Journal of
Low Power Electronics and Applications

MDPI

Article

FAC-V: An FPGA-Based AES Coprocessor for RISC-V

Tiago Gomes [1,*], Pedro Sousa [1], Miguel Silva [1], Mongkol Ekpanyapong [2] and Sandro Pinto [1]

[1] ALGORITMI Research Centre/LASI, University of Minho, 4800-058 Guimarães, Portugal
[2] School of Engineering and Technology, Asian Institute of Technology, Pathum Thani 12120, Thailand
* Correspondence: mr.gomes@dei.uminho.pt

Abstract: In the new Internet of Things (IoT) era, embedded Field-Programmable Gate Array (FPGA) technology is enabling the deployment of custom-tailored embedded IoT solutions for handling different application requirements and workloads. Combined with the open RISC-V Instruction Set Architecture (ISA), the FPGA technology provides endless opportunities to create reconfigurable IoT devices with different accelerators and coprocessors tightly and loosely coupled with the processor. When connecting IoT devices to the Internet, secure communications and data exchange are major concerns. However, adding security features requires extra capabilities from the already resource-constrained IoT devices. This article presents the FAC-V coprocessor, which is an FPGA-based solution for an RISC-V processor that can be deployed following two different coupling styles. FAC-V implements in hardware the Advanced Encryption Standard (AES), one of the most widely used cryptographic algorithms in IoT low-end devices, at the cost of few FPGA resources. The conducted experiments demonstrate that FAC-V can achieve performance improvements of several orders of magnitude when compared to the software-only AES implementation; e.g., encrypting a message of 16 bytes with AES-256 can reach a performance gain of around 8000× with an energy consumption of 0.1 µJ.

Keywords: RISC-V; Internet of Things (IoT); Field-Programmable Gate Array (FPGA); Advanced Encryption Standard (AES); RISC-V coprocessor

check for updates

Citation: Gomes, T.; Sousa, P.; Silva, M.; Ekpanyapong, M.; Pinto, S. FAC-V: An FPGA-Based AES Coprocessor for RISC-V. *J. Low Power Electron. Appl.* **2022**, *12*, 50. https://doi.org/10.3390/jlpea12040050

Academic Editors: Teresa Cervero, Kevin Martin, Mario Kovač and Maurizio Martina

Received: 6 September 2022
Accepted: 23 September 2022
Published: 27 September 2022

1. Introduction

The Internet of Things (IoT) is enabling the shift of computing workloads from traditional cloud facilities to the edge [1]. Nonetheless, most of the devices collaborating in this massive network infrastructure are often resource-constrained, which makes the handling of workloads intended for high-end devices quite challenging [2]. Moreover, fulfilling the different requirements for deploying IoT devices, e.g., real-time data gathering and processing, low-power festures, and connectivity aspects, often requires each final solution to be individually tailored to fit the different hardware and software constraints dictated by distinct target applications [3,4]. To cope with such diversity, and due to the lack of one-size-fits-all solutions, the software and hardware development processes face several trade-offs regarding power consumption, form factor, performance, etc.

Until recent years, Field-Programmable Gate Array (FPGA) technology was not suitable to be adopted in most IoT applications, which was mainly due to the high power consumption, large form factor, and difficulty of integration with processors. However, with the emergence of embedded FPGA solutions (low-power FPGA), these barriers have been minimized, and the applicability of this technology in low-end IoT devices is considerably increasing [5]. The industry is starting to adopt reconfigurable platforms to achieve desired metrics in custom-tailored embedded IoT solutions [1], outgrowing the capabilities of traditional Microcontroller Units (MCUs) by enabling the development of accelerators in hardware, which is often connected to the MCU as standard peripherals [6]. Nevertheless, deploying and optimizing accelerators on FPGA still faces several challenges usually imposed by closed computer architectures, which started to be mitigated with the emergence of the Reduced Instruction Set Computer (RISC)-V [7,8].

RISC-V is a novel open Instruction Set Architecture (ISA) that follows a RISC design. It was created to support a broad range of devices, spanning from high-performance application processors to low-power embedded microcontrollers, with applicability in a wide number of applications and scenarios. RISC-V enables a new level of software and hardware freedom by allowing the easy integration of dedicated and custom-tailored hardware devices with the application software [9–14]. Some RISC-V implementations, such as the Rocket core [15], already extend the ISA by defining a subset of instructions for user-defined coprocessors following a tightly coupled implementation while also providing a memory-mapped interface to manage coprocessors in a loosely coupled approach.

When deploying an IoT device, in addition to the hardware platform and computer architecture, programmers often resort to embedded IoT Operating Systems (OSes), such as Contiki-NG [16] and RIOT [17], among others [18,19]. These solutions provide a ready-to-use network stack compliant with several communication standards and facilitate the development and deployment of the final application. However, with the connectivity requirements arise several end-to-end security concerns, even at the network edge. Nowadays, every IoT device must perform secure data exchange and storage, supporting both application- and link-layer security for data integrity and encryption. Such features are also embedded in the network stack provided by the OS, which includes software-based security algorithms, such as the Advanced Encryption Standard (AES), the Triple Data Encryption Standard (3DES), the Rivest–Shamir–Adleman (RSA), and Datagram Transport Layer Security (DTLS), among others [20–23]. Nonetheless, the addition of more (and complex) processing overloads to low-end devices, such as the security layers, increases the need for accelerated solutions to mitigate performance and real-time concerns.

With all this in mind, this article presents FAC-V, an FPGA-based AES coprocessor for reconfigurable IoT devices. FAC-V is specially designed to provide hardware acceleration to connected low-end IoT solutions that require secure communications and secure data transfers based on the AES standard at different layers of the network stack. The FAC-V accelerator is implemented on a softcore RISC-V processor, and by taking advantage of the RISC-V ISA extensions, it is possible to be deployed following both the tightly- and loosely coupled approaches without requiring complex software abstraction layers. The main contributions of this paper are:

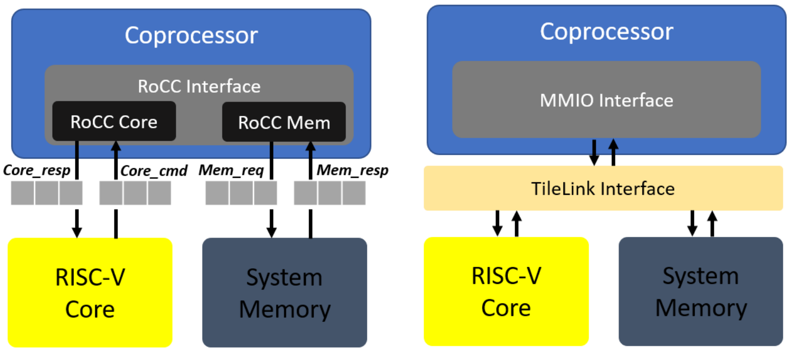
1. An AES coprocessor for low-end reconfigurable IoT devices that can be deployed following two different coupling approaches;
2. A user-friendly Application Programming Interface (API) that provides a complete abstraction from the accelerator and can be easily integrated with different IoT OSes or baremetal applications;
3. A complete evaluation and benchmarking of the FAC-V accelerator in terms of FPGA resources, latency, performance, and power consumption;
4. The integration and performance evaluation of FAC-V with RIOT, which is a well-known OS tailored for low-end IoT devices.

2. Background and Related Work

Regarding the development and deployment of the proposed cryptographic coprocessor, it is important to understand the different coupling interfaces provided by a RISC-V core, and the related work on hardware-based implementations of the AES algorithm.

2.1. Loosely and Tightly-Coupling Approaches in RISC-V

For the development of the FAC-V accelerator, it was used the Rocket core, which is a RISC-V processor that implements different variants of the RISC-V ISA, e.g., the RV64G and the RV32-IMAC, provides one integer Arithmetic Logic Unit (ALU) and an optional Floating-Point Unit (FPU), and includes a coprocessor interface, which is called Rocket Chip Coprocessor (RoCC) [15]. The customization and generation of a Rocket core processor, before being deployed in the target hardware platform, is completed with the Rocket chip generator tool and Chisel [24], which is a high-level Hardware Description Language (HDL) used to describe digital electronics and circuits at the Register-Transfer Level (RTL). This development process also enables the implementation of coprocessors following tightly and loosely coupled approaches, as depicted in Figure 1.



(**a**) (**b**)

Figure 1. Coupling styles on an RISC-V Rocket core: (**a**) Tightly coupled. (**b**) Loosely coupled.

In the tightly coupled implementation, as shown in Figure 1a, the RISC-V Rocket core uses the RoCC interface, which allows for creating custom processor instructions following the RISC-V R-type instruction format (as shown in Figure 2) to communicate with accelerators. The RoCC interface is composed of two sub-modules, the RoCC Core and the RoCC memory, which provide a command/response interface to communicate with the RISC-V Core. The RISC-V Core sends the instructions to the coprocessor via the *Core_cmd* interface (including registers) and receives the response in the *Core_resp* interface. The RoCC memory allows for the accelerator to perform memory-related accesses using the same command/response strategy. These interfaces also include other control signals, such as *busy* bits, IRQs, etc.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

Figure 2. ROCC instruction format (R-type).

In a loosely coupled way, as shown in Figure 1b, the RISC-V Rocket core supports the Tilelink Interface [25]. Tilelink follows a traditional shared-memory approach that maps memory accesses to a physical memory space. With a reserved address map description and following Memory-mapped IO (MMIO) requests, the TileLink can be a suitable solution based on a memory interface for communicating with accelerators, coprocessors, and Direct Memory Access (DMA) engines.

2.2. AES Accelerators

The AES standard is one of the most widely used cryptosystems in IoT low-end devices, which is mainly due to its easy implementation both in software and hardware, even in resource-constrained systems. The AES is a symmetric key encryption algorithm, which means that the same key is used in the data encryption and decryption operations. This algorithm features an initial key expansion step, which is followed by a data block encryption/decryption phase. The latter is executed in several rounds, using the generated keys and performing substitutions, transpositions, and linear combinations of bytes forming the data blocks. However, despite being lightweight and easy to deploy, security algorithms have more memory and processing power requirements due to the high number of operations that need to be executed, which may lead to some performance penalties [26]. To tackle these issues, some IoT devices already provide hardware-based loosely coupled implementations of some security algorithms [27–29]. Nonetheless, the performance gains come at the cost of extra hardware added to the device.

As the RISC-V has grown in popularity, several domain-specific coprocessor units have been implemented, and the first steps toward making an AES coprocessor, both in Application-Specific Integrated Circuit (ASIC) [30–35] or FPGA technologies [36–38], have already been taken. Table 1 shows a comparison among different state-of-the-art AES implementations according to the year, the technology, the type of architecture, the frequency, the number of clock cycles required per encryption, and the throughput. In comparison, this work intends to explore two different implementation strategies (rolling and unrolling), initially aiming for a high operating frequency and gradually moving toward a small circuit area and reduced power consumption. However, in these solutions, there is scarce research on evaluating the best methods of coupling accelerators to the core. Now, especially with the RISC-V Rocket Core that facilitates new methods of tightly coupling accelerators, it has become very important to evaluate the hardware costs, performance advantages, and which method is most suitable for resource-constrained systems. From the available works, most of them resort to a loosely coupled approach with a dedicated memory interface or are deployed in an ASIC solution.

Table 1. State-of-the-art AES implementations (2017–2020).

Work	Year	Technology	AES Architecture *	Frequency (MHz)	Cycles/ Encryption **	Throughput **
Agwa et al. [34]	2017	ASIC (Loosely)	Rolling	666	-	2.601 Gbps
Bui et al. [35]	2017	ASIC (Loosely)	Rolling	10	44	28 Mbps
Lu et al. [30]	2018	ASIC (-)	Rolling	50	213	30.05 Mbps
Banerjee et al. [31]	2019	ASIC (Loosely)	Rolling	16	11	-
Al-Gailani et al. [38]	2019	FPGA (-)	Unrolling	158	1	20.3 Gbps
Shahbazi et al. [36]	2020	FPGA (-)	Unrolling	622.4	1	79.7 Gbps
Marshall et al. [32]	2020	ASIC (Tightly)	Rolling and Unrolling	-	18–30	-
Pan et al. [33]	2021	FPGA and ASIC (Tightly)	Rolling	100	11–19	471–695 Mbps

* Unrolling structure (AES rounds are executed in parallel) and Rolling structure (AES rounds are executed recursively). ** Data excluding the cycle cost of communicating with peripherals.

3. FAC-V Design and Implementation

The overall system architecture, as depicted in Figure 3, follows a hardware–software co-design approach, enabling the fast deployment and evaluation of FAC-V and further comparison with state-of-the-art AES cryptography solutions. The architecture is composed of two main blocks: (i) the FAC-V coprocessor and (ii) the Abstraction Layer. Designed with flexibility in mind, the FAC-V can be deployed following both the tightly and loosely coupling styles.

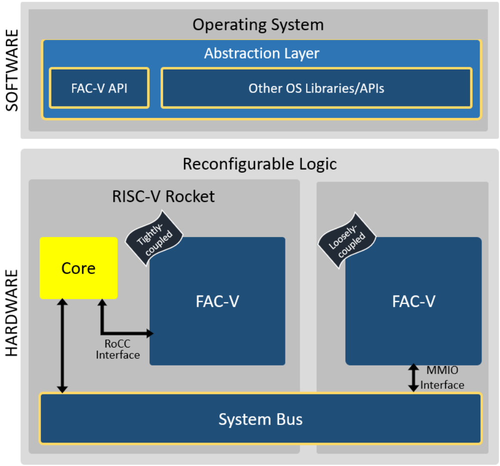


Figure 3. FAC-V architecture overview.

In the tightly coupled approach, the coprocessor is connected to the core through the RoCC interface, while in the loosely coupled approach, the accelerator is memory-mapped and connected with the Central Processing Unit (CPU) through the TileLink interface. The Abstraction Layer provides the FAC-V API that enables the interface between software cryptographic tasks and the hardware coprocessor while also supporting in software the same services provided by the hardware version of the accelerator. This eases the integration of the coprocessor with OSes that already support in their network stack the security layers, whose functionalities can now be remapped to respective hardware blocks.

3.1. FAC-V Coprocessor Architecture

The FAC-V coprocessor was designed following a modular methodology, and it was developed with the Scala-based hardware design language Chisel. Figure 4 depicts the FAC-V coprocessor architecture and its internal modules. The accelerator is divided into three main blocks: (i) the interface module, responsible for the communication and data exchange between the control unit, the CPU, and the system memory; (ii) the control unit, which manages the other modules according to the received inputs and outputs; and, (iii) the AES block cipher, which performs the key expansion, and the message encryption and decryption tasks. The design of the interface module and the control unit varies according to the coupling style and communication interface that was chosen prior to the coprocessor deployment. FAC-V supports both the encryption and decryption operations, but since the decryption performs the same steps as the encryption (executed in a reverse way), this work only discusses and evaluates the encryption process.

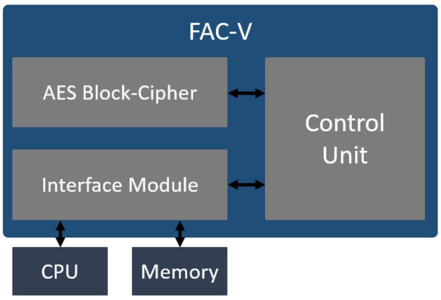


Figure 4. FAC-V Coprocessor Architecture.

AES Block Cipher

The AES block cipher core was designed to execute AES encryption with all the key sizes supported by the AES standard (128, 192, and 256 bits) [39]. The AES block cipher receives a block of plaintext data and outputs an encrypted data block of the same size.

Figure 5 illustrates the AES block cipher, which performs two main steps: (i) the key expansion, where the received key is expanded (*exp_key*) and sent to the *rounds encryption* module; and, (ii) the rounds encryption, which receives the expanded key from the key expansion, and the entire 128-bit *input_message*. The *cipher_sel* signal is used to select the encryption or decryption task, and the *start* bit is used to start the encryption/decryption process. Then, the *rounds encryption* module outputs the entire 128-bit *out_message* and the *ready* bit to signalize the completion of the selected process.

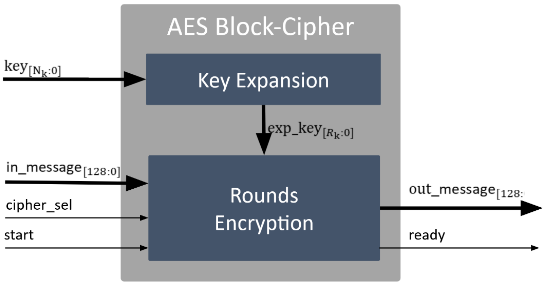
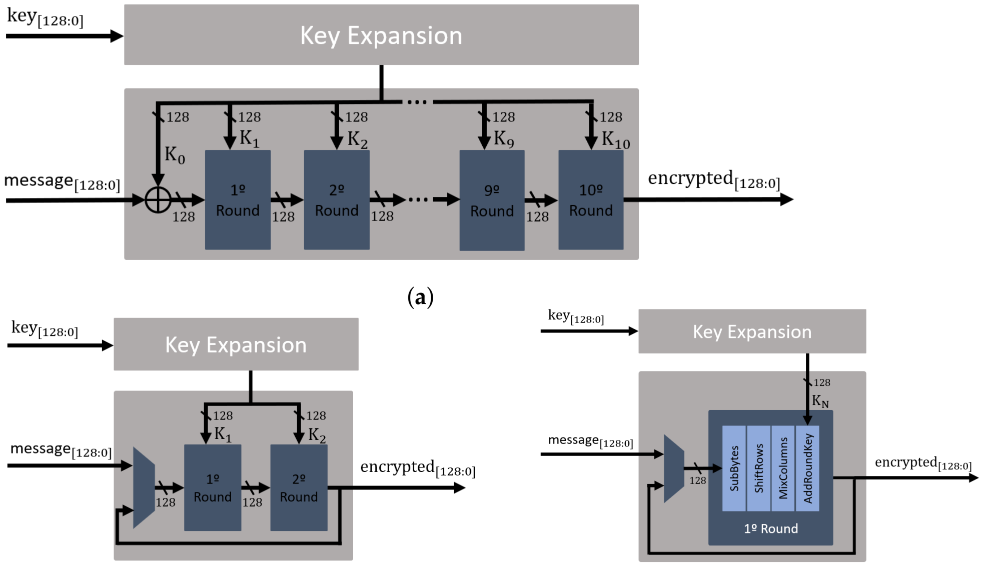


Figure 5. Schematic view of the block cipher.

To further evaluate the impact of the design choices, in terms of clock cycles per encryption, throughput and FPGA resources utilization, this main block was designed following three AES architecture approaches: (i) the fully unrolled architecture, as shown in Figure 6a, where all encryption rounds are performed in parallel; (ii) the partly unrolled architecture, as shown in Figure 6b, which is designed to perform two rounds in parallel, representing the lightest unrolled architecture; and (iii) the rolled architecture, as shown in Figure 6c, which results in a lighter approach than the previous ones. The architecture of the AES block cipher is a configurable parameter in the FAC-V, allowing the implementation of any of the three possible approaches.



(**a**)

(**b**) (**c**)

Figure 6. Block cipher structure for AES-128: (**a**) Fully unrolled. (**b**) Partly unrolled. (**c**) Rolled.

3.2. Tightly Coupled FAC-V

For the tightly coupled coprocessor, a set of RoCC instructions (RISC-V R-type) were created to interact with all hardware components, leveraging the *funct7* field to have unique arbitrary values that correspond to the functions detailed in Table 2, specifying the input and output of each function after translating the *rs1*, *rs2*, and *rd* fields of the RoCC instruction, as depicted in Listing 1. Therefore, by using the dedicated functions presented in Table 2, we implemented the following API functions to mimic the OS cryptographic libraries: the *FACV_init()* which initializes the AES accelerator by providing the key; the *FACV_encrypt()*; and *FACV_decrypt()*. To perform the encryption, a set of steps must be sequentially executed. If the message is delivered to the core through instructions, the *FACV_encrypt()* needs to send the message data and the message size, send a command to trigger the encryption, and read the result from the encryption operation. On the other hand, if the message is already allocated in memory, the *FACV_encrypt()* only needs to send the message size and the memory address of the message to encrypt.

Table 2. Software RoCC interface description.

Function (*funct7*)	Input1 (*rs1*)	Input2 (*rs2*)	Output (*rd*)	Description
Send Key	Key	Key	Null	Sends 2×32-bit of the AES key
Send Size	Size	Null	Null	Sends the message size
Send Message	Message	Message	Null	Sends 2×32-bit of the message
Send Addresses EN	Message address	Result address	Null	Sends the addresses, and sets load and encryption
Send Addresses DE	Message address	Result address	Null	Sends the addresses, and sets load and decryption
Start Encryption	1	Null	Null	Sets the start encryption flag on
Start Decryption	Null	1	Null	Sets the start decryption flag on
Read Result	Null	Null	Result	Reads one word of the result

Listing 1. API RoCC instruction format.

```
#define CUSTOM_0 0b0001011
#define ROCC_INSTRUCTION(rd , rs1 , rs2 , func7) \
__asm__ volatile (".insn r " STR(CUSTOM_0) ", " STR(0x7) ", " STR(func7) ", %0, %1, %2" \
        : "=r"(rd) \
        : "r"(rs1), "r"(rs2))
```

3.3. Loosely Coupled FAC-V

For the loosely coupled version of the FAC-V coprocessor, instead of dedicated instructions, the API defines the default register's addresses, as depicted in Table 3, and it provides the set of functions, as detailed in Table 4, that directly interact with the memory registers using the MMIO interface. For the encryption task, the *FACV_encrypt()* function firstly reads the *Status* register to check when the coprocessor is free to be used; when free, the message is written (in blocks of 32 bits); then, the function writes to the *cipher_sel* register to select the desired encryption or decryption process, which simultaneously signals the start of the operation. Finally, the *Status* register is read again until the encryption is ready and the result can be retrieved from memory.

Table 3. API MMIO registers default addresses.

Registers	Default Address
Status	0×2000
Key	0×2004
Message	0×2024
Cipher_sel	0×2034

Table 4. Software MMIO interface description.

Function	Input1	Input2	Ouput	Description
Read Status	Status Address	Null	Null	Reads the Status register for checking the ready signal
Write Key	Key Address	Key	Null	Writes 32 bits of the key in the specified memory address
Write Message	Message Address	Message	Null	Writes 32 bits of the message in the specified memory address
Read Message	Message Address	Null	Message	Reads 32 bits message from the specified memory address
Write Cipher_sel	Cipher_sel Address	Cipher_sel	Null	Writes to the Cipher_sel register, true for encryption and false for decryption.

4. FAC-V Evaluation

Regarding the experimental setup, the FAC-V was deployed and evaluated on an Arty A7-35T hardware platform, which features a Xilinx XC7A100TCSG324-1 FPGA core. The accelerator is connected to a SiFive E31 RISC-V processor (RV32-IMAC specification) operating at a clock speed of 65 MHz (it can go up to 72 MHz), which corresponds to the default clock frequency provided by the Rocket Core. Although higher clock speeds for the RISC-V core and the accelerator could be achieved, the required number of clock cycles to perform the encryption/decryption tasks is low. Therefore, to keep the hardware logic as simple as possible, the FAC-V uses the same clock frequency as the RISC-V core and the communication buses. Both the RISC-V core and the coprocessor were implemented using the SiFive Freedom E300 Arty FPGA Dev Kit and synthesized with Xilinx Vivado 2020.2. For the performance evaluation of an IoT OS, we used the RIOT OS version 2021.04 [17], and all binaries were generated with the RISC-V GNU Compiler Toolchain (version 8.3.0).

The evaluation of the FAC-V coprocessor comprises three main experiments: (1) the FPGA resources required to deploy the coprocessor in FPGA; (2) the latency evaluation of the cryptographic API functions; and (3) the impact caused by the FAC-V coprocessor in the OS performance, which is measured with the Thread-Metric RTOS Test Suite. Each experiment was performed in different configurations: (i) the native software implementation (provided by the OS) referred to as *SW*; (ii) the tightly coupled version of FAC-V without memory allocation, referred to as *RoCC*; (iii) the tightly coupled configuration with memory allocation, referred to as *RoCC Mem*; and (iv) the loosely coupled configuration, referred to as *MMIO*.

4.1. Hardware Resources

The FPGA resource utilization was retrieved with the Vivado post-implementation report, which is presented in Table 5 in terms of Look-up Tables (LUTs), Muxes, and Flip-Flops. For comparison purposes, the baseline configuration has only the Rocket core deployed in the FPGA, which requires 17903 LUTs, 714 Muxes, and 10161 Flip-Flops. For the Rolling architecture of the AES algorithm, this evaluation includes the three different versions of the FAC-V regarding the key size of the AES in use, i.e., 128, 192, or 256 bits. Additionally, it shows the resources increase when an Unrolled AES architecture is used. For the Rolled FACV-128, adding the *RoCC* configuration results in a resource utilization increase of 22.7% for LUTs, 24.1% for Muxes, and 18.3% for Flip-Flops. The *RoCC Mem* achieved a resource increase of 22.8% for LUTs, 23.5% for Muxes, and 21.3% for Flip-Flops. Concerning the *MMIO*, the resources increase is 23.3% for LUTs, 18.1% for Muxes, and 17.9% for Flip-Flops. While the *RoCC* and the *MMIO* configurations have similar resources utilization, the *RoCC Mem* configuration requires more hardware to be deployed. This is caused by the extra registers and control required to manage the memory sub-interface. For the Rolled FACV-192 and FACV-256, the increase in the key size resulted also in higher hardware resources. This is due to the extra calculations required in the key expansion task.

Table 5. Estimated hardware resources utilization.

		FACV-128			FACV-192			FACV-256			Unrolled FACV-128	
	Rocket Core	**RoCC**	**RoCC Mem**	**MMIO**	**RoCC**	**RoCC Mem**	**MMIO**	**RoCC**	**RoCC Mem**	**MMIO**	**RoCC 2-Rounds**	**RoCC 9-Rounds**
LUTs	17,903	+4056 +22.7%	+4086 +22.8%	+4174 +23.3%	+4328 +24.2%	+4464 +24.9%	+4432 +24.8%	+4824 +26.9%	+4923 +27.5%	+4913 +27.4%	+8845 +49.4%	+18,452 +103.1%
Muxes	714	+172 +24.1%	+168 +23.5%	+129 +18.1%	+301 +35.6%	+297 +35.1%	+258 +32.9%	+351 +49.2%	+332 +46.5%	+340 +47.6%	+1339 +187.5%	+6297 +881.9%
Flip-Flops	10,161	+1864 +18.3%	+2169 +21.3%	+1822 +17.9%	+2187 +21.5%	+2496 +24.6%	+2146 +21.1%	+2507 +24.7%	+2818 +27.7%	+2465 +24.3%	+3718 +36.6%	+4638 +45.6%
Logic Gates	181,955	+29,894 +16.43%	+32,621 +17.93%	+32,621 +17.93%	+33,269 +18.28%	+35,835 +19.69%	+28,867 +15.86%	+36,526 +20.07%	+39,710 +21.82%	+32,392 +17.80%	+197,924 +108.77%	+322,450 +177.21%

For the unrolled architecture, we have generated the hardware for the FACV-128. In a two-round architecture, the required hardware resulted in a resource increase of 49.4% for LUTs, 187.5% for Muxes, and 36.6% for Flip-Flops. For the nine-round architecture, the required resources increased by 103.1% for LUTs, 881.9% for Muxes, and 45.6% for Flip-Flops. Since the unrolled version of the FAC-V coprocessor required more resources than the ones available in the Arty platform, it was only possible to deploy the different versions of the FAC-V (FACV-128, FAC-V-192, and FACV-256) following the Rolled architecture.

Despite FAC-V targeting reconfigurable platforms, we also estimate the logic gates (calculated from the different FPGA resources utilization) required to potentially deploy our accelerator in an ASIC. However, in a real implementation, these values would vary according to the selected ASIC technology and hardware layout. From the results available in Table 5, deploying the Rocket Core without the FAC-V accelerator would require around 181,966 logic gates. Adding the accelerator increases the logic gate count by 16.43%, 17.93%, and 17.93% for the FACV-128 in the *RoCC*, *RoCC Mem*, and *MMIO* architectures, respectively. Using the FACV-192 configuration, the logic gate count is increased by 18.28% for the *RoCC* architecture, 19.69% for the *RoCC Mem*, and 15.86% for the *MMIO*. Finally, when the accelerator uses the FACV-256 configuration, the logic gate count increases in 20.07%, 21.82%, and 17.80% in the *RoCC*, *RoCC Mem*, and *MMIO* architectures, respectively. As expected, the required logic gate number increases as the AES key size also increases. Nonetheless, the architecture that presents better results in the FACV-128 configurations is the *RoCC* interface, being the *MMIO* the interface that gives less gate count increase in the FACV-192 and FACV-256. For the unrolled FACV-128, these numbers further increase to 108.77% for the *RoCC (two rounds)* and 177.21% for the *RoCC (nine rounds)*.

4.2. API Latency

The API latency is obtained through micro-benchmarks that count the clock cycles required by each cryptographic service performed by each developed implementation, both in the software and hardware configurations. This test covers all the secret key sizes for different message payloads, starting from a minimum of 16 bytes (one 128-bit message block) up to 80 bytes (five 128-bit message blocks).

4.2.1. AES Initialization

The AES initialization function initializes the AES Cipher with the secret key, and its latency results are depicted in Table 6. For the FACV-128 implementation, the software-based initialization (used as the baseline) requires 163 cycles to perform, while the *RoCC Mem* and *RoCC* configurations, which share the same initialization implementation, require a total of 34 clock cycles, representing a performance increase of 4.8× relative to the baseline. Regarding the *MMIO* configuration, it needs 83 clock cycles to execute, representing a performance increase of 1.9×. For the FACV-192 and FACV-256 implementations, due to the instructions sending bigger key sizes, the number of clock cycles required to compute the initialization is 183 and 244, respectively. These values are slightly higher than the software-only version (as expected), but when performed in hardware, better performance gains can be achieved, i.e., 29 clock cycles (4.7×) and 45 clock cycles (5.4×) for the FACV-192 and FACV-256, respectively, in both the *RoCC Mem* and *RoCC* configurations. Regarding the *MMIO* configuration, the FACV-192 required 113 clock cycles (1.6×) to execute, while the FACV-256 executed in 142 (1.7×) clock cycles. In all experiments, the standard deviation (SD) value is zero. This is explained by the small number of clock cycles required to perform the initialization task.

Table 6. *aes_init()*: AES initialization latency results.

		FACV-128	FACV-192	FACV-256
Clock Cycles	Software	163	183	244
	RoCC Mem	34	39	45
	RoCC	34	39	45
	MMIO	83	113	142
Standard Deviation	Software	0	0	0
	RoCC Mem	0	0	0
	RoCC	0	0	0
	MMIO	0	0	0
Performance Increase	RoCC Mem	4.8×	4.7×	5.4×
	RoCC	4.8×	4.7×	5.4×
	MMIO	1.9×	1.6×	1.7×

4.2.2. AES Encryption

Software Version: Table 7 shows the results of the AES encryption task for the different secret key sizes and message payloads of 16, 32, 48, 64, and 80 bytes (one to five 128-bit message blocks). These results are supported by Figure 7, which displays the latency results for the software version, and Figure 8, which corresponds to the hardware implementations. Regarding the software-only implementation, the FACV-128 required around 119,229 clock cycles (SD of 620 clock cycles) to handle a 16-byte message and 596,208 clock cycles (SD of 1355) for a message size of 80 bytes. These values increase when the key size also increases, e.g., for the FACV-192, the number of clock cycles required to encrypt a 16 and 80 bytes message is, respectively, 131,602 (SD of 701) and 659,334 (SD of 1090), while for the FACV-256, the number of clock cycles is 160,833 (SD of 659) for a message size of 16 bytes and 804,283 (SD of 1420) for an 80-byte message.

Table 7. *aes_encrypt()*: AES encryption latency results.

		FACV-128					FACV-192					FACV-256				
Size of Message		16 Bytes	32 Bytes	48 Bytes	64Bytes	80 Bytes	16 Bytes	32 Bytes	48 Bytes	64 Bytes	80 Bytes	16 Bytes	32 Bytes	48 Bytes	64 Bytes	80 Bytes
Clock Cyles	Software	119,229	238,444	357,718	476,841	596,208	131,602	263,497	394,708	526,963	659,334	160,833	321,743	482,443	643,405	804,283
	RoCC Mem	20	38	77	121	165	20	38	81	126	166	20	38	77	125	168
	RoCC	66	126	188	251	314	68	130	194	259	324	68	134	200	267	334
	MMIO	189	367	554	741	922	189	373	557	749	921	191	385	554	745	1013
Standard Deviation	Software	620	886	757	1024	1355	701	763	1104	1090	1606	659	1032	1121	1123	1420
	RoCC Mem	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	RoCC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	MMIO	0	0	0	1	1	0	0	0	0	1	0	7	1	0	3
Performance Increase	RoCC Mem	5961×	6275×	4646×	3941×	3613×	6580×	6934×	4873×	4182×	3972×	8042×	8467×	6265×	5147×	4787×
	RoCC	1807×	1892×	1903×	1900×	1899×	1935×	2027×	2035×	2035×	2035×	2365×	2401×	2412×	2410×	2408×
	MMIO	631×	650×	646×	644×	647×	696×	706×	709×	704×	716×	842×	836×	871×	864×	794×

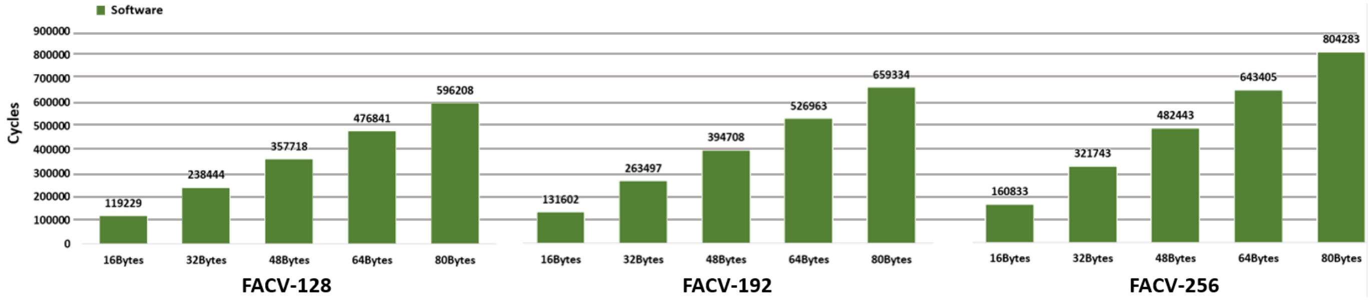


Figure 7. Software version AES encryption latency results.

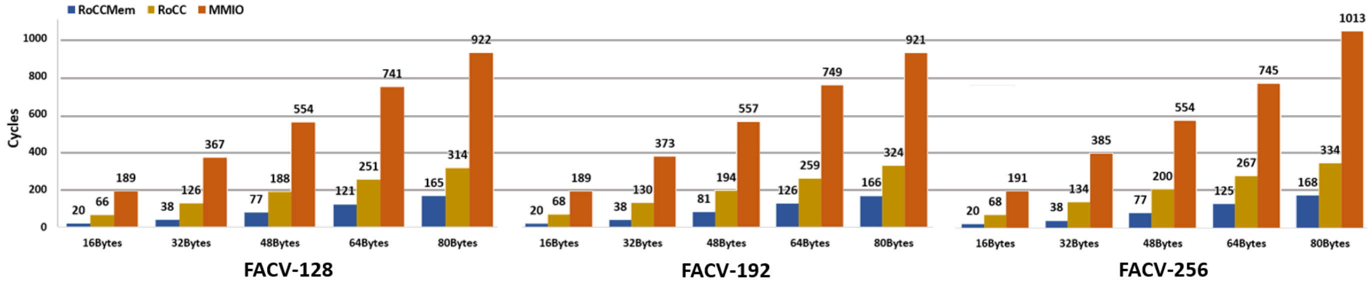


Figure 8. FAC-V AES encryption latency results.

RoCC Mem: When performing the encryption task with the FAC-V coprocessor, the number of clock cycles required to encrypt different message sizes (for the three values of the secret key) greatly decreases. These values are present in Table 7 and supported by Figure 8. Regarding the *RoCC Mem* implementation, the FACV-128 requires 20 clock cycles to execute the encryption of a message size of 16 bytes and 165 clock cycles when the message size is 80 bytes. When compared with the software-only version, it corresponds to a performance increase of 5961× and 3613×, respectively. In the FACV-192 and FACV-256, these remain the same when the message size is 16 bytes, i.e., 20 clock cycles, but they slightly increase to 166 and 168 clock cycles when the message size increases to 80 bytes. Despite presenting similar values when varying the message size, the performance gains in comparison to the software implementation are considerably higher, i.e., 6580× and 3972× for the FACV-192 and 8042× and 4787× for the FACV-256.

RoCC: The number of clock cycles required by the FACV-128 to encrypt a 16-byte and an 80-byte message is, respectively, 66 and 314, which corresponds to a performance increase of 1807× and 1899×. The FACV-192 requires 68 and 324 clock cycles to encrypt a message of 16 and 80 bytes, respectively, which corresponds to a performance increase of 6580× and 3972×. Regarding the FACV-256, these values are nearly the same as the FACV-192, corresponding to a performance gain of 1935× and 2035× for a message size of 16 and 80 bytes in the FACV-192 configuration and a performance increase of 2401× and 2408× for a respective message size of 16 and 80 bytes in the FACV-256 implementation. Overall, the performance gains for both the FACV-192 and the FACV-256 for the different message sizes increase when compared with the corresponding software version.

MMIO: Regarding the *MMIO* configuration, the FAC-128 requires 189 clock cycles to encrypt a 16-byte message and 922 clock cycles to encrypt a message with a payload of 80 bytes, corresponding to a performance gain of 631× and 647×, respectively. When using a key size of 192 bits, FACV-192, encrypting a message of 16 and 80 bytes requires 189 and 921 clock cycles, respectively, representing a performance gain of 696× and 716×. For the FACV-256, the number of clock cycles to encrypt a message of 16 and 80 bytes is, respectively, 385 and 1013, representing a performance increase of 842× and 794×. Again, comparative to the respective software version, the performance gains increase when the key size also increases.

Discussion: Regarding the software implementation, the latency increase results are directly related to the message and key sizes, which require more clock cycles to execute when a higher number of encryption rounds need to be performed. The different hardware approaches, from a macro perspective, provide nearly the same performance results for the different message sizes in the different sizes of the encryption key. However, the performance gains provided by *MMIO* and *RoCC* configurations follow a different behavior than the *RoCC Mem* implementation. While for the *MMIO* and *RoCC*, the performance gains keep increasing when the message size also increases, this is not always true for the *RoCC Mem* configuration. The main reason for this behavior is related to the required memory accesses used by this configuration, which is affected when the message size increases from 32 to 48 bytes. This may be caused by two different situations: (i) since the implemented RoCC memory sub-interface needs to wait for the "ready to use" memory validation signal, some unexpected delays can occur, which are mainly caused by concurrent system bus masters, which can increase the overall latency of the encryption task; and (ii) since there are more data blocks to process, the additional function calls may also cause extra bus contention during the prologue and epilogue execution. Considering the standard deviation in the hardware configurations, this value is zero or nearly zero, showing that the encryption task in the different versions of the FAC-V is fully deterministic. In the software setup, the standard deviation values are mainly related to the extensive multiple mathematical and arithmetic operations that need to be performed, resulting in more memory accesses and processing time, which is susceptible to change due to other software tasks being executed by the CPU in the OS thread scheduling, interrupts, etc. Lastly, it is possible to conclude that the tightly coupled versions of the FAC-V, i.e., *RoCC* and *RoCC Mem*, perform better than the loosely coupled approach deployed through the *MMIO* configuration.

4.3. OS Performance

To evaluate the impact of the accelerator on the OS performance, we used the Thread-Metric RTOS Test Suite with the RIOT OS [40]. This synthetic suite implements several benchmarks that stress a singular Real-Time OS (RTOS) service. After the RIOT initialization and the main thread is reached, each test creates the necessary OS threads and system configurations. The output of each test is the number of times each loop has been executed, wherein a higher loop count indicates better performance. The performance evaluation of RIOT included the following tests:

1. **Basic Processing**: A single thread performs mathematical operations in a loop.
2. **Cooperative Scheduling**: Five threads with the same priority execute concurrently, yielding in a loop.
3. **Preemptive Scheduling**: Five threads with increasing priorities, each resuming the next thread with a higher priority and suspending themselves in a loop.
4. **Interrupt Processing**: A single thread is interrupted each time it executes, being resumed afterwards.
5. **Interrupt Preemption Processing**: Two threads with different priorities, where one of them triggers an interrupt responsible for resuming the other suspended thread.
6. **Message Processing**: A thread sends a message to itself through a queue in a loop.
7. **Synchronization Processing**: A single thread gives and takes a semaphore in a loop.
8. **Memory Allocation**: A thread allocates and de-allocates memory blocks in a loop.

Regarding this evaluation, we have created the network topology depicted in Figure 9, which is composed of three nodes, two edge devices, and a gateway, recreating a real-world situation of two IoT devices communicating with each other and/or the Internet. In this setup, we have created a simple User Datagram Protocol (UDP) connection between a ready-to-use RIOT node based on a STM32f767ZI board (with an Arm Cortex-M7 32-bit RISC core operating at 216 MHz connected to a CC2520 device, an IEEE 802.15.4-compliant radio transceiver) and the Arty platform (containing the RISC-V core, the FAC-V coprocessor, and also a CC2520 radio). On both nodes, the RIOT OS uses the full network stack with the following configurations: IEEE 802.15.4 PHY and MAC layers; 6LoWPAN for the adaptation layer; UDP for the transport layer; IPv6, and a simple application that exchanges messages of 80 bytes (five blocks of 128 bits) each, encrypted with the AES-256, through a UDP socket. The client node keeps sending around 15 encrypted UDP messages per second. The server node receives the encrypted messages, recovers the original *plaintext*, validates and changes the content of the message, encrypts the message again, and sends it back to the UDP client. The Thread-Metric runs side by side with the UDP server, and it intends to evaluate the impact of the encryption and decryption tasks on the OS performance by resorting to the software version of the AES and all versions of the FAC-V coprocessor.

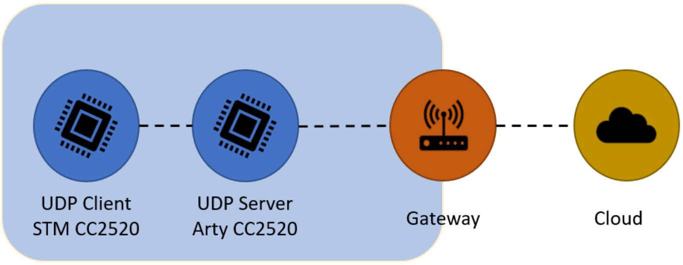


Figure 9. Network topology used in the Thread-Metric evaluation.

Table 8 summarizes the Thread-Metric results obtained in the following scenarios: (i) the *Baseline*, which corresponds to all tests running with no communication between the UDP server and the client; (ii) the *Echo*, where the Server receives and echoes the packets from the UDP client without data encryption. This test evaluates the impact of simply adding a communication link to the OS; and the *Echo* with all possible AES configurations, i.e., *(iii) Echo + AES Software, (iv) Echo + RoCC Mem, (v) Echo + RoCC, and (vi) Echo + MMIO*. The relative performance change for each Thread-Metric test according to each test scenario is summarized in Table 9.

Table 8. Thread-Metric benchmark results.

Version	Basic Processing	Cooperative Scheduling	Preemptive Scheduling	Interrupt Processing	Interrupt Preemptive	Message Processing	Synchronization Processing	Memory Allocation
(i) Baseline (no network)	73,012	4,049,986	1,552,968	5,746,523	3,051,415	5,480,340	5,676,620	2,810,406
(ii) Echo	38,994	2,145,623	869,367	3,212,352	1,707,203	3,013,826	3,180,201	1,489,664
(iii) Echo + SW	24,678	1,288,700	461,632	1,490,272	1,015,795	2,227,820	2,275,369	1,109,816
(iv) Echo + RoCC Mem	37,560	2,100,921	830,111	3,035,100	1,594,133	2,779,515	2,895,005	1,411,679
(v) Echo + RoCC	35,546	2,053,071	817,707	2,955,054	1,586,991	2,778,290	2,901,002	1,412,516
(vi) Echo + MMIO	37,060	2,137,461	806,495	3,066,915	1,566,676	2,849,630	2,849,586	1,437,321

Table 9. Relative performance decrease.

Version	Basic Processing	Cooperative Scheduling	Preemptive Scheduling	Interrupt Processing	Interrupt Preemptive	Message Processing	Synchronization Processing	Memory Allocation
(i) Baseline (no network)	−46.6%	−47.0%	−44.0%	−44.1%	−44.1%	−45.0%	−44.0%	−47.0%
(ii) Echo	0%	0%	0%	0%	0%	0%	0%	0%
(iii) Echo + SW	36.7%	39.9%	46.9%	53.6%	40.5%	26.1%	28.5%	25.5%
(iv) Echo + RoCC Mem	3.7%	2.1%	4.5%	5.5%	6.6%	7.8%	9.0%	5.2%
(v) Echo + RoCC	8.8%	4.3%	5.9%	8.0%	7.0%	7.8%	8.8%	5.2%
(vi) Echo + MMIO	5.0%	0.4%	7.2%	4.5%	8.2%	5.4%	10.4%	3.5%

When the network is being used, i.e., packets are being exchanged between the UDP server and client, the OS performance decreases between 44% and 47%, corresponding to the Preemptive Scheduling and the Memory Allocation tests, respectively, showing that the network utilization has a huge impact on the overall performance of the device. The performance further decreases when the encryption is enabled and used in the communication. To evaluate the impact of enabling the security features over the network data exchange, the *(ii) Echo* experiment is now assumed as the baseline (we assume that AES is mainly needed when communications are required). When the data exchange uses the AES services in software (native RIOT drivers and API), the performance decreases between 26.1% and 53.6%, corresponding to the Message and the Interrupt Processing tests, respectively.

Basic Processing: In this test, the OS performance decreased by 36.7% when using the software libraries of the AES algorithm. However, when resorting to the FAC-V coprocessor, this value can be reduced to 3.7% when using the *RoCC Mem* configuration.

Cooperative and Preemptive Scheduling: Regarding the Cooperative Scheduling and the Preemptive Scheduling tests, the Cooperative achieved lower performance degradation than the Preemptive, which is mainly explained by the microkernel architecture used by RIOT. While in the Cooperative Scheduling test, the executing task yields itself, in the Preemptive Scheduling test, the executing task resumes to a different one, which involves more system calls and more processing from the scheduler due to the tasks having different priorities. When using the pure software version of the AES algorithm, the performance decrease is around 39.9% for the Cooperative Scheduling and 46.9% for the Preemptive Scheduling. However, resorting to the FAC-V coprocessor, the performance decrease can be reduced to 0.4% for the Cooperative test using the *MMIO* interface and 4.5% for the Preemptive test in the *RoCC Mem* configuration.

Interrupt Processing and Preemptive: When resorting to the AES in software, the Interrupt Processing test shows a performance decrease of 53.6%, while the Interrupt Preemptive achieved a performance decrease of 40.5%. This is mainly explained by the fact that the Interrupt Processing test, since it involves more hardware interrupts, requires more system calls and more processing time from the scheduler. When resorting to the FAC-V, the performance decrease of the Interrupt Processing is around 4.5% in the *MMIO* configuration and 6.6% in the Interrupt Preemptive when the *RoCC Mem* configuration is used.

Message Processing, Synchronization Processing, and Memory Allocation: For these tests, the results show the lowest performance decrease. The Message Processing has a 26.1% performance decrease, while the Synchronization Processing and the Memory Allocation show performance decreases of 28.5% and 25.5%, respectively. This shows that the memory management system implemented in RIOT has less impact on the overall system's performance in contrast to the tests that require preemption and interrupts. For the hardware configurations, the lowest performance decreases were 5.4%for Message Processing, 8.8% for Synchronization Processing, and 3.5% for Memory Allocation.

4.4. FAC-V Power Estimation

To estimate the power consumption of the RISC-V core along with the FAC-V accelerator with different AES key size configurations, we used the Power Analysis tools included in the Vivado Design Suite. The tool was run in vectorless mode with the default settings and power optimizations disabled and with the platform constraints for the Arty A7-100 board. The report includes the dynamic power consumption, which is determined by the switching activity of clocks and datapaths, and the static power consumption, which represents the minimum power consumption required to operate the hardware blocks. Table 10 summarizes the gathered results for the following configurations: (1) the RISC-V core only (Rocket Core without RoCC/ MMIO interfaces; and (2) the RISC-V core with the FAC-V accelerator in different AES configurations (FACV-128, FACV-192, and FACV-256) for the three different coprocessor interfaces (RoCC, RoCC and memory, and MMIO).

Table 10. FAC-V Power Estimation.

		FACV-128			FACV-192			FACV-256		
	Rocket Core	**+RoCC**	**+RoCC Mem**	**+MMIO**	+RoCC	**+RoCC Mem**	**+MMIO**	**+RoCC**	**+RoCC Mem**	**+MMIO**
Static Power (W)	0.099	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%	0.100 +1.0%
Dynamic Power (W)	0.196	0.218 +11.2%	0.220 +12.3%	0.221 +12.7%	0.218 +11.2%	0.217 +10.7%	0.215 +9.7%	0.232 +18.4%	0.231 +17.9%	0.217 +10.7%
Total Power (W)	0.295	0.318 +7.8%	0.320 +8.5%	0.321 +8.8%	0.318 +7.8%	0.317 +7.5%	0.315 +6.8%	0.332 +12.5%	0.331 +12.2%	0.317 +7.5%

Deploying the RISC-V Core in the hardware platform without the RoCC and MMIO interface corresponds to a total power dissipation of 0.295 W. Adding the FACV with the RoCC interface increases the power consumption to 0.318 W when using the FACV-128 and FACV-192 configurations, and 0.332 W when the FACV-256 is deployed. When combining the RoCC interface with memory, the power dissipation is around 0.320 W for the FACV-128, 0.317 W for the FACV-192, and 0.331 W for the FACV-256. Deploying the coprocessor with the MMIO interface corresponds to a power consumption of 0.321 W for the FACV-128, 0.315 W for the FACV-192, and 0.317 W for the FACV-256.

In the FACV-128, the power consumption increases when changing the interface from the RoCC to RoCC Mem as well as when changing it from RoCC Mem to MMIO. This power consumption increase behaves in the opposite direction for the FACV-192 and FACV-256; i.e., MMIO is the most power consuming interface, and the RoCC is the interface that achieves less power dissipation. We believe this is mainly caused by different hardware resources required to deploy the different FAC-V configurations, which can be optimized during the Implementation phase in the Vivado tool. However, as expected, it is observed that increasing the key size in the AES configuration, e.g., from the FACV-128 to FACV-256, provides high power consumption results. Comparing with the RISC-V core, adding these AES configurations only causes a power consumption increase between 6.8% and 12.5%. Thus, the impact of each solution on the overall energy consumption is mainly dictated by the time each configuration takes to process the encryption of different message sizes.

Table 11 depicts the energy estimated to process a message size of 16 and 80 bytes for the different configurations of the AES when using different coprocessor interfaces. As expected, and following the trend previously found (and discussed) in Table 7, performing the encryption task in hardware causes less energy dissipation. This is directly related to the required processing time to encrypt a message in the different coprocessor interfaces for the different AES key sizes. Hence, and according to the message size, the RoCC Mem interface is the one that requires less energy consumption, being the MMIO the interface that dissipates more energy during the AES encryption task.

Table 11. AES encryption energy estimation.

		FACV-128		FACV-192		FACV-256	
Size of Message		**16 Bytes**	**80 Bytes**	**16 Bytes**	**80 Bytes**	**16 Bytes**	**80 Bytes**
Energy/ Message (µJ)	**Software**	541.12	2705.87	597.27	2992.36	729.93	3650.21
	RoCC Mem	0.10	0.81	0.10	0.81	0.10	0.86
	RoCC	0.32	1.55	0.33	1.58	0.35	1.70
	MMIO	0.93	4.55	0.93	4.46	0.93	4.94

4.5. Discussion

The extensive evaluation performed on the FAC-V processor has shown the huge benefits of deploying an AES accelerator in hardware, following both the loosely and the tightly coupled approaches, of which the latter coupling style provided better performance and power consumption results. However, both versions provide similar hardware costs for the implementation of the Unrolled AES architecture. The results show that in the

RoCC and MMIO configurations, exchanging data with the CPU imposes large overhead latencies, which were mainly due to contention problems in memory and system buses. This triggered the exploration of the memory-based tightly coupled version of FAC-V, the RoCC Mem configuration, which resulted in the best approach when compared to the other two. Extending the loosely coupled version in the same way would require the implementation of a DMA device whose implementation is currently under investigation.

When comparing FAC-V with other state-of-the-art solutions, as shown in Table 12, several trade-offs must be taken into good consideration before choosing and deploying the approach that best suits the final application. Regarding the AES architecture, the best performance is achieved when a fully or partially unrolled architecture is deployed. This strategy is mainly adopted by solutions that are deployed in ASIC or that support more powerful FPGA platforms [32,36,38]. However, this feature comes at a great cost in terms of hardware resources; for the FAC-V deployment and target platform, the required resources were above those available on the Arty board. We tested the Fully Unrolled architecture in simulation, showing that we could also achieve 1 clock cycle per encryption, while the Rolled architecture requires 10 clock cycles in the encryption process. Despite being 10x slower, the performance gains compared with the software are already about hundreds of magnitude better, as previously shown in Table 9. Thus, at this level, and since this work targets reconfigurable resource-constrained IoT devices, using the Unrolled architecture would not bring much more benefits in terms of performance but would carry extremely high hardware costs.

Table 12. Comparison with FAC-V AES Implementation.

Work	Year	Technology	AES Architecture	Frequency (MHz)	Cycles/ Encryption *	Throughput *
Agwa et al. [34]	2017	ASIC (Loosely)	Rolling	666	-	2.601 Gbps
Bui et al. [35]	2017	ASIC (Loosely)	Rolling	10	44	28 Mbps
Lu et al. [30]	2018	ASIC (-)	Rolling	50	213	30.05 Mbps
Banerjee et al. [31]	2019	ASIC (Loosely)	Rolling	16	11	-
Al-Gailani et al. [38]	2019	FPGA (-)	Unrolling	158	1	20.3 Gbps
Shahbazi et al. [36]	2020	FPGA (-)	Unrolling	622.4	1	79.7 Gbps
Marshall et al. [32]	2020	ASIC (Tightly)	Rolling and Unrolling	-	18–30	-
Pan et al. [33]	2021	FPGA & ASIC (Tightly)	Rolling	100	11–19	471–695 Mbps
FAC-V	**2022**	**FPGA (Loosely and Tightly)**	**Fully Unrolled and Partly Unrolled and Rolled**	**65**	**1** **5** **10**	**8.32 Gbps** **1.66 Gbps** **832 Mbps**

* Excluding the cost of communicating with peripherals.

Concerning the technology used to deploy the accelerator, FAC-V is the only solution that targets FPGA-based low-end IoT devices deployed both in the tightly and loosely coupled approaches. In addition to the deployment of the Rocket Core without any coprocessor, the Unrolled architecture of the FACV-256 version needs up to 47.6% more of the available hardware resources, while the Unrolled FACV-128 with two rounds and using the RoCC interface would add up to 187% more. However, despite operating at a lower clock frequency (65 MHz), the FAC-V achieves a higher throughput than other rolled architectures. In terms of hardware resources used, it was not possible to make an accurate comparison since the state-of-the-art implementations present the values in ASIC Gate Count, which cannot be directly mapped to FPGA cells (LUTs, Flip-Flops, and Muxes) in Xilinx FPGAs.

5. Conclusions

This article presents FAC-V, a hardware coprocessor connected to an RISC-V core that implements the AES algorithm in FPGA. The FAC-V hardware supports AES key sizes of

128, 192, and 256 bits, and by taking advantage of the RISC-V ISA, the coprocessor can be deployed following a tightly and a loosely coupled approach, using, respectively, the RoCC and the TileLink communication interfaces. Both coupling styles provided good performance results when compared with the pure software version of the AES algorithm present in the RIOT OS. For instance, the FACV-256 can achieve a performance improvement around 8000× when processing the encryption of 16-byte messages with the RoCC Mem interface, at the energy cost of around 0.10 µJ. The benefits of the coprocessor are further noticed in the overall OS performance, which can suffer a degradation of up to 53.6% in the interruption services when the network is continuously operating and the AES algorithm is being used to secure data exchanged with other IoT devices. With the FAC-V, the OS performance only decreases, for the same OS services, up to 8.2%. Hereafter, the FAC-V can be further improved to support DMA devices with the MMIO-based TileLink interface, which would mitigate the bus contention problems found in the current implementation.

Author Contributions: Conceptualization, T.G. and S.P.; methodology, T.G. and M.S.; software, P.S.; validation, P.S. and T.G.; formal analysis, P.S., M.S. and T.G.; investigation, P.S. and T.G.; resources, T.G., S.P. and M.E.; data curation, P.S. and T.G.; writing—original draft preparation, P.S. and T.G.; writing—review and editing, T.G., M.E. and S.P.; visualization, P.S. and T.G.; supervision, M.E., S.P. and T.G.; project administration, T.G. and S.P.; funding acquisition, S.P. and T.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by FCT—*Fundação para a Ciência e Tecnologia* within the R&D Units Project Scope UIDB/00319/2020 and Grant SFRH/BD/146678/2019.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Oliveira, D.; Costa, M.; Pinto, S.; Gomes, T. The Future of Low-End Motes in the Internet of Things: A Prospective Paper. *Electronics* **2020**, *9*, 111. https://doi.org/10.3390/electronics9010111.
2. Sundmaeker, H.; Guillemin, P.; Friess, P.; Woelfflé, S. Vision and Challenges for Realizing the Internet of Things. *Clust. Eur. Res. Proj. Internet Things EU Commision* **2010**, *3*, 34–36. https://doi.org/10.2759/26127.
3. Perera, C.; Liu, C.H.; Chen, M. A Survey on Internet of Things From Industrial Market Perspective. *IEEE Access* **2014**, *2*, 1660–1679. https://doi.org/10.1109/ACCESS.2015.2389854.
4. Yu, W.; Liang, F.; He, X.; Hatcher, W.G.; Lu, C.; Lin, J.; Yang, X. A Survey on the Edge Computing for the Internet of Things. *IEEE Access* **2018**, *6*, 6900–6919. https://doi.org/10.1109/ACCESS.2017.2778504.
5. Elnawawy, M.; Farhan, A.; Nabulsi, A.; Al-Ali, A.; Sagahyroon, A. Role of FPGA in Internet of Things Applications. In Proceedings of the IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Ajman, United Arab Emirates, 10–12 December 2019. https://doi.org/10.1109/ISSPIT47144.2019.9001747.
6. Valdés, M.; Rodriguez-Andina, J.; Manic, M. The Internet of Things: The Role of Reconfigurable Platforms. *IEEE Ind. Electron. Mag.* **2017**, *11*, 6–19. https://doi.org/10.1109/MIE.2017.2724579.
7. Waterman, A.S. *Design of the RISC-V Instruction Set Architecture*; University of California: Berkeley, CA, USA, 2016.
8. Waterman, A.; Lee, Y.; Patterson, D.A.; Asanovi, K. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA*, version 2.0; Technical Report; California Univ Berkeley Dept of Electrical Engineering and Computer Sciences: Berkeley, CA, USA, 2014.
9. Asanović, K.; Patterson, D.A. *Instruction Sets Should Be Free: The Case for RISC-V*; Tech. Rep. UCB/EECS-2014-146; EECS Department, University of California: Berkeley, CA, USA, 2014.
10. Azad, Z.; Yang, G.; Agrawal, R.; Petrisko, D.; Taylor, M.; Joshi, A. RACE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Computation. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, Boston, MA, USA, 1–3 August 2022; Association for Computing Machinery: New York, NY, USA, 2022. https://doi.org/10.1145/3531437.3539725.
11. Costa, M.; Costa, D.; Gomes, T.; Pinto, S. Shifting Capsule Networks from the Cloud to the Deep Edge. *arXiv* **2022**, arXiv:2110.02911. https://doi.org/10.1145/3544562.
12. Wu, N.; Jiang, T.; Zhang, L.; Zhou, F.; Ge, F. A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set. *Electronics* **2020**, *9*, 1005. https://doi.org/10.3390/electronics9061005.
13. De, A.; Basu, A.; Ghosh, S.; Jaeger, T. Hardware Assisted Buffer Protection Mechanisms for Embedded RISC-V. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 4453–4465. https://doi.org/10.1109/TCAD.2020.2984407.

14. Silva, M.; Gomes, T.; Pinto, S. Agnostic Hardware-Accelerated Operating System for Low-End IoT. In Proceedings of the 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Taipei, Taiwan, 23–25 August 2022; pp. 21–30. https://doi.org/10.1109/RTCSA55878.2022.00009.
15. Asanović, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, P.; Hauser, J.; Izraelevitz, A.M.; et al. *The Rocket Chip Generator*; EECS Department, University of California: Berkeley, CA, USA, 2016; UCB/EECS-2016-17. Available online: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (accessed on 4 September 2022).
16. Oikonomou, G.; Duquennoy, S.; Elsts, A.; Eriksson, J.; Tanaka, Y.; Tsiftes, N. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* **2022**, *18*, 101089. https://doi.org/10.1016/j.softx.2022.101089.
17. Baccelli, E.; Hahm, O.; Günes, M.; Wählisch, M.; Schmidt, T.C. RIOT OS: Towards an OS for the Internet of Things. In Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Turin, Italy, 14–19 April 2013; pp. 79–80. https://doi.org/10.1109/INFCOMW.2013.6970748.
18. Hahm, O.; Baccelli, E.; Petersen, H.; Tsiftes, N. Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet Things J.* **2016**, *3*, 720–734. https://doi.org/10.1109/JIOT.2015.2505901.
19. Silva, M.; Cerdeira, D.; Pinto, S.; Gomes, T. Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking. *IEEE Internet Things J.* **2019**, *6*, 10375–10383. https://doi.org/10.1109/JIOT.2019.2939008.
20. Fritzmann, T.; Sigl, G.; Sepúlveda, J. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 239–280. https://doi.org/10.13154/tches.v2020.i4.239-280.
21. Surendran, S.; Nassef, A.; Beheshti, B.D. A survey of cryptographic algorithms for IoT devices. In Proceedings of the 2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT), Farmingdale, NY, USA, 4 May 2018; pp. 1–8. https://doi.org/10.1109/LISAT.2018.8378034.
22. Henriques, M.S.; Vernekar, N.K. Using symmetric and asymmetric cryptography to secure communication between devices in IoT. In Proceedings of the 2017 International Conference on IoT and Application (ICIOT), Nagapattinam, India, 19–20 May 2017; pp. 1–4. https://doi.org/10.1109/ICIOTA.2017.8073643.
23. Goyal, T.K.; Sahula, V. Lightweight security algorithm for low power IoT devices. In Proceedings of the 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Jaipur, India, 21–24 September 2016; pp. 1725–1729. https://doi.org/10.1109/ICACCI.2016.7732296.
24. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. Chisel: Constructing hardware in a Scala embedded language. In Proceedings of the DAC Design Automation Conference 2012, San Francisco, CA, USA, 3–7 June 2012; pp. 1212–1221. https://doi.org/10.1145/2228360.2228584.
25. Cook, H.; Terpstra, W.; Lee, Y. Diplomatic design patterns: A TileLink case study. In Proceedings of the 1st Workshop on Computer Architecture Research with RISC-V, Boston, MA, USA, 14 October 2017.
26. Canright, D.; Osvik, D.A. A More Compact AES. In *Proceedings of the Selected Areas in Cryptography*; Jacobson, M.J., Rijmen, V., Safavi-Naini, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 157–169.
27. Singh, A.; Prasad, A.; Talwar, Y. *Compact and Secure S-Box Implementations of AES—A Review*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 857–871. https://doi.org/10.1007/978-981-13-8406-6_80.
28. Dhede, O.S.; Shah, S.K. A review: Hardware Implementation of AES using minimal resources on FPGA. In Proceedings of the 2015 International Conference on Pervasive Computing (ICPC), Pune, India, 8–10 January 2015; pp. 1–3. https://doi.org/10.1109/PERVASIVE.2015.7087187.
29. Mohurle, M.; Panchbhai, V.V. Review on realization of AES encryption and decryption with power and area optimization. In Proceedings of the 2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES), Delhi, India, 4–6 July 2016; pp. 1–3. https://doi.org/10.1109/ICPEICES.2016.7853276.
30. Lu, M.; Fan, A.; Xu, J.; Shan, W. A Compact, Lightweight and Low-Cost 8-Bit Datapath AES Circuit for IoT Applications in 28nm CMOS. In Proceedings of the 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), New York, NY, USA, 1–3 August 2018; pp. 1464–1469. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00204.
31. Banerjee, U.; Wright, A.; Juvekar, C.; Waller, M.; Arvind.; Chandrakasan, A.P. An Energy-Efficient Reconfigurable DTLS Cryptographic Engine for Securing Internet-of-Things Applications. *IEEE J. Solid-State Circuits* **2019**, *54*, 2339–2352. https://doi.org/10.1109/JSSC.2019.2915203.
32. Marshall, B.; Newell, G.R.; Page, D.; Saarinen, M.J.O.; Wolf, C. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2021*, 109–136.
33. Pan, L.; Tu, G.; Liu, S.; Cai, Z.; Xiong, X. A Lightweight AES Coprocessor Based on RISC-V Custom Instructions. *Secur. Commun. Netw.* **2021**, *2021*, 9355123. https://doi.org/10.1155/2021/9355123.
34. Agwa, S.; Yahya, E.; Ismail, Y. Power efficient AES core for IoT constrained devices implemented in 130nm CMOS. In Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, USA, 28–31 May 2017; pp. 1–4. https://doi.org/10.1109/ISCAS.2017.8050361.
35. Bui, D.H.; Puschini, D.; Bacles-Min, S.; Beigné, E.; Tran, X.T. AES Datapath Optimization Strategies for Low-Power Low-Energy Multisecurity-Level Internet-of-Things Applications. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 3281–3290. https://doi.org/10.1109/TVLSI.2017.2716386.

36. Shahbazi, K.; Ko, S.B. High throughput and area-efficient FPGA implementation of AES for high-traffic applications. *IET Comput. Digit. Tech.* **2020**, *14*, 344–352. https://doi.org/10.1049/iet-cdt.2019.0179.
37. Zgheib, A.; Potin, O.; Rigaud, J.B.; Dutertre, J.M. Extending a RISC-V core with an AES hardware accelerator to meet IOT constraints. In Proceedings of the SMACD/PRIME 2021; International Conference on SMACD and 16th Conference on PRIME, Online, 19–22 July 2021; pp. 1–4.
38. Al-Gailani, M.F.; Al-Khafaji, A.Q. Loop Unrolling Implementation of an AES Algorithm Using Xilinx System Generator. *Iraqi J. Inf. Commun. Technol.* **2019**, *2*, 38–45. https://doi.org/10.31987/ijict.2.3.85.
39. Dworkin, M. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*; National Inst of Standards and Technology (NIST), Computer Security Div.: Gaithersburg, MD, USA, 2001. https://doi.org/10.6028/NIST.SP.800-38A.
40. Microsoft Corporation. Azure RTOS ThreadX. 2022. Available online: https://github.com/azure-rtos/threadx (accessed on 4 September 2022).