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

Francesco Barchi
University of Bologna, Italy

Giacomo Pasini
University of Bologna, Italy

Emanuele Parisi
University of Bologna, Italy

Giuseppe Tagliavini
University of Bologna, Italy

Andrea Bartolini
University of Bologna, Italy

Andrea Acquaviva
University of Bologna, Italy

Abstract

Nowadays, the development of security applications is a relevant topic in the Internet of Things (IoT) and cyber-physical systems (CPS) fields. Different embedded architectures have been adopted in these areas, but the RISC-V parallel ultra-low-power (PULP) architecture stands out as a particularly efficient system. However, it has never been proposed to enable cryptography. In the context of video stream security, stream ciphers enable an efficient solution to ensure data privacy, and the exploitation of the PULP multi-core accelerator cluster paves the way to an efficient implementation of these ciphers. In this paper, we exploit the capability of the PULP architecture coupled with the code safety provided by the RUST programming language to design and implement an efficient stream encryption algorithm. We present a wrapper system between the development libraries of a PULP platform enabling the secure execution of a verified RUST-written implementation of ChaCha20 and AES-CTR, targeting a microdrones based video surveillance system. Experimental tests have resulted in an encryption efficiency of ChaCha20 of 2.3 cycles per Byte (cB), placing the resulting implementation at the state-of-the-art, in direct competition with higher-class architectures like Apple M1 (2.0 cB).

2012 ACM Subject Classification
Hardware → Emerging languages and compilers; Computer systems organization → Multicore architectures; Computer systems organization → Embedded software; Software and its engineering → Embedded software

Keywords and phrases Parallel Low-Power Embedded Systems, Rust, RISC-V, Stream Cipher

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2023.3
Category Invited Paper

Funding HORIZON-KDT-JU-2021-2-RIA-Focus-Topic-1, EdgeAI, 101097300

1 Introduction

New challenges require new technologies, and new technologies pose new challenges; this is particularly evident, in these last years, for Cyber-Physical Systems (CPS), whose challenges are becoming more and more evident. Will we be able to cope with the growing complexity of these devices that are increasingly interconnected and able to act and manipulate the surrounding reality? The CPS enabling factor is today, without doubt, the possibility to create pervasive and interconnected systems through the contribution of increasingly efficient
System of Chip (SoCs) and wireless communication technologies (e.g., 5G, NB-IoT). The CPS topic reached the peak of inflated expectation in a recent Gartner analysis [1], and CPS risk management, an innovation trigger, started its ascent among the potentially relevant topics for the next five years. In our vision, the risk management of CPS passes through two orthogonal fields. Open instruction set architectures (ISA) for the next generations of embedded computing systems and new programming languages capable of capturing in their expressiveness the management of memory at such a level as to guarantee at compile time the absence of the most common threats to integrity and security of working memory.

More specifically, we identified in RISC-V and Rust language two enabling factors for the future of CPS. In this work, we face the interoperability challenge of compiling and executing RUST encoded software in an existing RISC-V platform; PULP [15, 16]. PULP is a parallel ultra-low-power system composed of a cluster in a chip. This architecture is commercially available as the GAP8 SoC of GreenWaves Technology and adopted by BitCraze to implement an expansion deck of Crazyflie, a state-of-art miniaturised Unmanned Aerial Vehicle (UAV) [9, 14]. Although versions of PULP equipped with hardware accelerators for cryptography operations have been previously presented [7], GAP8 has no dedicated hardware for security. We select this architecture to create a first attempt to integrate and parallelise in the GAP8 cluster the RUST implementation of the most used stream ciphers, ChaCha20 and AES-CTR. We will then exploit the high parallelism provided by the PULP architecture to be able to accelerate any algorithm that implements specific RUST traits (StreamCipher, StreamCipherSeek, and KeyIvInit) defined in the cipher crate. Then, an analysis of unsafe code regions, an analysis of parallelism scalability, and the framework’s use in a real-world scenario will be provided. Moreover, we will consider a secure video surveillance scenario using a microdrone (the previously mentioned Crazyflie) equipped with a GAP8 processor capable of sending an encrypted video stream via the WiFi network. The main contributions of the work are: i) We designed a method to interact with the PULP cluster from Rust code, describing the Foreign Function Interface (FFI) we used to interact with the specific platform SDK; ii) We provide a security analysis of unsafe regions and an optimised version of ChaCha20 for PULP without compiler support; iii) We demonstrated that existing RUST code (already assessed from a security point of view) can efficiently be integrated with a CPS, showing a real usage application and the performances obtained.

The rest of the paper is organised as follows. In Section 2 we provide some background on Rust and PULP. In Section 3, we describe the procedure we follow to expose Rust code for the PULP cluster. Finally, we discuss the results obtained on GVSOC and GAP8 in Section 4.

2 Background and Related Work

2.1 Rust and RISC-V architectures

The Rust programming language positions itself as a language that offers both high-level safety and low-level control and speed. Thanks to its rich type system and ownership model, many classes of bugs (e.g., dangling pointers, double frees, and data races) are eradicated at compile time [11]. At the same time, having no runtime environment or garbage collection facilitates integration on different classes of devices as well as with other languages. While a GCC backend is in the works, the current compiler toolchain still relies on LLVM for code generation and general optimizations. Formally, Rust can target different platforms and any architecture for which LLVM has support.

The language comes with a rich standard library, but a significant effort has been put into separating what a core part of the language is and what is not. From this perspective, it is worth mentioning that the standard library (std) has no privileged support. This design
choice stems from the fact that these additional pieces, particularly the standard library (or part of it), might not always be available on all targets, especially in bare metal systems or without operating system support. For instance, it is necessary to provide platform-specific allocators to employ dynamic allocation.

A formal comparison between similar C and Rust codebases in terms of safety (primarily related to memory bugs) is not yet available, partially due to the novelty of Rust. However, many high-profile adopters like Amazon, Google, and Microsoft have started migrating parts of their systems from C to Rust due to the vast majority of security bugs related to memory safety. In addition, formal efforts to support these claims have been started in recent years. In particular, the RustBelt project [10] provided the first formal (and machine-checked) safety proof for a realistic subset of the language.

2.2 PULP – Parallel Ultra-Low-Power Platform

The target architecture of this work is the Parallel Ultra-Low-Power Platform (PULP), an open-source architecture SoC including a microcontroller-class RISC-V core (fabric controller) coupled with a cluster of RI5CY [8] cores (up to 16). RI5CY is a RISC-V based processor with dedicated extensions for Digital Signal Processing (DSP) and machine learning workloads. The cluster cores share a multi-banked scratchpad memory called Tightly-Coupled Data Memory (TCDM, or L1), enabling single-cycle data access and promoting data-parallel programming models such as OpenMP. At the SoC level, the architecture features an L2 memory hierarchy level composed of multi-banked scratchpad memory; the L2 access latency is one cycle for the fabric controller and 15 cycles for the cluster cores. A DMA engine enables data transfers between the two memory levels. We consider a PULP instance including 8 cores, 512 KiB L2 memory, and 64 KiB TCDM.

2.3 Stream Ciphers

Stream ciphers are a particular type of symmetric ciphers that encrypt a sequence of plaintext digits by combining it with an equal length pseudo-random digit stream, usually obtained from the key. They bear a resemblance with one-time-pad (OTP), although the keystream is not truly random in this case. Unlike raw block ciphers, they can work on messages of arbitrary length without padding and are thus generally easier to employ in different application contexts.

While stream ciphers are enough for confidentiality, they do not always guarantee the authenticity of the ciphertext. For this reason, it is generally recommended to use Authenticated Encryption with Associated Data (AEAD), which combines encryption with some mechanism for tampering prevention. Examples of such authenticated ciphers are ChaCha20Poly1305, combining ChaCha20 with Poly1305, or AES with GCM or CCM modes. As support for the relevance of the aforementioned ciphers, they are the only ones allowed in TLS 1.3. [5]

In this work, we will only focus on the encryption part, which is suitable for parallelization, leaving authentication for future work. Note that the encryption component can be entirely reused when implementing the full AEAD algorithm. We thus chose ChaCha20 and AES-CTR for integration in our system, focusing primarily on ChaCha20 for its simplicity [13].
3 Methods

Figure 1 depicts the structure of modules we developed (dashed lines) to allow a GAP8 application (yellow box) to use a stream cipher implemented in RUST, exploiting the PULP extensions and the cluster-on-chip parallelism. Coloured dots are the interfaces between language domains. When RUST code needs to use functions implemented in C, we use the Foreign Function Interface (FFI) capabilities of Rust. On the contrary, when C code needs to use functions implemented in Rust, we use a `extern` block to guarantee the same memory layout C would use. The module structure is divided into three components:

- **gap_rust_sdk** is a wrapper of **gap_sdk**, it is described in Section 3.1.
- **gap_rust_sdk_w** is a C library developed to decouple certain **gap_sdk** functions that are otherwise not accessible to Rust code.
- **gap_rust_wrapper** is the wrapper between the stream cipher implementation and the **gap_rust_sdk**. It is described in Section 3.2.
- **gap_rust_cipher_s** contains the entry point functions (C compatible) to execute a cipher procedure. It also contains the optimised code of streaming algorithms.

3.1 RUST Wrapper for PULP-SDK

The PULP SDK contains all the software stack of the PULP platform, including a C library that exposes all the features and capabilities to the programmer. Such a library is the perfect starting point for porting PULP functionalities to the Rust world, as it acts as the basic building block on top of which other libraries can provide their services. Thanks to Rust native compatibility with C, making those functionalities available in Rust is as easy as writing FFI bindings and linking against the PULP library binary. Automated tools to write the bindings exist but require the source code to be processed with Clang/LLVM. Unfortunately, this is not always possible due to compiler-dependent extensions in some implementations, like in our scenario with the PULP SDK that depends on specific GCC extensions available in the PULP toolchain.

**pulp-sdk-rust**, the Rust port of PULP SDK, comprises two different parts. One essentially exposes as-is the functions of PULP SDK as Rust functions. For this purpose, apart from copying the signatures of the selected functions, it is necessary to map C types to Rust. The primitive C types have an equivalent Rust type either in the core language itself, like all
numeric types, or in the \textit{cty} library, like void pointers. However, custom structs usually require a corresponding field-per-field definition in Rust, where the use of the \texttt{#[repr(C)]} attribute guarantees the same memory layout as C.

A special case is given by opaque structs or structs, for which only pointers are used, and no allocation in the Rust world is necessary. In this case, while void pointers are a valid representation, it is preferable to use opaque Rust structs to accurately map each type and provide type safety for function arguments. To represent such opaque structs in Rust, we can create a type with [2]: i) at least a private field so that it is not possible to instantiate it outside of the module it is defined in; ii) attribute \texttt{#[repr(C)]} across FFI boundaries; iii) special markers for the compiler not to derive any unwanted property. Rust has special traits, called markers, to represent intrinsic properties of types. The ones that we are interested in here are \texttt{Send}, \texttt{Sync}, and \texttt{Unpin}. \texttt{Send} and \texttt{Sync} are used to regulate how types can be moved and shared in a multi-threaded environment. \texttt{Unpin} is used to signal that a type can be moved in memory after being explicitly pinned. Since we do not know how the C code accesses those structs and what they represent, a safe choice is not to let the Rust compiler infer any of those traits, which are automatically derived in regular circumstances.

An example of an opaque struct in Rust is:

\begin{verbatim}
//\[repr(C)\]
pub struct Foo {
    _data: [u8; 0],
    _marker: PhantomData<*mut u8, PhantomPinned>
}
\end{verbatim}

However, in some cases, critical features like DMA functionalities are declared as \texttt{static\ inline} functions in the PULP SDK. Unfortunately, this means those functions are not visible to the linker and cannot be directly exposed to Rust code. Since maintaining compatibility with the PULP SDK is an important requirement, we chose to write a small C wrapper and provide it in the linking step like so:

\begin{verbatim}
void pi_cl_ram_read_wait_wrap(pi_cl_ram_req_t* r)
{
    pi_cl_ram_read_wait(r);
}
\end{verbatim}

Thanks to Cargo, the official Rust package manager, such a wrapper library is built and linked at compile time without any user intervention.

This first component alone is enough to provide all PULP-related functionalities in Rust, but it is not very ergonomic to use. For instance, it is likely to contain raw pointers as function arguments, which are unsafe to use in Rust and require special care. Hence, a good port should provide Rust abstractions over those functionalities when possible and encapsulate the use of complex or unsafe components. For example, pulp-sdk-rust exposes an abstraction over the cluster type, which takes care of the initialisation and offloading of computation, all of which require to use possibly unsafe FFI functions. Designing a correct API for offloading computation to the cluster requires great care since it is necessary to handle multiple threads without native Rust support.

The Rust language provides two important marker traits, Send and Sync, specifically to handle concurrency and avoid data races at compile time.

- A type is \texttt{Send} if it is safe to send it to another thread.
- A type is \texttt{Sync} if it is safe to share between threads. A generic type $T$ is \texttt{Sync} if and only if a reference to $T$ is \texttt{Send}.

\texttt{pi_cl_team_fork}, the C function to fork execution on the cluster cores, accepts as arguments a function to execute in each core and a pointer to a memory location which will be provided to all of the function instances. Apart from using raw pointers, which is unsafe
on its own, it is easy to see that providing access to some data type that does not implement Send or Sync could break Rust guarantees. For example, sharing a reference to Rc, Rust single-threaded reference-counting pointer, would result in data races if multiple cluster cores try to update the reference counter simultaneously. Indeed, the safe API for computation offloading provides access to a reference of a Sync type.

### 3.2 RUST Streaming Cipher Wrapper for PULP Cluster

To provide a reusable component for parallel computation on PULP systems, we designed the PULP Stream Cipher wrapper. It is a bridge between the hardware specifics on one side and a generic stream cipher implementation on the other. It builds on top of traits from a popular crate [3] and is general enough so that it can be used with different algorithms. In practice, it schedules encryption/decryption for execution on the cores of the PULP cluster. Such stream ciphers, apart from implementing the traits StreamCipher and KeyIvInit, which are relatively standard and for which the requirements can be easily relaxed or adapted, need to support seeking freely within the keystream to allow efficient parallelization. This way, different cores can work on different portions of the stream of bytes without any overlap or additional work required. Examples of stream ciphers that support this operation are ChaCha20 and block ciphers operating in CTR mode.

To fully exploit the PULP cluster, it is necessary to accurately design memory accesses. Working directly from L2 memory in the cluster could result in significant performance penalties due to contention on the memory bus, while the use of L1 memory allows better latency and throughput for both reads and writes. However, moving data from L2 to L1 does not come for free and has to be explicitly instructed. Fortunately, the PULP system provides a DMA and a uDMA specifically for this task, offloading the transfer from external memory (L2 or RAM) to the cluster L1. In our application, we need to copy the plaintext to L1, encrypt or decrypt it, and then copy it back to external memory. Naturally, the size of L1 is limited, and we cannot expect to fully fit every message there as we want to enable the processing of messages bigger than L1 (in our case 64 KiB). Thus, we split the input message into multiple chunks so that each one can fit entirely into L1, and we process them incrementally one at a time.

To avoid waiting for the completion of DMA/uDMA transfers, we designed a solution that makes use of triple buffering, essentially keeping three separate buffers in L1 memory, letting them be A, B, and C. Each portion has an assigned role: working buffer, pre-fetch, and commit. The working buffer is used for computation, pre-fetch to load the next chunk of the input message, and commit to store the processed data back into external memory. At the beginning of the program execution, we load the first portions of the plaintext into A. Then, processing starts on portion A, which is the current work buffer, while the DMA/uDMA is instructed to load the next chunk of the plaintext into portion B, the pre-fetch buffer. After all of the cores have completed processing on A and the DMA/uDMA has loaded data into B, roles change. Portion A, which contains the encrypted/decrypted message, now becomes the commit buffer, and the DMA/uDMA is instructed to copy it back to external memory. Portion B, which contains new input data, will serve as the work buffer, and the old commit buffer (C) will become the pre-fetch buffer for the next chunk. The result is that each portion is assigned a new “role” each round in a round-robin fashion until all of the input has been processed.

We now focus on how processing on the work buffer is handled. To avoid data races and allow concurrent access to multiple cores, the working buffer is partitioned into chunks, one assigned to each core. The chunks are non-overlapping (i.e., in other words, core pointers do not alias). This property enables each core to work independently, and the only synchronisation needed is the one with the DMA/uDMA, controlled by core 0.
3.3 Architecture Specific Optimization

ChaCha20 is the cipher we primarily work on and optimized for this task. As described in 2.3, it is a modern, high-speed, low-footprint algorithm, easy to implement in software without having access to specific hardware instructions and features. On the other side, a constant time AES software implementation requires special care, to the point where multiple techniques have been developed, like bitslicing [12] and fixslicing [6]. To obtain a highly efficient implementation of ChaCha20, we started from the software implementation in [3] and optimized the parts that resulted in sub-optimal performance in our target system. Unfortunately, while the Rust compiler is able to compile for generic RISC-V targets, it cannot yet make use of specific PULP extensions like hardware loops, post-increment load and stores, or bit manipulation operations. In addition, memory accesses are not always optimal for a system without out-of-order execution like PULP and often result in stalls.

To avoid these limitations, we implemented the ChaCha20 core loop directly in assembly, which is quite easy to do given the simplicity of the ChaCha20 algorithm. Note that this is not in contradiction with Rust philosophy: it is true that by writing in assembly we lose some of the guarantees of Rust, but it is only for a very limited (albeit extremely hot) portion of the codebase, and we can treat it as a standard black-box function from outside, building the rest of the framework in plain Rust. Even better, once the PULP integration for the Rust compiler is completed and all hardware features supported, we can just use a full Rust implementation.

The ChaCha20 quarter-round is only comprised of 12 add, xor, and shifts instructions and can be very easily implemented using inline assembly. Starting from the quarter-round, we can obtain a full round by essentially replicating it on different data. The Rust declarative macro system allows us to do that without having to write it entirely by hand since inline assembly has to be provided at compile time, and we wanted to avoid unnecessary loops. However, there is a catch: mnemonics for PULP-specific extensions cannot be used as they are not supported by the compiler backend. An interesting solution to this can be designed on top of Rust procedural macros, which are more expressive than declarative macros: they allow to write arbitrary Rust code that consumes and produces Rust syntax. In our case, we implemented a macro that produces a raw hex-encoded assembly instruction for each of the unsupported mnemonics while still being very descriptive at the call site. For example, the implementation of a macro for right rotate bit-wise operation would look like this:

```rust
use std::convert::TryInto;

fn ror!(in: TokenStream) -> TokenStream {  
    let (rd, rs1, rs2) = get_operands(in);
    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()  
    }
}
```

It is noteworthy how the call site of such a macro is extremely similar to how a programmer would write it by using native mnemonics `ror t0, t0, t1` with macro `ror!(t0, t0, t1)`.
3.4 Safety Evaluation of Rust Wrappers

Where the Rust safety features clash with optimizations, or when we need to interact with other languages like C, it is possible and often necessary to temporarily “disable” safety checks and rely on the total power without control given by raw pointers. Of course, with great power comes great responsibility, and it is necessary to guarantee the correct usage of those unsafe functions, not to undermine the foundations of the whole system. The good news is that a detailed check for such errors has to be performed only on a relatively small portion of the program. What is usually done when developing a library like pulp-sdk-rust is to encapsulate the unsafe Rust features under a safe API and only expose the safe ones to the outside world. In this sense, unsafe Rust is transparent to users.

We can now examine why and where we reverted to unsafe Rust in the implementation: The Good, The Bad and The Ugly.

3.4.1 The Good: FFI bindings for the pulp_sdk

C libraries often expose APIs that make use of raw pointers, which fall outside of Rust’s safe memory model. In addition, the Rust compiler cannot guarantee that those functions are valid for all possible inputs or do not mess with memory in invalid ways. Thus, foreign functions are assumed to be unsafe in Rust and require unsafe blocks as a promise to the compiler that everything contained within truly is safe. To improve the usability on the Rust side, we provided safe Rust wrappers, where possible, for some of the library functions, for which we rely on PULP SDK correctness.

3.4.2 The Bad: Optimizations

The Rust compiler does not always provide the best optimised code for a system like PULP and is currently lacking support for PULP extensions, which results in subpar performance when writing idiomatic Rust code (e.g., iterators). For example, consider the following function written using iterators. It takes two slices as input, computes the element-wise sum, and stores the result in the first slice.

```rust
fn rotate_right_slow(a: &mut [u32], b: &[u32]) {
    for (a, b) in a.iter_mut().zip(b.iter()) {
        *a = *a + b
    }
}
```

As of rustc v1.63, with options `-C opt-level=2` `-target riscv32imc-unknown-none-elf`, it outputs the following assembly code for the inner loop:

```
.LBB0_3:
    lw   a1, 0(a0)
    lw   a4, 0(a2)
    add  a1, a1, a4
    sw   a1, 0(a0)
    addi a3, a3, -1
    addi a0, a0, 4
    addi a2, a2, 4
    bnez  a3, .LBB0_3
```

This code stalls while executing the third instruction as the second operand (a4) is not available yet. Resolving this stall requires moving any of the following `addi` instructions between the second and the third instruction. For this reason, the core of the ChaCha20...
algorithm has been written in assembly. However, it is noteworthy that our assembly implementation delegates all memory write operations to the Rust language through inline assembly macro outputs, thus operating in readonly mode and ruling out any memory corruption.

While it would certainly be better to improve the compiler understanding of the target system, in the meanwhile, it is possible to use unsafe code for optimization, which is generally speaking an accepted practice for hot paths.

3.4.3 The Ugly: Synchronization primitives

No native support for PULP systems also means no support for synchronisation primitives available. A complete and throughout approach, which would require providing the basic primitives in Rust (Mutex, RwLock) and adapting them for use both within the cluster and between the cluster and fabric controller, necessitates of a significant amount of work and is left for future implementation.

4 Results

This section is divided into two parts. The first part describes a virtual environment through which we validated the system presented in the previous section. The second part presents a real-life scenario consisting of a video surveillance application implemented on a micro UAV by operating application-level encryption.

4.1 Wrapper analysis

We used GVSOC to obtain the execution traces and infer the speedup gain obtainable in ChaCha20 and AES-CTR (written in RUST) when executing the code on PULP. GVSOC is the virtual platform included in the PULP-SDK. It is faster compared to an RTL simulation and provides good cycle accuracy. Such properties are key requirements for integration into development flows. GVSOC also provides execution traces describing cluster components’ status during the program execution. This tool was also helpful in validating code and debugging code fragments during development. 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, and multicore. Moreover, we varied the parallelism from two to eight cores in the multicore implementation. The results were obtained by analysing the GVSOC traces. We express the efficiency in terms of cycles needed to encrypt one byte (cB). This unit of measurement allows us to evaluate the goodness of the implementation and the contribution of the optimisations made. Taking a payload of 128 KiB as a reference, we started evaluating the software implementation provided in the RUST crate of ChaCha20. Without any optimisation, we obtain 92 cB. By running the optimised version of ChaCha20, which we developed exploiting the architectural extensions of PULP, we obtain 16 cB, an improvement of about 6 times. The optimised and parallelised versions (2 \times, 4 \times, 8 \times) obtain 8.4 cB, 4.3 cB and 2.2 cB respectively. The optimised version on eight cores is about 42 \times faster than the starting version and 7.7 \times faster than the single-core version. Figure 2 shows speedup curves obtained varying the parallelism and the payload. These graphs clearly highlight the effect of memory latencies (L2-L1 movements) when the triple buffer is not fully utilised. With 8 KiB buffers, ideal parallelism is only achieved if the payload is greater than 24 KiB. Results in cB of some reference architectures for long messages are: 35.3 cB (riscv64) U54 – SiFive Freedom U540, 5.3 cB for (aarch64) A72 – Broadcom BCM2711, 2.6 cB for (ppc64) POWER9 – IBM 02CY642, 2.0 cB for (aarch64) Firestorm – Apple M1, 1.04 cB for (amd64) Zen3 – AMD Ryzen 9 5950X [4].

The single core execution of AES-CTR has an efficiency of 128 cB. The parallelised versions (2 \times, 4 \times, 8 \times) obtain 79 cB, 36 cB and 18 cB respectively. The version with maximum parallelism is 7.1 \times faster than the single-core version.

4.2 Real World Scenario

In this test environment, we implemented a video surveillance application. Specifically, using a Crazyflie equipped with an AI-deck, it was possible to create an encrypted video stream. We integrated the ChaCha20 implementation into a video stream application written for FreeRTOS. The application captures a 324 \times 244 greyscale frame from a camera, encrypts the frame on the cluster and sends the encrypted frame to an ESP32 processor. The latter is then in charge of sending data to a remote application using WiFi. This scenario can be a seed for a zero-trust CPS, where the component responsible for transmitting the data cannot steal sensitive information thanks to application-level encryption.

We used a Crazyflie 2.1, an AI-deck 1.1, a GAP8 rev.C, and an Olimex ARM-USB-OCD-H needed to flash the GAP8 firmware. We measured the clock cycles required to encrypt a single byte (cB) and the time required to complete the three phases: frame acquisition, frame encryption, and frame sending (in turn, divided into communication with ESP32 and WiFi communication). In the GAP8, we varied the working frequency of the fabric controller (FC) and the cluster cores (CL) to characterise the execution times required by the application.

Acquiring a frame takes about 62 ms regardless of the fabric controller’s working frequency. Forwarding a frame takes 106 ms when the fabric controller runs at 50 MHz, 67 ms at 150 MHz, and 62 ms at 250 MHz. This time comprises two parts: the time required for the communication between GAP8 and ESP32 (SPI) and WiFi transmission. The first part (GAP8 and ESP32 communication) takes 60% of the frame sending time when the fabric controller runs at 50 MHz, 40% at 150 MHz, and about 30% at 250 MHz. The PULP-optimised and cluster-parallelised ChaCha20 implementation encrypts a frame in 3.7 ms when the cluster cores run at 50 MHz, 2.0 ms at 100 MHz and 1.2 ms at 150 MHz. This results in an efficiency of 2.3 cB, confirming the results obtained with the virtual platform. Eliminating the use of DMA for L2-L1 transfers reduces the efficiency from 2.3 to 2.9 cycles/Byte.
In the fastest configuration tested (1.2 V, FC 250 MHz, CL 150 MHz), an encrypted video stream is obtained at approximately 8 fps. The encryption time is a secondary factor compared to the other two phases.

5 Conclusion

This work represents the first attempt to add software support for stream ciphers in the software ecosystem of the PULP architecture, a state-of-art ultra-low-power embedded microcontroller equipped with a multi-core accelerator cluster. PULP has become a reference architecture for mission computer and video stream processing in micro UAVs, an application field characterized by stringent security requirements. Exploiting the capabilities of the RUST programming language in terms of code safety and modularity, we designed a wrapper for the PULP runtime library to enable the secure execution of a verified RUST-written implementation of ChaCha20 and AES-CTR algorithms. Our experimental assessment on a commercial device demonstrated a high encryption efficiency (2.3 cB for ChaCha20), a result aligned with higher-class architectures but achieved on a resource-constrained embedded device. For comparison, SiFive U540 obtain 35.3 cB and Apple M1 2.0 cB in ChaCha20.

In future work, we plan to extend the RUST cryptographic library by including an implementation of AEAD optimized for the PULP target. We will also integrate a new backend compiler into the RUST toolchain to support the experimental PULP LLVM toolchain and guarantee seamless integration between the PULP SDK ecosystem and the RUST language. Finally, we will also implement a set of native synchronization primitives in Rust working for cluster cores and fabric controller, providing better safety guarantees for implementing parallel algorithms.

References

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