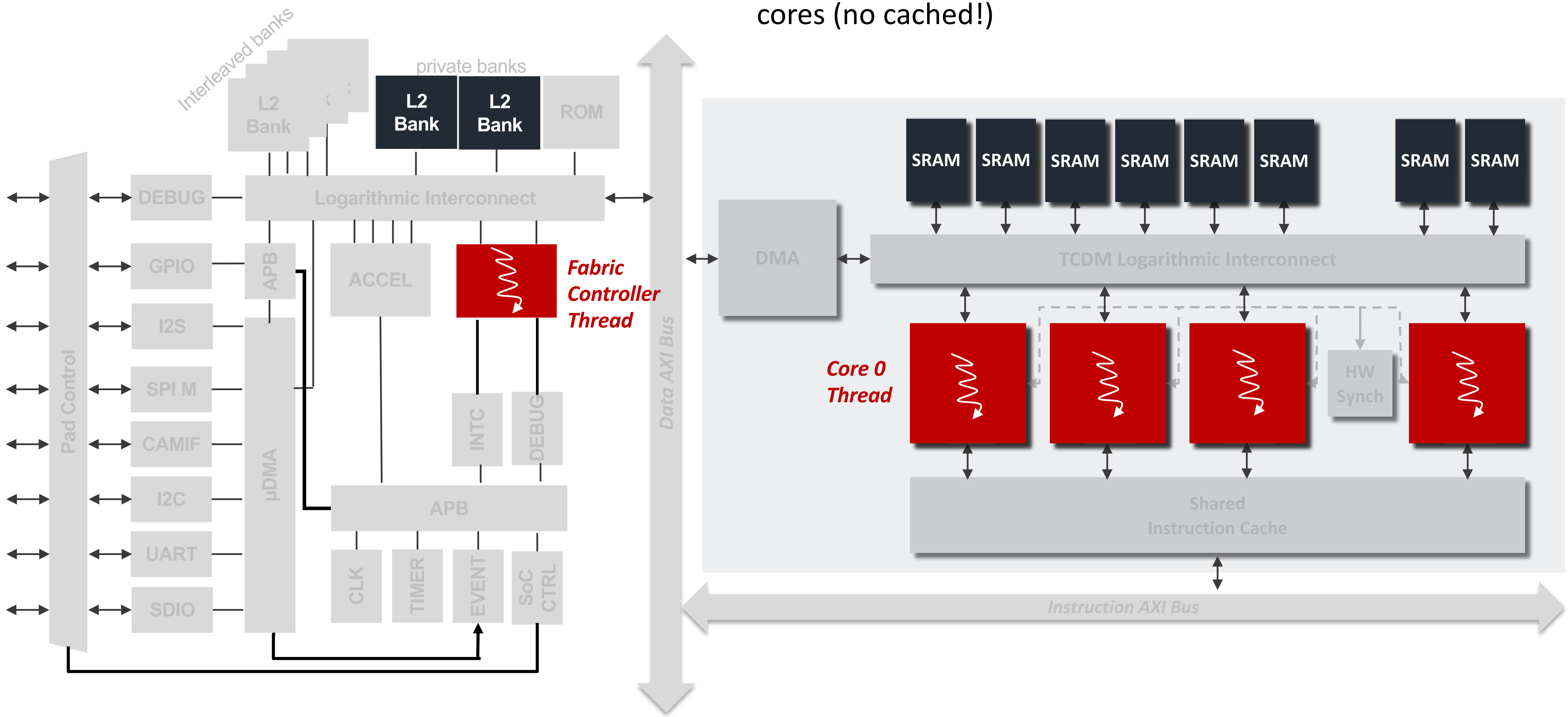
PULP and GAP8

All code is in **L2 memory** (cached for cluster threads)



PULP and GAP8

Stacks are in L2 for the FC and in **L1 TCDM** for the cluster cores (no cached!)



Outilne

- Introduction
- **Background**
  - Hardware: PULP and GAP8
  - **Language: RUST and RUST for embedded devices**
  - Software: Stream ciphers and Chacha20
- Method
- Results

RUST

Rust is a programming language that was designed with security in mind.
Several RUST features help to make it more difficult for developers to introduce vulnerabilities into their code:

- **Ownership** and **Borrow**: Rust’s strict ownership model and the borrow checker prevents common programming errors such as buffer overflows and use-after-free issues.
- **Type safety**: Rust’s type system helps prevent type confusion bugs, a common source of security vulnerabilities.
- **Concurrency safety**: Rust's approach to concurrency is designed to prevent data race conditions, which can lead to security vulnerabilities.
- **Error handling**: Rust has a built-in approach to error handling, avoiding undefined behaviours.

RUST

Rust is a programming language that was designed with security in mind.
Several RUST features help to make it more difficult for developers to introduce vulnerabilities into their code:

- **Ownership** and **Borrow**: Rust's strict ownership model and the borrow checker prevents common programming errors such as buffer overflows and use-after-free issues.
- **Type safety**: Rust's type system helps prevent type confusion bugs, a common source of security vulnerabilities.
- **Concurrency safety**: Rust's approach to concurrency is designed to prevent data race conditions, which can lead to security vulnerabilities.
- **Error handling**: Rust has a built-in approach to error handling, avoiding undefined behaviours.

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

→

```
error[E0382]: borrow of moved value: `s1`
```

```
fn main() {
  let s = String::from("hello");
  change(&s);
}
fn change(some_string: &String) {
  some_string.push_str(", world");
}
```

→

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
```

RUST

Rust is a programming language that was designed with security in mind.
Several RUST features help to make it more difficult for developers to introduce vulnerabilities into their code:

- **Ownership** and **Borrow**: Rust’s strict ownership model and the borrow checker prevents common programming errors such as buffer overflows and use-after-free issues.
- **Type safety**: Rust’s type system helps prevent type confusion bugs, a common source of security vulnerabilities.
- **Concurrency safety**: Rust's approach to concurrency is designed to prevent data race conditions, which can lead to security vulnerabilities.
- **Error handling**: Rust has a built-in approach to error handling, avoiding undefined behaviours.

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;
println!("{}, {}", r1, r2);
```

```
error[E0499]: cannot borrow `s` as
mutable more than once at a time
```

```
fn main() {
  let reference_to_nothing = dangle();
}
fn dangle() -> &String {
  let s = String::from("hello");
  &s
}
```

```
error[E0106]: missing lifetime specifier
```

RUST

Rust is a programming language with a robust macro system, a true metalanguage able to modify at compile time portion of the language itself.

It exposes two macro-categories:

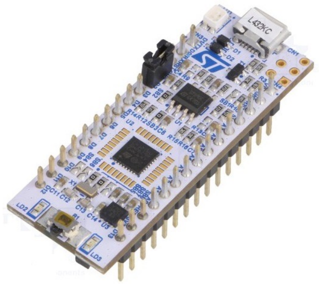
Declarative macro

- Using a matching system based on RUST tokens can emit RUST code.

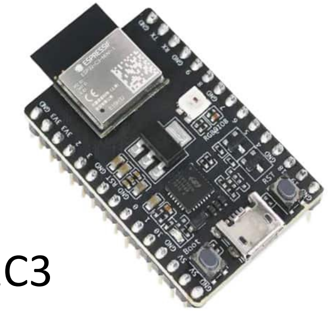
Procedural macro

- Take in input a RUST TokenStream and can modify it, emitting RUST code.
- Function
  - `#[proc_macro]`
  - Function-like macros define macros that look like function calls.
- Attribute and Derive
  - `#[proc_macro_attribute]`
  - `#[proc_macro_derive(CustomTrait)]`
  - Complex, Derive macro works on structs and enums, Attribute macro allows to create new attributes.

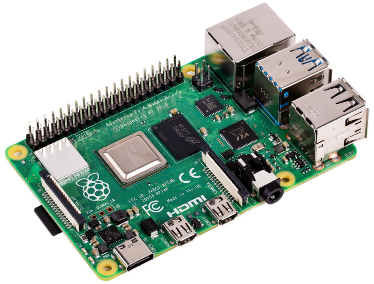
RUST for embedded



STMF0



ESP32C3



RPi4

Bare Metal Environments

Hosted Environment

«Unfortunately, hardware is basically nothing but a mutable global state, which can feel very frightening for a Rust developer. Hardware exists independently from the structures of the code we write and can be modified at any time by the real world.» **RUST Embedded Book**

RUST for embedded

Bare Metal Environments → Hosted Environment

«Unfortunately, hardware is basically nothing but a mutable global state, which can feel very frightening for a Rust developer. Hardware exists independently from the structures of the code we write and can be modified at any time by the real world.» **RUST Embedded Book**

libcore

Using `#![no_std]`

libstd

- OS abstraction
- Main thread

RUST for embedded

Bare Metal Environments → Hosted Environment

libcore
Using `#![no_std]`

libstd
- OS abstraction
- Main thread

libcore: a platform-agnostic subset of libstd

No upstream libraries, system libraries, or libc.
No heap allocation*
No concurrency
No I/O

Supported Types:
array, bool, char, fn,
i8, i16, i32, i64, i128, isize,
u8, u16,u32,u64,u128,usize, ecc.

RUST for embedded

Bare Metal Environments → Hosted Environment

libcore
Using `#![no_std]`

libstd
- OS abstraction
- Main thread

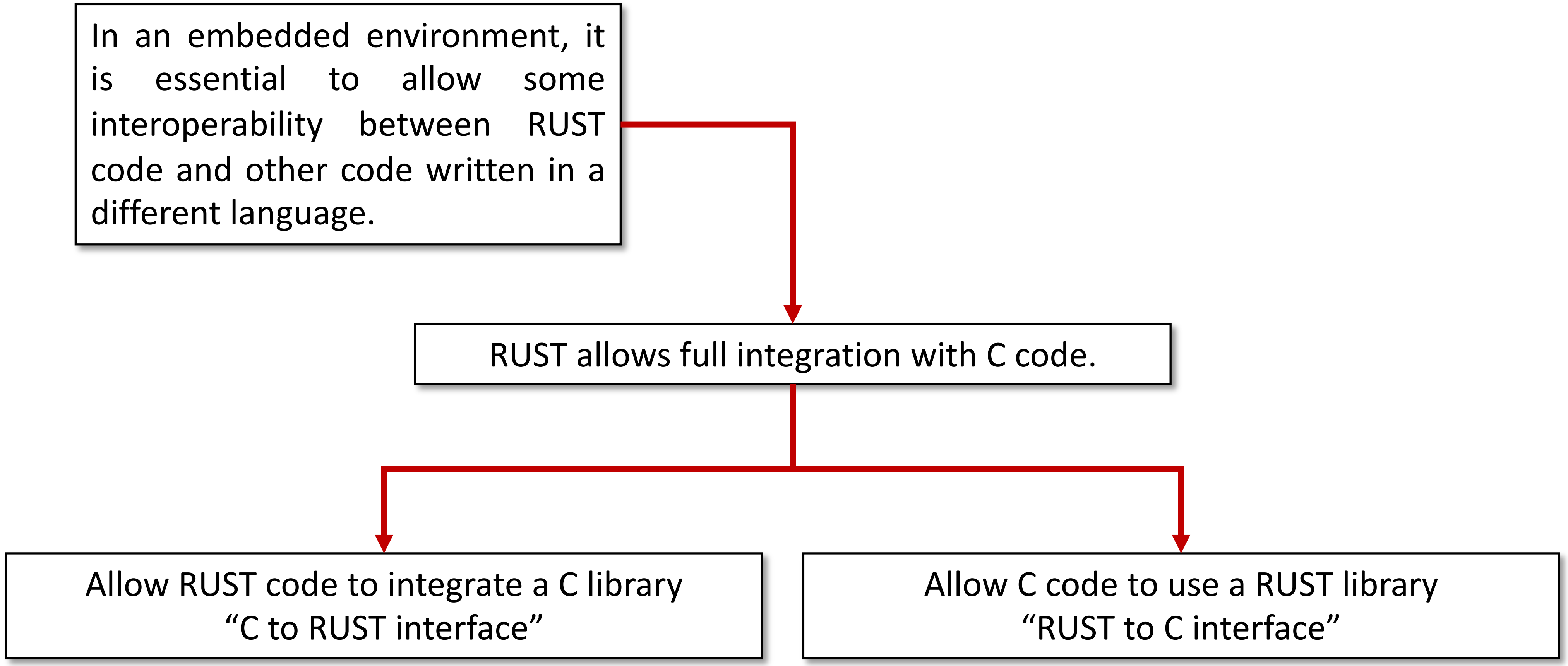
libcore: a platform-agnostic subset of libstd

No upstream libraries, system libraries, or libc.
No heap allocation*
No concurrency
No I/O

Supported Types:
array, bool, char, fn,
i8, i16, i32, i64, i128, isize,
u8, u16,u32,u64,u128,usize, ecc.

*Using the crate `core::alloc` It is possible to create a custom allocator!

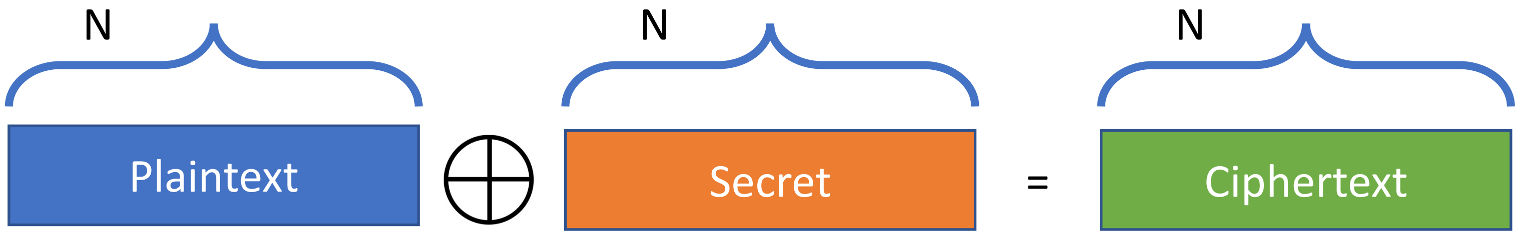
RUST for embedded



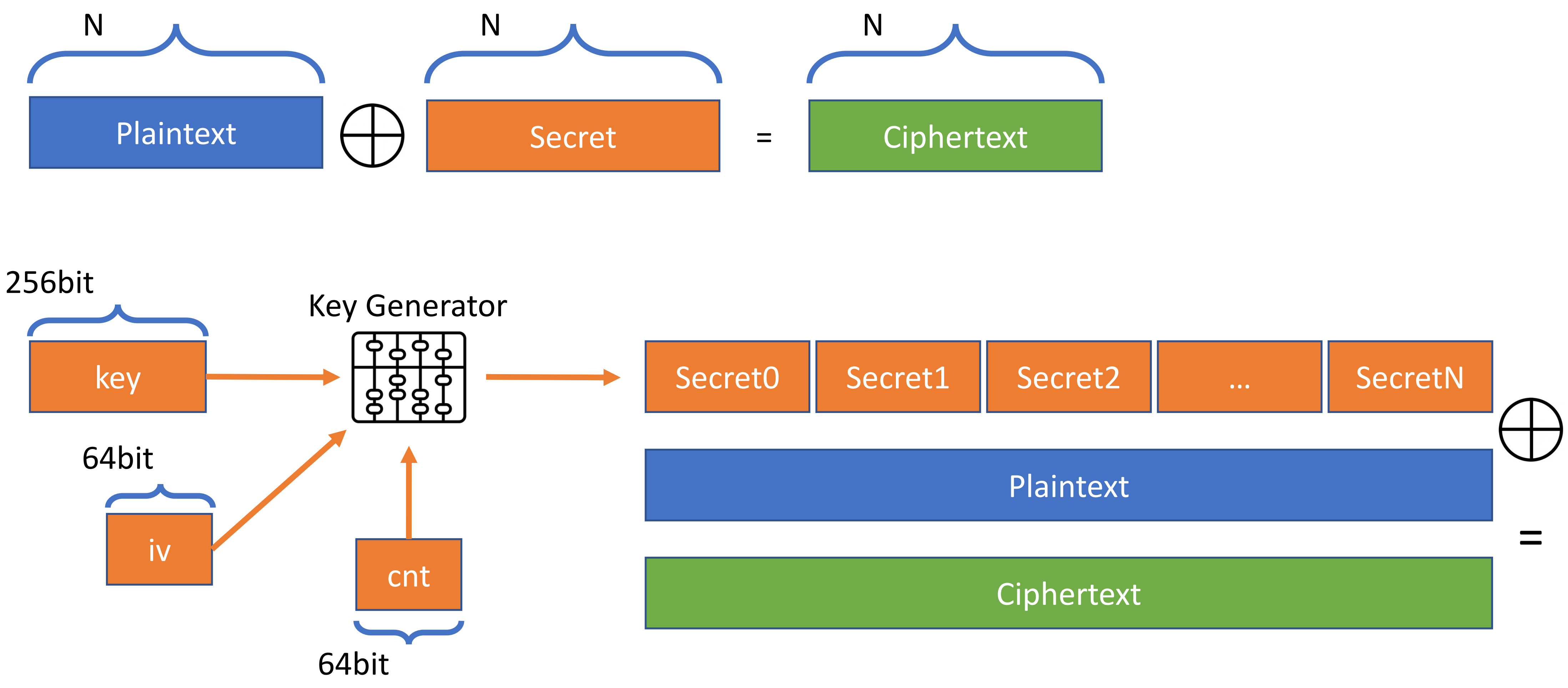
Outilne

- Introduction
- **Background**
  - Hardware: PULP and GAP8
  - Language: RUST and RUST for embedded devices
  - **Software: Stream ciphers and Chacha20**
- Method
- Results

Stream Ciphers



Stream Ciphers



Stream Ciphers – Chacha20 By Daniel J. Bernstein

X_0:

0 **0x61707865**	1 **0x3320646e**	2 **0x79622d32**	3 **0x6b206574**
4 **key**	5 **key**	6 **key**	7 **key**
8 **key**	9 **key**	10 **key**	11 **key**
12 **cnt**	13 **cnt**	14 **iv**	15 **iv**

QuarterRound

```
QR(A,B,C,D):
• A+=B; D^=A; D<<<=16;
• C+=D; B^=C; B<<<=12;
• A+=B; D^=A; D<<<=8;
• C+=D; B^=C; B<<<=7;
```

OddRound

```
• QR(0, 4,  8, 12)
• QR(1, 5,  9, 13)
• QR(2, 6, 10, 14)
• QR(3, 7, 11, 15)
```

EvenRound

```
• QR(0, 5, 10, 15)
• QR(1, 6, 11, 12)
• QR(2, 7,  8, 13)
• QR(3, 4,  9, 14)
```

A double round (X_i->X_{i+1}) is a subsequent execution of an OddRound and an EvenRound.

Chacha20 execute ten double rounds to complete a block, generating $S = X_0+X_{10}$ (64 Byte)

Matrix cells of 32bit little-endian

Stream Ciphers – Chacha20

By IETF - rfc7539,rfc8439

X_0:

0 **0x61707865**	1 **0x3320646e**	2 **0x79622d32**	3 **0x6b206574**
4 **key**	5 **key**	6 **key**	7 **key**
8 **key**	9 **key**	10 **key**	11 **key**
12 **cnt**	13 **iv**	14 **iv**	15 **iv**

QuarterRound

QR(A,B,C,D):

- `A+=B; D^=A; D<<<=16;`
- `C+=D; B^=C; B<<<=12;`
- `A+=B; D^=A; D<<<=8;`
- `C+=D; B^=C; B<<<=7;`

OddRound

- `QR(0, 4,  8, 12)`
- `QR(1, 5,  9, 13)`
- `QR(2, 6, 10, 14)`
- `QR(3, 7, 11, 15)`

EvenRound

- `QR(0, 5, 10, 15)`
- `QR(1, 6, 11, 12)`
- `QR(2, 7,  8, 13)`
- `QR(3, 4,  9, 14)`

A double round (X_i->X_{i+1}) is a subsequent execution of an OddRound and an EvenRound.

Chacha20 execute ten double rounds to complete a block, generating $S = X_0+X_{10}$ (64 Byte)

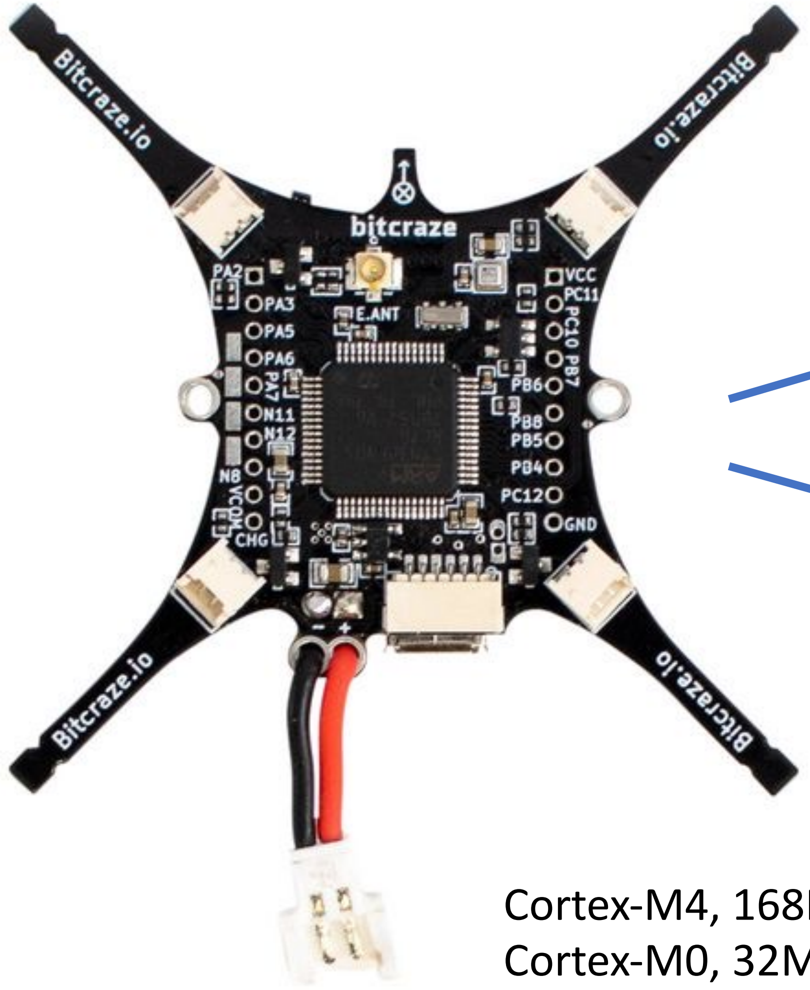
(Max 256 GiB)

Matrix cells of 32bit little-endian

Outilne

- Introduction
- Background
- **Method**
- Results

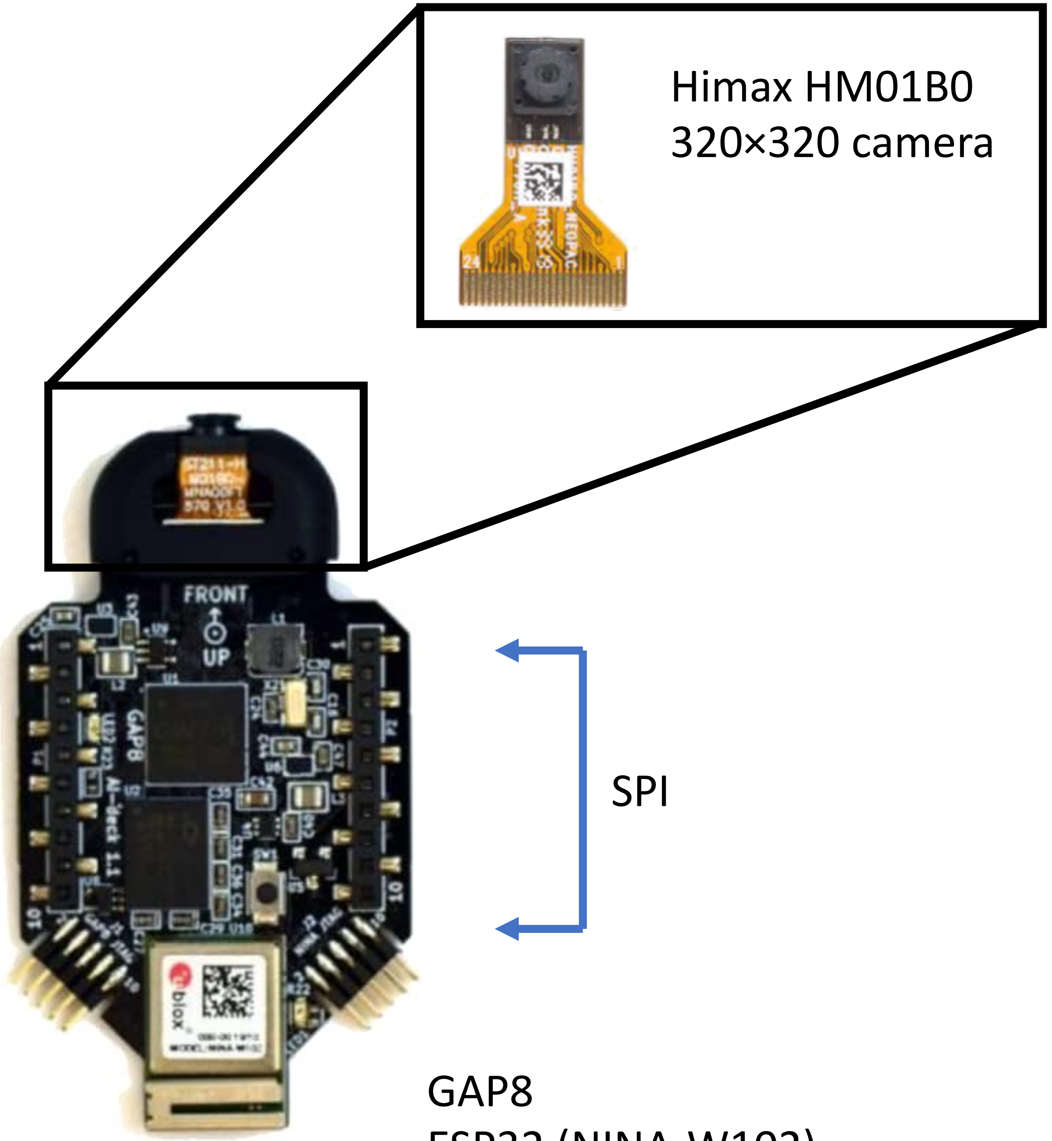
Method



UART
(STM32-GAP8)

+

UART
(STM32-ESP32)

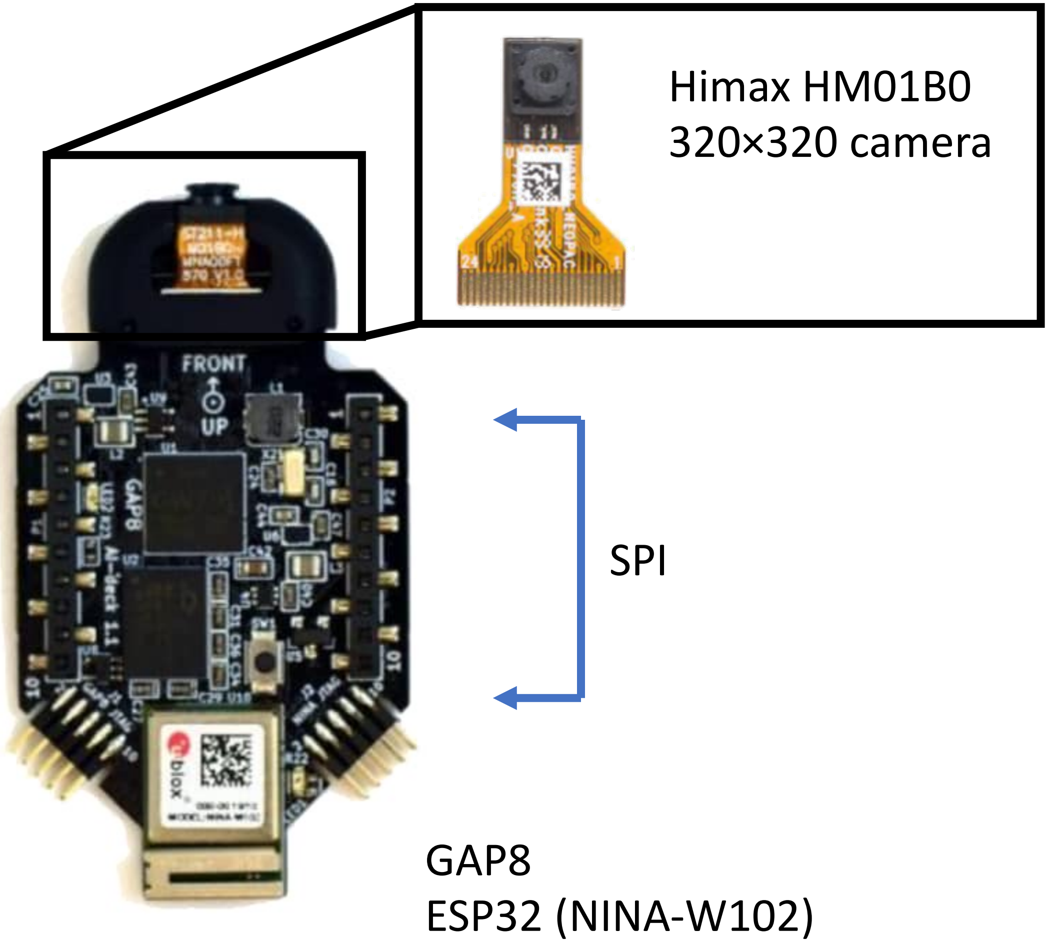


SPI

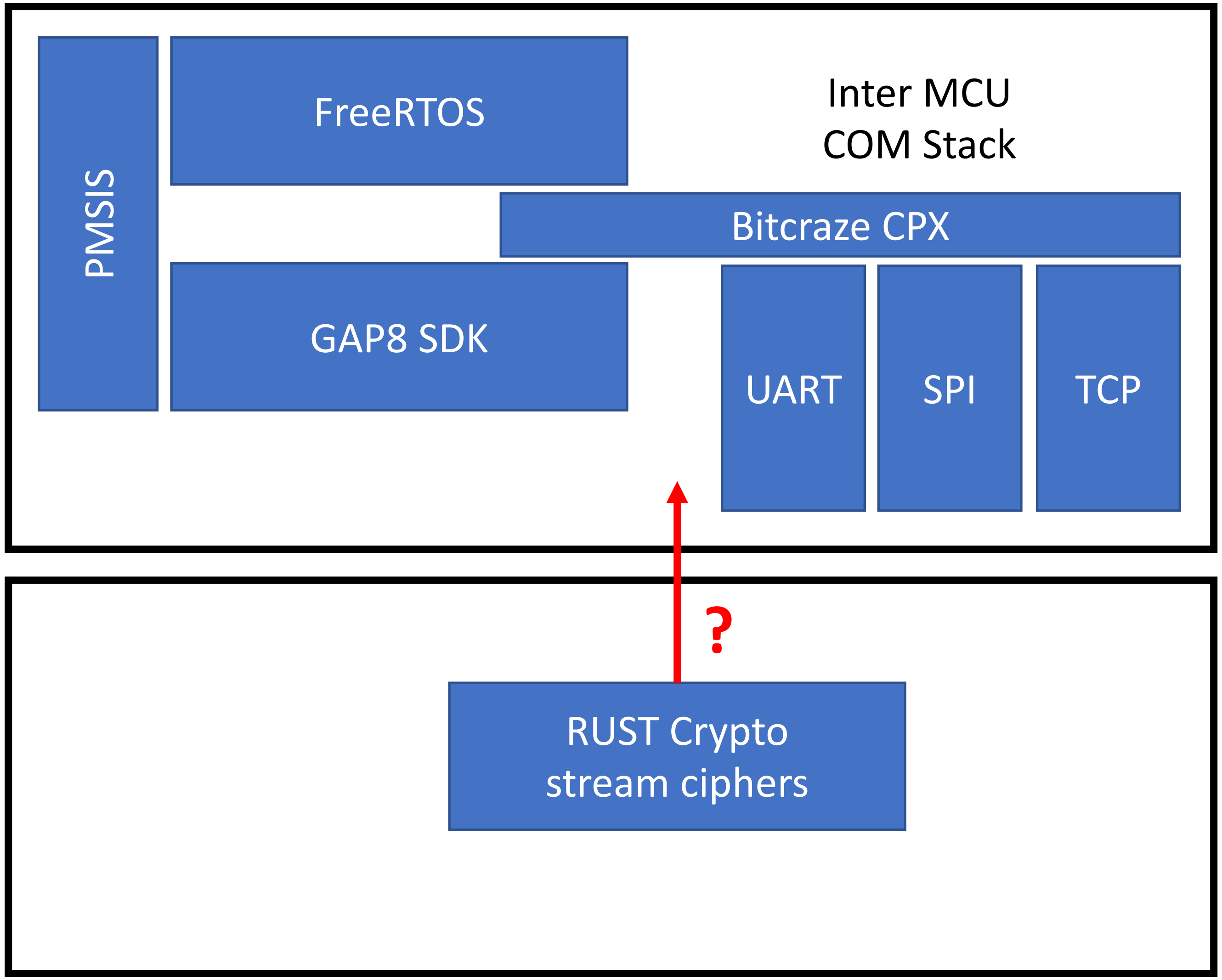
GAP8
ESP32 (NINA-W102)

Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash (STM32F405)
Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash (nRF51822)

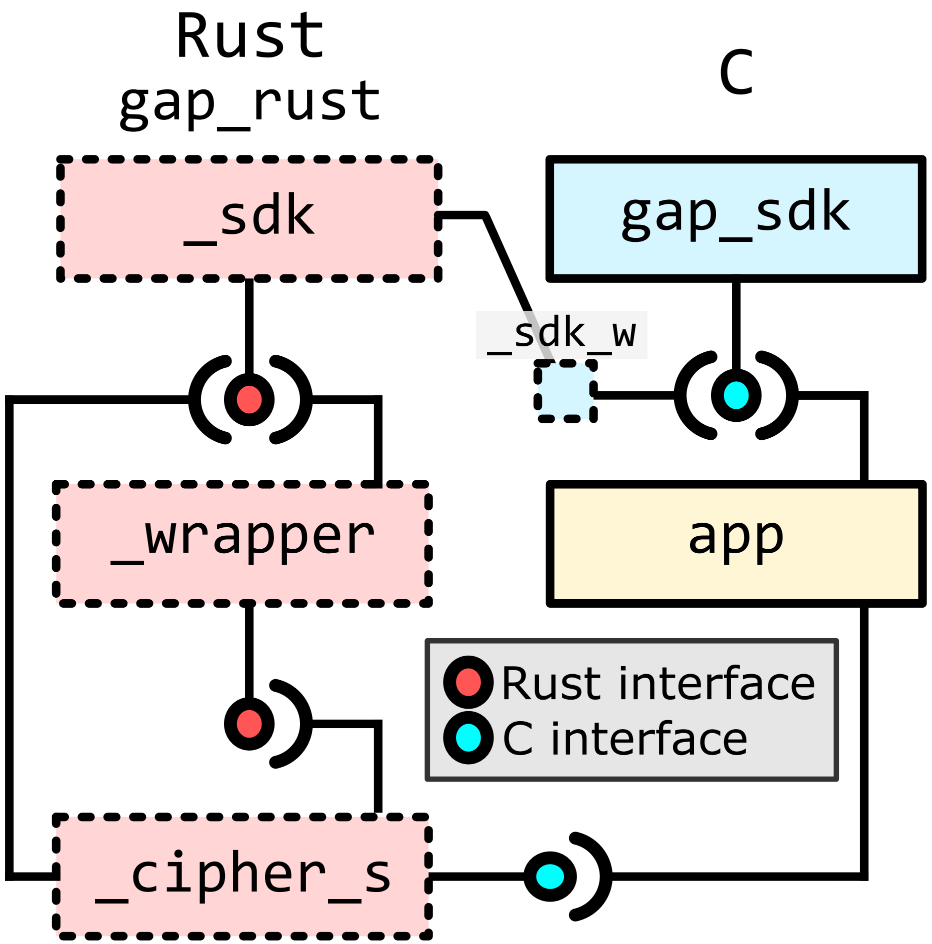
Method



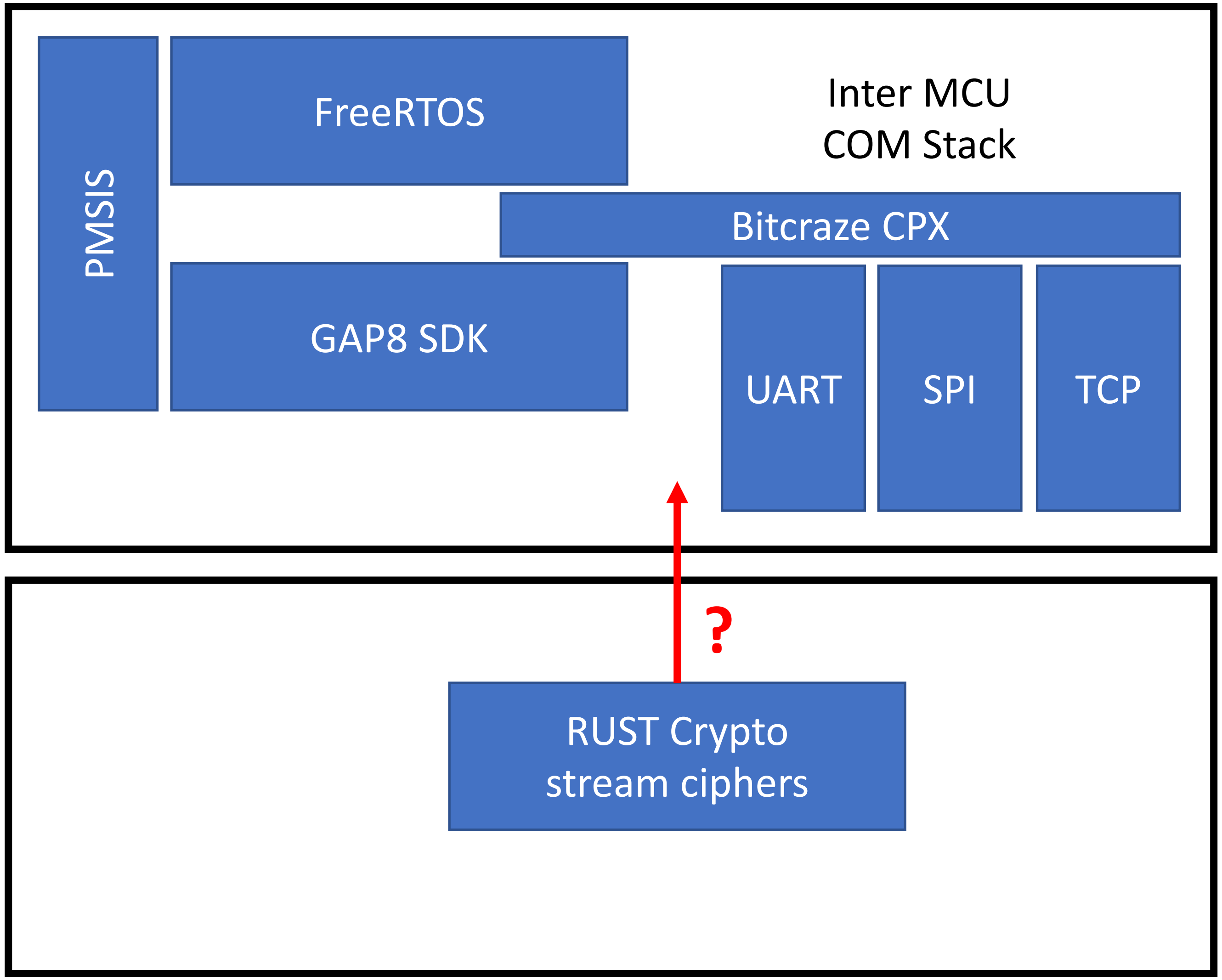
Software Stack
Pulp Microcontroller Software Interface Standard (PMSIS)
+
AiDeck Board Support Package (BSP)



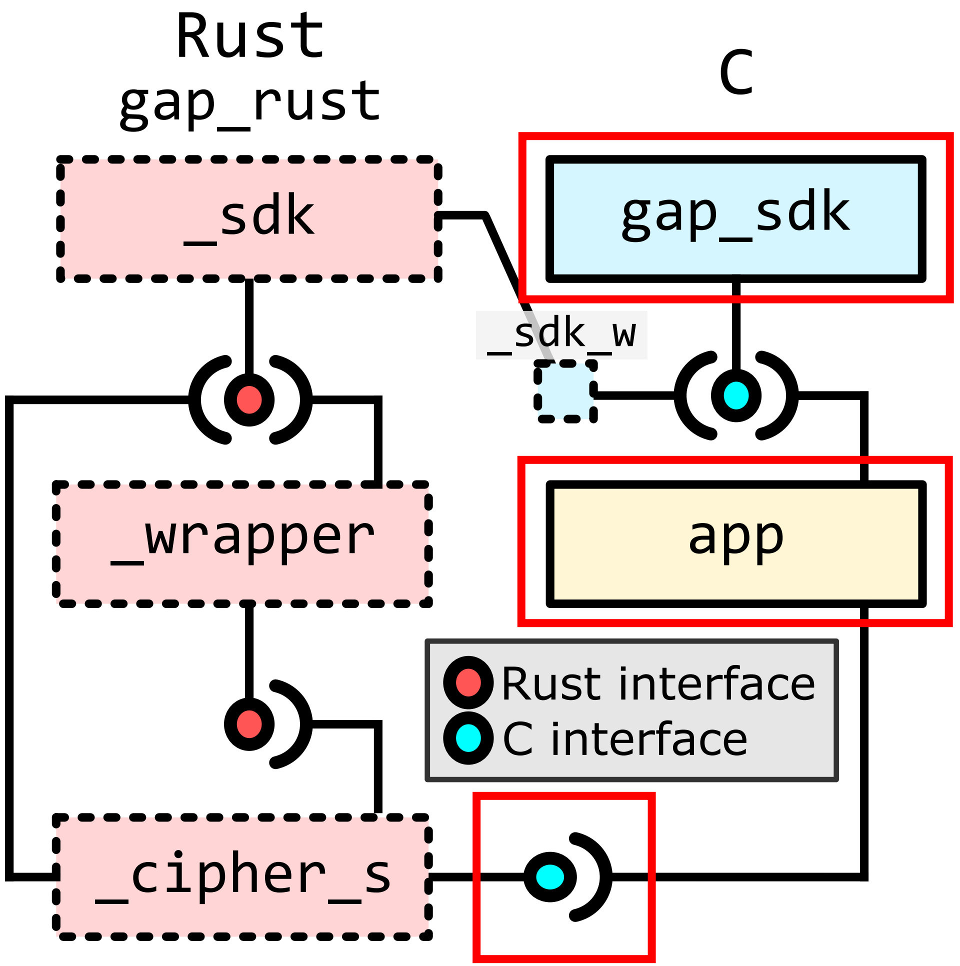
Method



Software Stack
Pulp Microcontroller Software Interface Standard (PMSIS)
+
AiDeck Board Support Package (BSP)

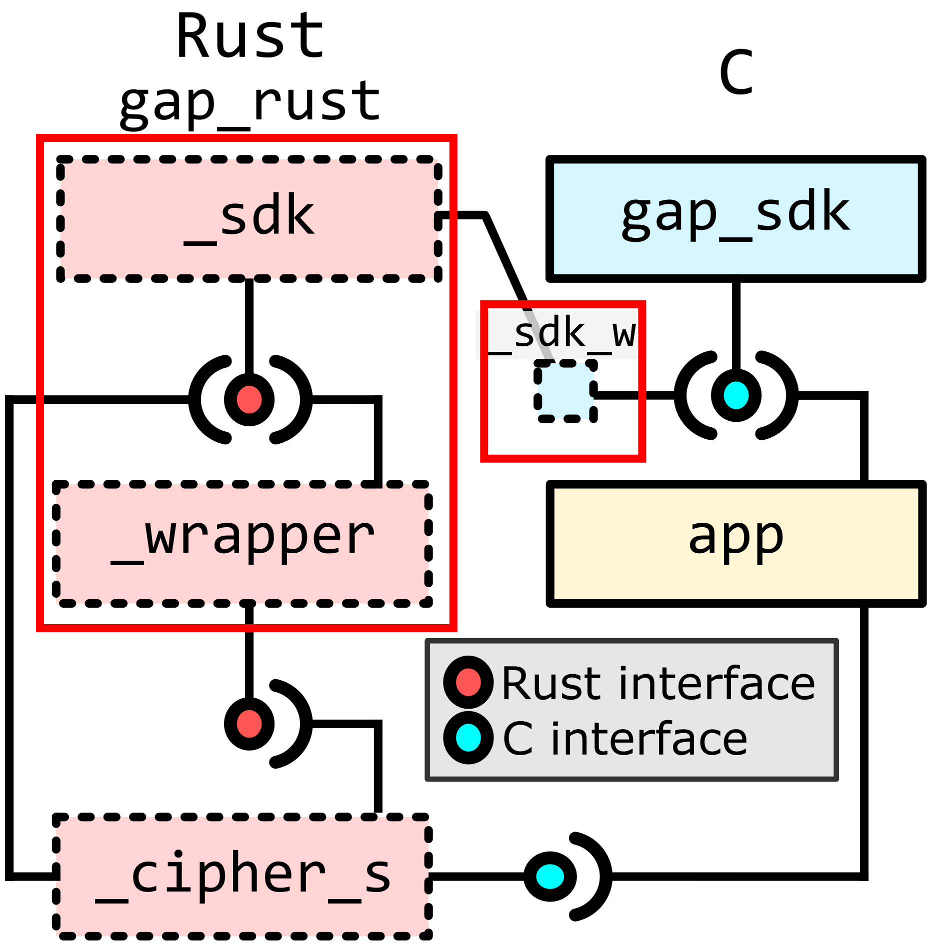


Method



A GAP SDK-based application (written in C) that wants to use the RUST streaming cipher library can import a “RUST to C library” that exposes the library entry point to the application.

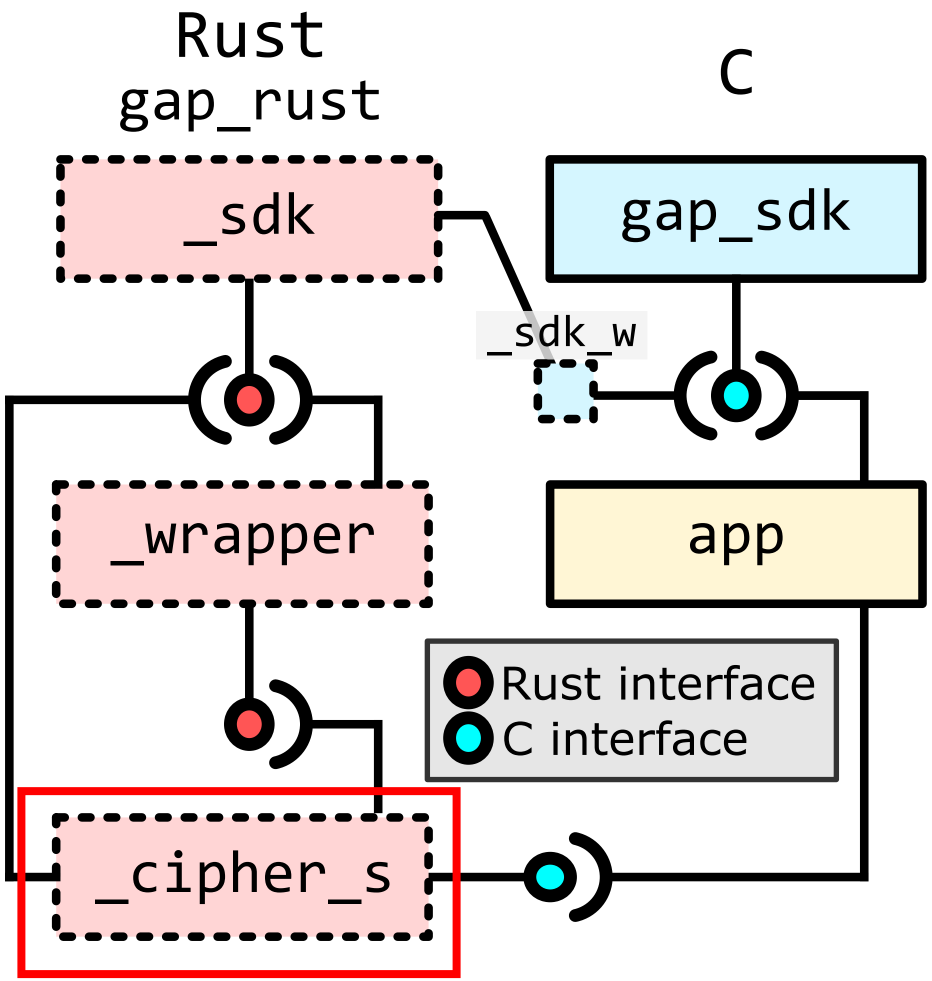
Method



A GAP SDK-based application (written in C) that wants to use the RUST streaming cipher library can import a "RUST to C library" that exposes the library entry point to the application.

In this scenario, to adapt the RUST library to the embedded device, we need to use some GAP SDK features and integrate the GAP SDK as a C to the RUST library.

Method

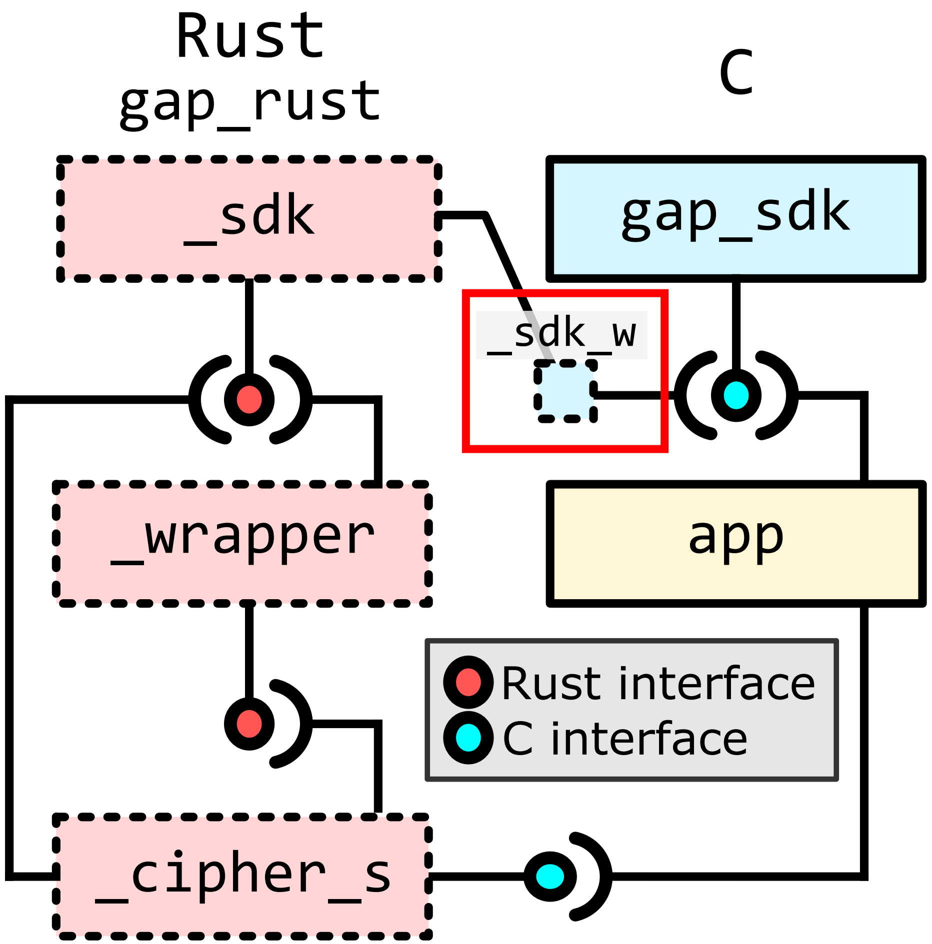


A GAP SDK-based application (written in C) that wants to use the RUST streaming cipher library can import a “RUST to C library” that exposes the library entry point to the application.

In this scenario, to adapt the RUST library to the embedded device, we need to use some GAP SDK features and integrate the GAP SDK as a C to the RUST library.

To obtain better performance by exploiting the PULP features, we need to provide an encapsulated library version able to use the previous RUST libraries.

Method



Problem:

Some gap_sdk functions cannot be integrated into RUST directly; they are declared static in the header file, limiting their visibility.

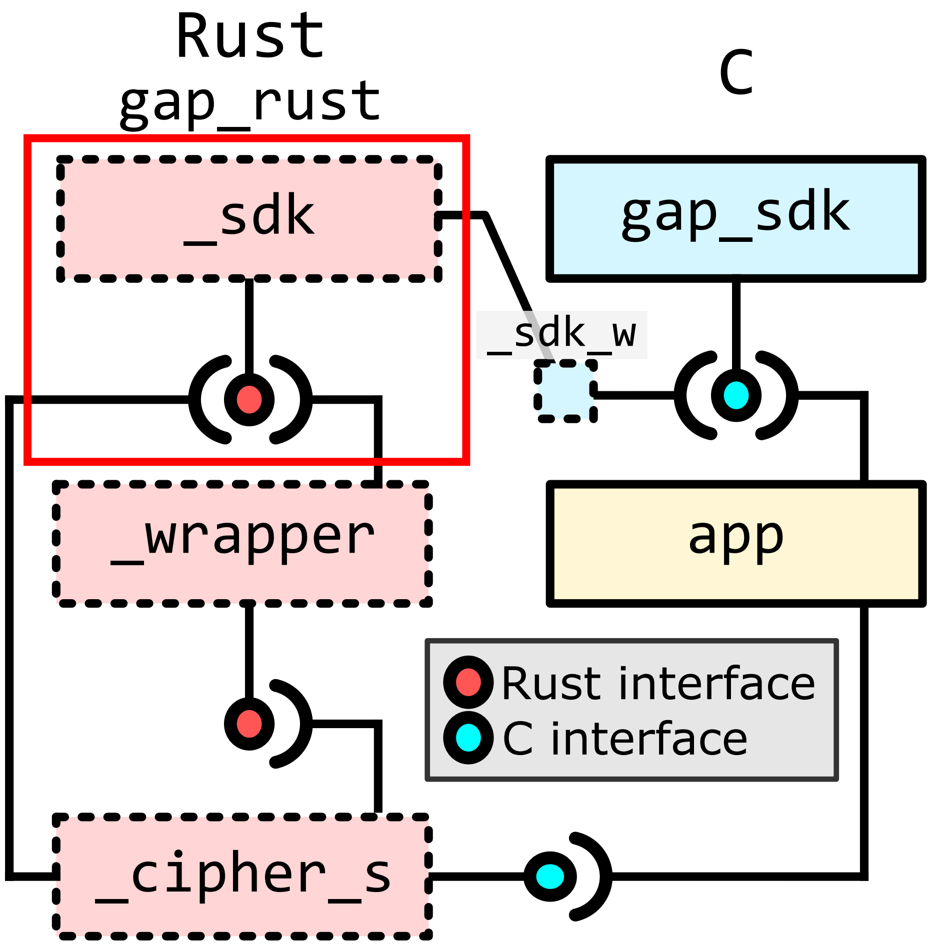
Solution: developed a C library (gap_rust_sdk_w) able to directly expose the functions.

Example:

```
#include <pmsis.h>
#include <bsp/bsp.h>

// Allocate memory in L2
void *pmsis_l2_malloc_wrap(uint32_t size)
{
  return pmsis_l2_malloc(size);
}
```

Method



Problem:
Wrap the gap_sdk functions and structures.

Solution:
Wrap extern function declaration code in:

```
extern "C" { … }
```

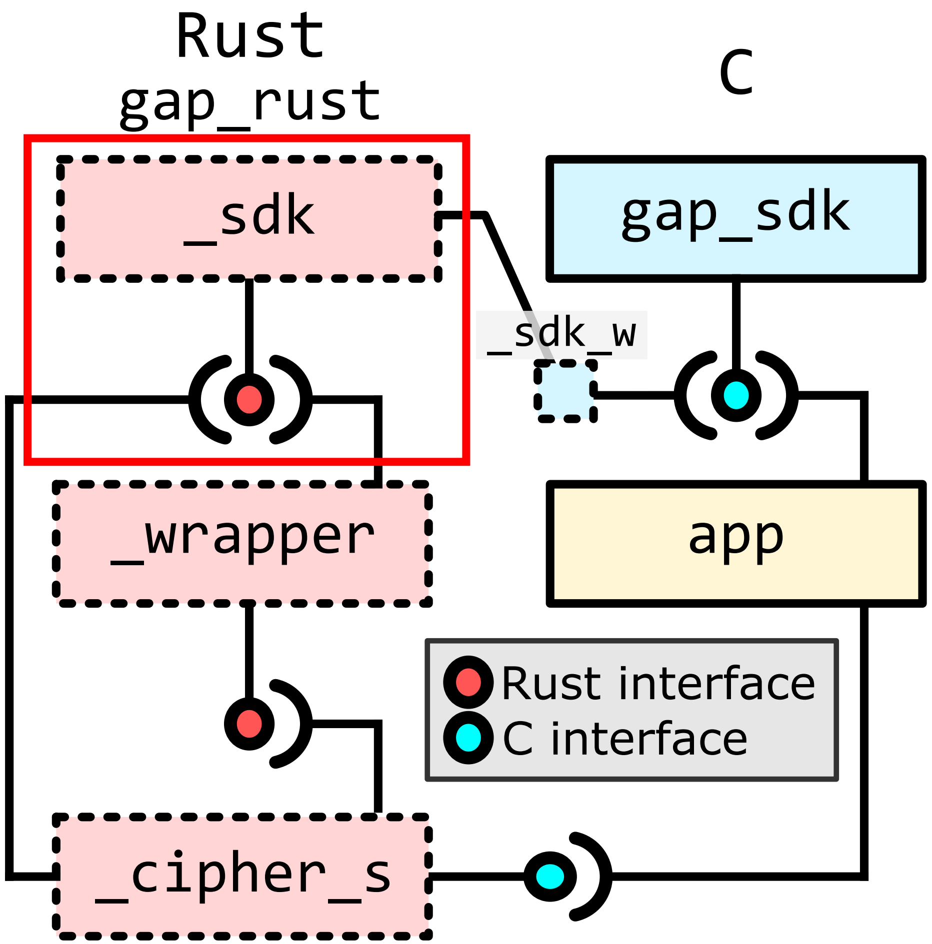
Use cty crate for types:

```
pub fn pmsis_l2_malloc_wrap(
 size: cty::uint32_t) -> *mut cty::c_void;
```

Rewrite C structures in RUST using the macro #[repr(C)] and types defined in cty crate:

```
#[repr(C)]
pub struct PiClusterConf {
  device_type: PiDeviceType,
  id: cty::c_int,
  heap_start: *mut cty::c_void,
  heap_size: u32,
  event_kernel: *mut PmsisEventKernelWrap,
  flags: PiClusterFlags,
}
```

Method



Problem:
Avoid compilation for wrong architectures.

Solution:
```
#[cfg(not(target_arch="riscv32"))]
compile_error!("unsupported target");
```

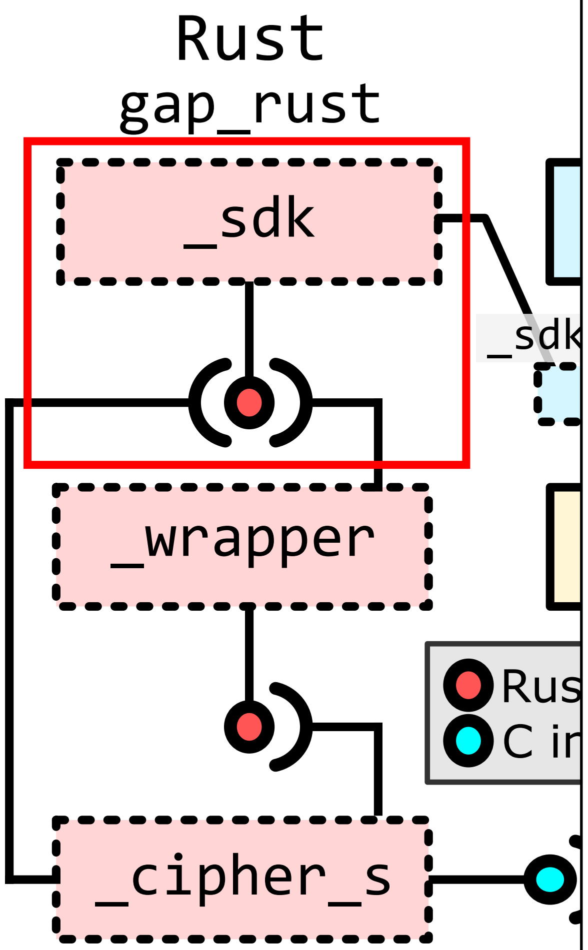
Problem:
Disable libstd in order to use only libcore (RUST embedded).

Solution:
```
#![no_std]
```

Problem:
GAP8 use different allocators but using libcore we completely lack the memory allocation features.

Solution:
```
#![feature(allocator_api)]
```
Now we can pass in an instance of an AllocRef to each collection for which we want a custom allocator.

Method



Problem:

Create a custom allocator

Solution:

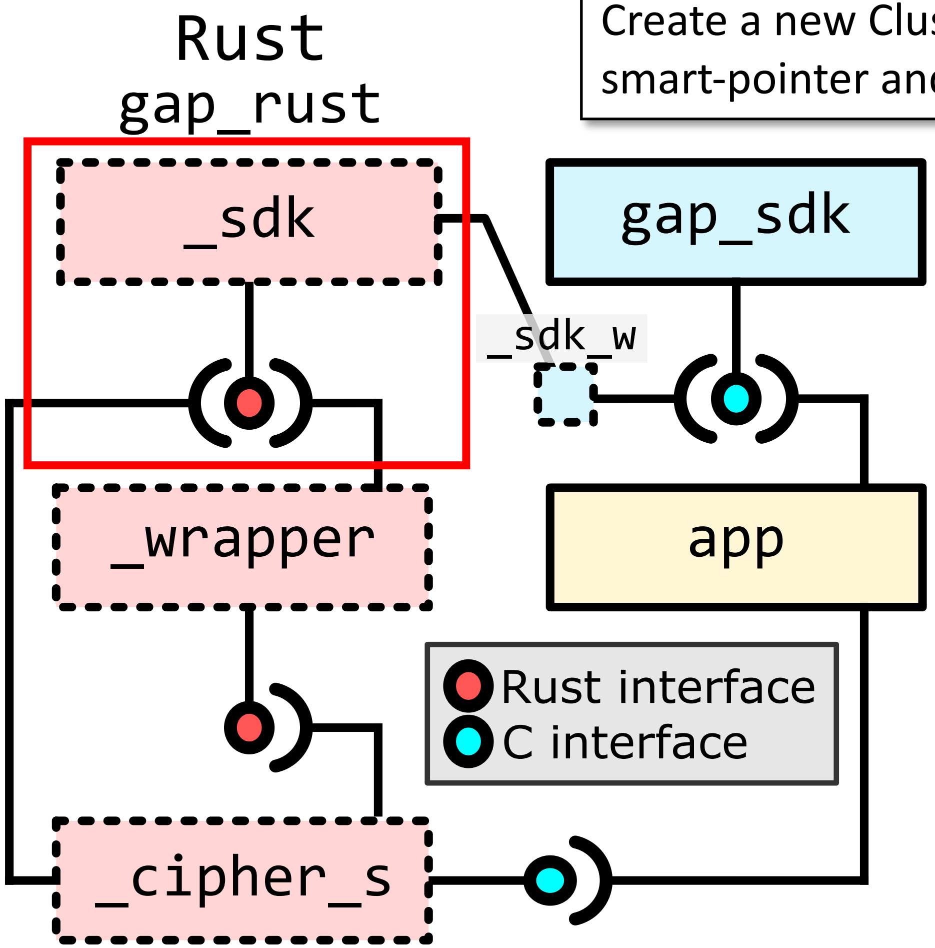
Create one using gap_sdk functions:

```
pub struct L2Allocator;
unsafe impl Allocator for L2Allocator {
  fn allocate(&self, layout: Layout) ->
   Result<NonNull<[u8]>, AllocError> {
    …
    let ptr = pmsis_l2_malloc(
      layout.size()
      .try_into()
      .map_err(|_| AllocError)?) as
      *mut u8;
    NonNull::new(ptr)
    .map(|ptr| NonNull::slice_from_raw_parts(
                                 ptr, layout.size()))
    .ok_or(AllocError)
  … } …
}
```

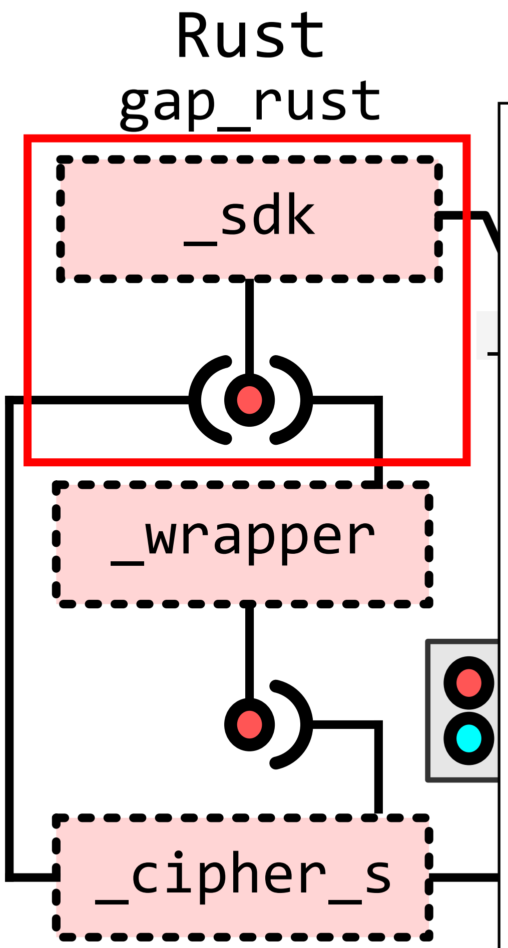
Method

Problem:
Abstracting the Cluster

Solution:
Create a new Cluster type and implement functions on it exploiting the Box RUST smart-pointer and the custom allocator.



Method



Problem:

Abstracting the Cluster

Solution:

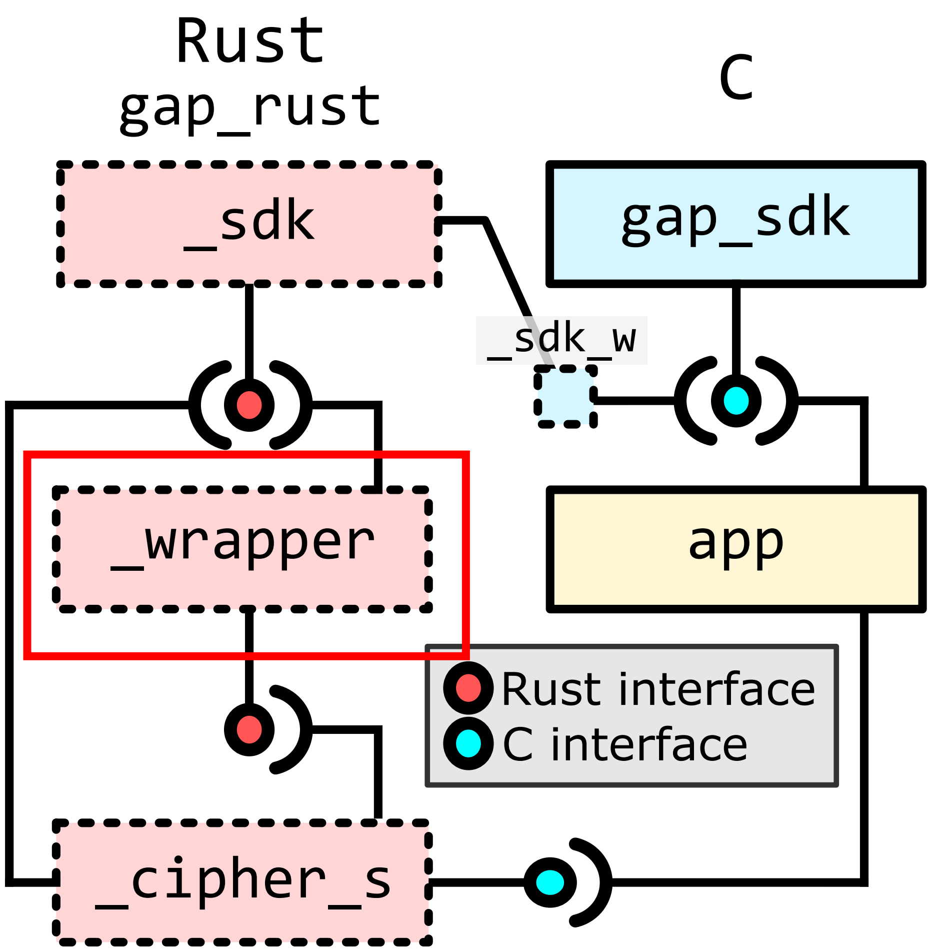
Create a new Cluster typ

smart-pointer and the c

```
pub struct Cluster<const CORES: usize> {
  device: *mut PiDevice,
  _conf: *mut PiClusterConf,
}
```

```
impl<const CORES: usize> Cluster<CORES> {
  pub fn new() -> Result<Self, ()> {
    let device: *mut _ = Box::leak(
      Box::new_in(PiDevice::uninit(), L2Allocator));
    let _conf: *mut _ = Box::leak(
      Box::new_in(PiClusterConf::uninit(), L2Allocator));
    unsafe {
      pi_cluster_conf_init(_conf);
      pi_open_from_conf(device, _conf as *mut cty::c_void);
      if pi_cluster_open(device as *mut PiDevice) != 0 {
        return Err(());
      }
      Ok(Self { device, _conf })
  }
  …
}
```

Method



Problem:
Masking the data transfer latency between L2 and Cluster L1 Memory.

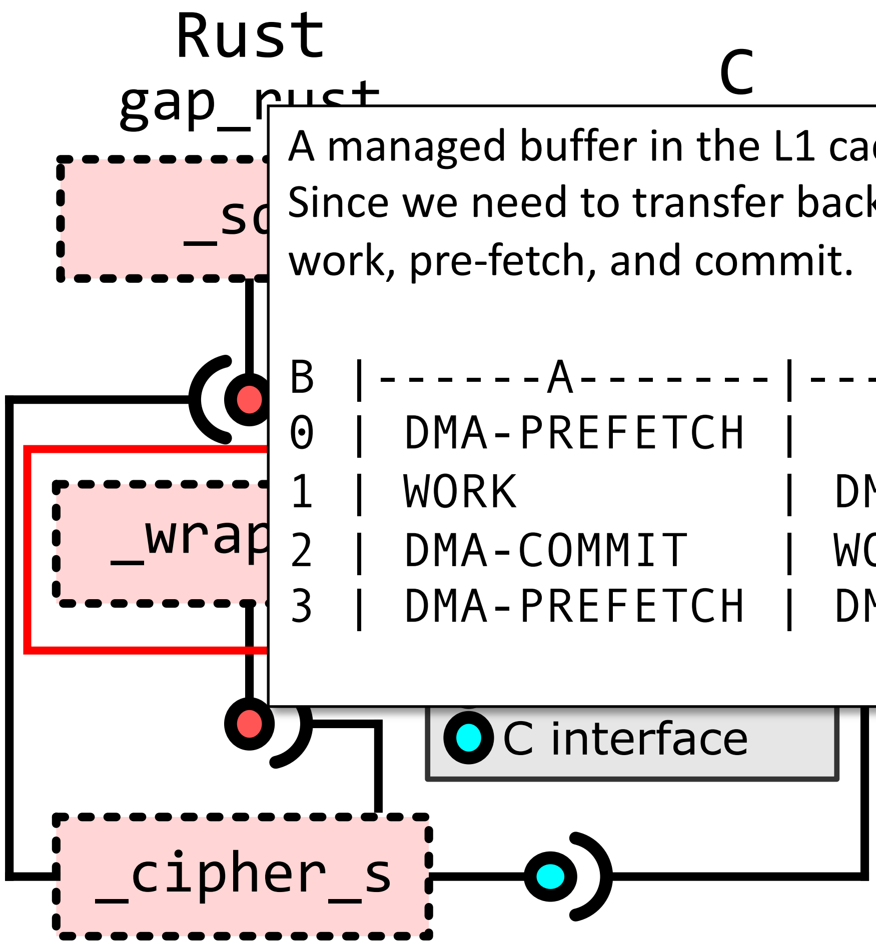
Solution:
Create a crate able to abstract a buffer and the DMA in order to move data between L2 and cluster L1.

Problem:
Wrapping the usage of the cluster in order to parallelise a Stream Cipher (StreamCipher + StreamCipherSeek + KeyIvInit)

Solution:
Write a function generic enough to execute the algorithm on the cluster exploiting a triple buffering provided by the DMA.

Lifetime, a RUST feature, help us to keep alive the raw data pointers offered by the GAP SDK.

Method



Problem:

Masking the data transfer latency between L2 and Cluster L1 Memory.

Solution:

Create a crate able to abstract a buffer and the DMA in order

a

lvInit)

n on

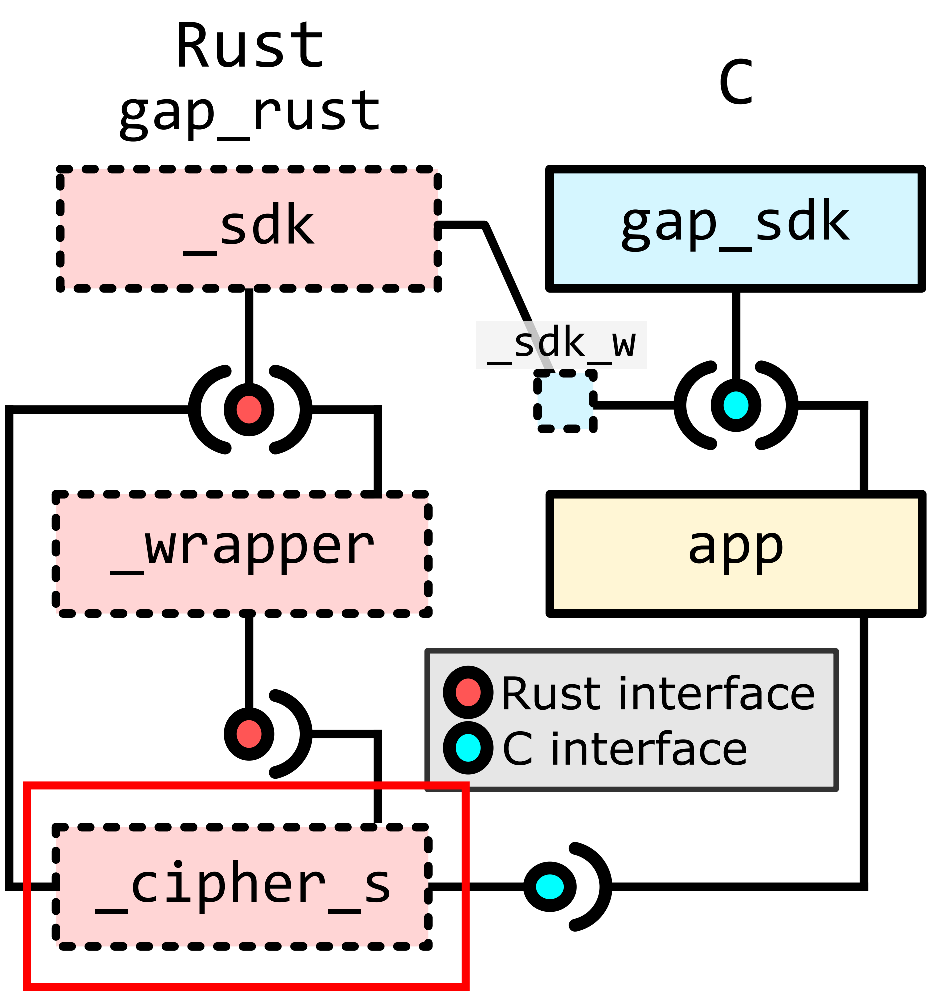
MA.

Lifetime, a RUST feature, help us to keep alive the raw data pointers offered by the GAP SDK.

A managed buffer in the L1 cache with automatic DMA transfers in and out based on rounds. Since we need to transfer back the modified data, we divide the L1 allocation into three buffers: work, pre-fetch, and commit.

```
B |------A-------|------B-------|------C--------|
0 | DMA-PREFETCH |              |               |
1 | WORK         | DMA-PREFETCH |               |
2 | DMA-COMMIT   | WORK         | DMA-PREFETCH  |
3 | DMA-PREFETCH | DMA-COMMIT   | WORK          |
```

Method



Problem:

Chacha20 QR() require bitwise operations like Rotate Left and Xor. PULP has an ISA extension that provides these opcodes, but they are not accessible to RUST for the lack of specific architecture support, riscv32imcXpulp, from the compiler.

Solution:

In gap_rust_cipher_s, we provide an enhanced version of the chacha20 core to exploit the p.ror opcode and optimise memory access.

More specifically, we preload a whole Chacha20 matrix using 16 registers, and with the usage of the RUST macro system, we can emit the desired opcode directly.

Moreover, we expose and use the hardware loop feature that allows PULP to mange in hardware the loop counter.

Problem:

te Left and
opcodes,
pecific
mpiler.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0		
funct7	rs2	rs1	funct3	rD	opcode		
000 0100	src2	src1	101	dest	011 0011	**p.ror**	**rD, rs1, rs2**

```
#[proc_macro]
pub fn ror(input: TokenStream) -> TokenStream {
  let (rd, rs1, rs2) = get_reg_operands(input);
  let hex = encode_hex(
    &[
      "0000100",
      &bin_5(rs2),
      &bin_5(rs1),
      "101",
      &bin_5(rd),
      "0110011",
    ].join(""),);
  let res = format!(".4byte {}", hex);
  quote::quote! { #res }.into()
}
```

rsion of the
imise

atrix using
o system, we

eature that
er.

Outilne

- Introduction
- Background
- Method
- **Results**

Results

We performed an encryption procedure of an increasing amount of data (from 1 Byte to 128 KiB) using three different implementations:

- single core without optimisation
- single core
- multicore
    - We varied the parallelism from two to eight cores in the multicore implementation.

We express the efficiency in terms of cycles needed to encrypt one byte (cB)

Results

We performed an encryption procedure of an increasing amount of data (from 1 Byte to 128 KiB) using three different implementations:

- single core without optimisation
- single core
- multicore
    - We varied the parallelism from two to eight cores in the multicore implementation.

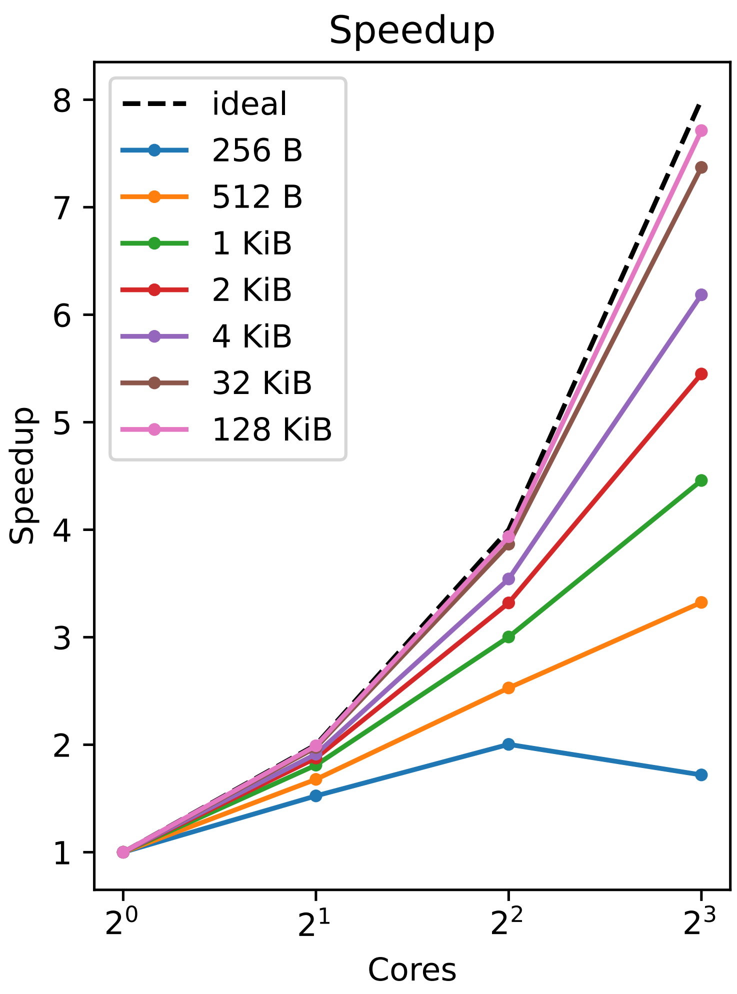
We express the efficiency in terms of cycles needed to encrypt one byte (cB)

Results for 128 KiB of payload

- Single Core no-opt: 92 cB
- Single Core opt: 16 cB

6x faster

Results



We performed an encryption procedure of an increasing amount of data (from 1 Byte to 128 KiB) using three different implementations:

- single core without optimisation
- single core
- multicore
  - We varied the parallelism from two to eight cores in the multicore implementation.

We express the efficiency in terms of cycles needed to encrypt one byte (cB)

Results for 128 KiB of payload

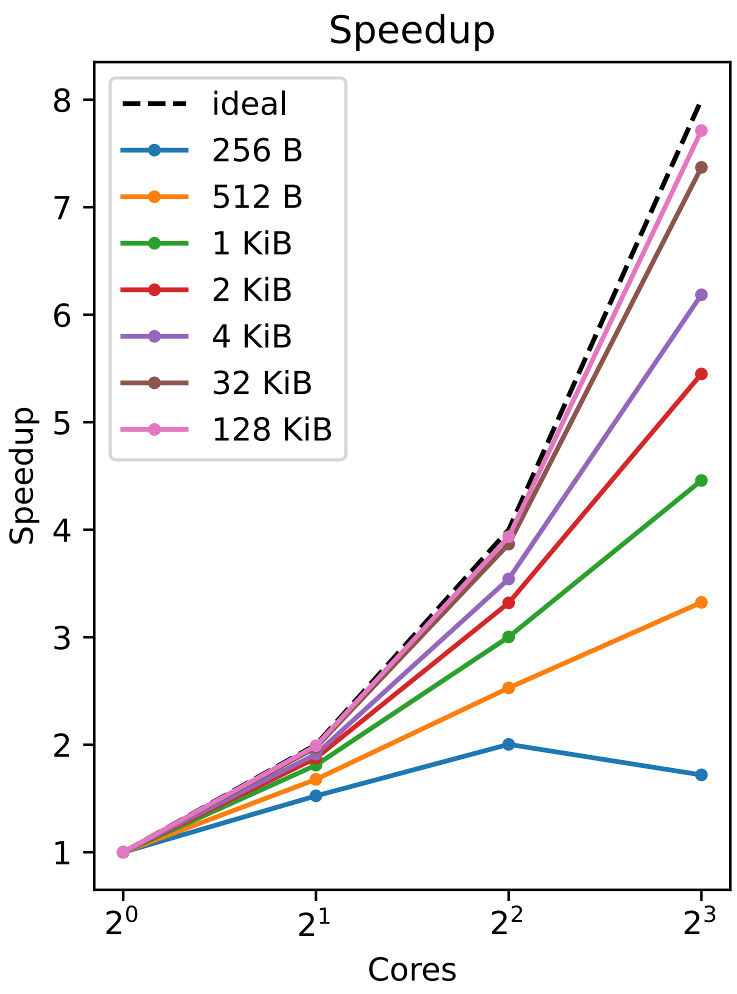
- Single Core no-opt: 92 cB
- Single Core opt: 16 cB

6x faster

Results for 128 KiB of payload

- CORES 2: 8.4 cB
- CORES 4: 4.3 cB
- **CORES 8: 2.2 cB (42x, 7.7x)**

Results



We performed an encryption procedure of an increasing amount of data (from 1 Byte to 128 KiB) using three different implementations:

- single core without optimisation
- single core
- multicore
  - We varied the parallelism from two to eight cores in the multicore implementation.

We express the efficiency in terms of cycles needed to encrypt one byte (cB)

Results for 128 KiB of payload

- Single Core no-opt: 92 cB
- Single Core opt: 16 cB

6x faster

Results for 128 KiB of payload

- CORES 2: 8.4 cB
- CORES 4: 4.3 cB
- **CORES 8: 2.2 cB (42x, 7.7x)**

Other architectures

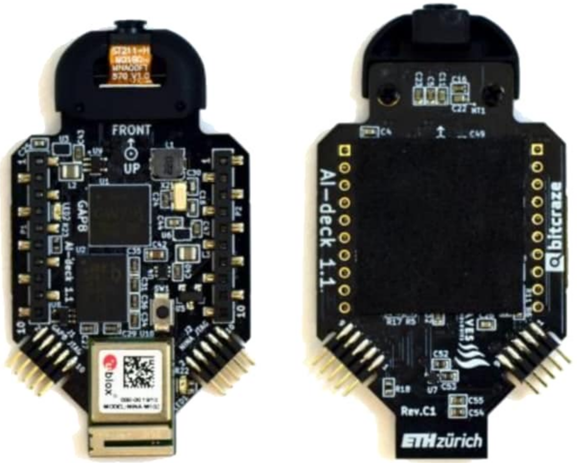
- U54 - SiFive Freedom U540
  - 35.3 cB
- A72 - Broadcom BCM2711
  - 5.3 cB
- POWER9 - IBM 02CY642
  - 2.6 cB
- Firestorm - Apple M
  - 2.0 cB
- Zen3 - AMD Ryzen 9 5950X
  - 1.04 cB

Results

In Real World scenario: 2.3 cB, no impact on fps!

CPS
Crazyflie

AI-Deck

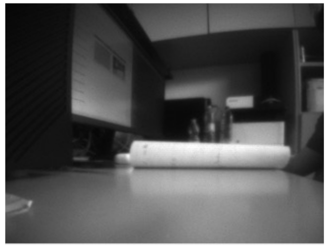


Example application
Encrypted video surveillance in micro-UAV

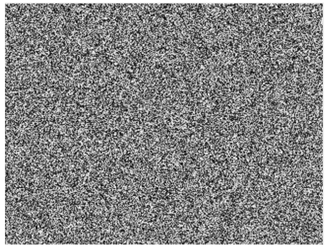
62 ms

RISC-V SoC

(60% of) 106ms @ 50MHz
(40% of) 67ms @ 150MHz
(30% of) 62ms @ 250MHz



324×244
greyscale frame



3.7ms @ 50MHz
2.0ms @ 100MHz
1.2ms @ 150MHz

2.3 cB with DMA
2.9 cB no DMA