International Journal of Computers and Informatics (IJCI)
Vol. (5), No. (11)
IJCI
November 2026
المجلة الدولية للحاسبات والمعلوماتية
الإصدار (5)، العدد (11)

Ascon-AEAD128 Vs AES-GCM and ChaCha20-Poly1305 on Arm Cortex-M Microcontrollers: An Emulation-Based Evaluation of Speed, Memory and Energy for IoT Nodes

Shaymaa Alhayali
Computer Center, University of Mosul, Iraq
shymaa_alhyali@uomosul.edu.iq

Abstract

In 2025, NIST standardized Ascon-AEAD128 as the authenticated encryption scheme for constrained devices, but engineers of Internet of Things (IoT) nodes still lack a consistent comparison with the schemes already available in common libraries. This study compares Ascon-AEAD128 with AES-128-GCM and ChaCha20-Poly1305 on the Arm Cortex-M4 and Cortex-M0+. Nine public, unmodified implementations from the Ascon designers' package, Mbed TLS and Monocypher were compiled into 30 bare-metal firmware images at two optimization levels. The images were executed in an instruction-accurate emulator, which gave exact instruction counts, cycle estimates from the published instruction timing of each core, flash, RAM and stack footprint, and estimated energy. Every image was first verified against official test vectors or an independent library. For a sensor packet with a 32-byte payload and a 16-byte header, the fastest Ascon implementation needed 4,805 cycles on the Cortex-M4, 29% fewer than ChaCha20-Poly1305 and 68% fewer than AES-GCM, with an estimated 1.44 µJ against 2.04 and 4.50 µJ. For 1-KB messages, Ascon and ChaCha20-Poly1305 were equivalent (about 40 cycles per byte), while software AES-GCM was three to six times slower and needed the largest code and RAM. The choice of implementation changed the cost of Ascon by up to 2.3 times and its code size by 27 times. All implementations had data-independent instruction counts and rejected forged packets. Ascon-AEAD128 with a core-specific implementation is recommended as the default for IoT nodes without an AES accelerator.

Keywords: Ascon, Authenticated Encryption, Lightweight Cryptography, Internet of Things, ARM Cortex-M, Embedded Systems.

1. Introduction

Most nodes of the Internet of Things (IoT) are small microcontrollers that read a sensor, build a short packet and send it over a low-power radio. These packets often carry data that must not be altered or read by others, such as meter readings, alarm states or actuator commands. Authenticated encryption with associated data (AEAD) protects both the payload and its header in one operation: it encrypts the payload, and it produces a tag that lets the receiver reject any packet that was modified in transit. On a microcontroller the AEAD algorithm competes for flash memory, RAM, processor time and battery energy with the application itself, so its cost is a design decision and not a detail.

For many years the default AEAD choices were AES in Galois/Counter Mode (AES-GCM) (Dworkin, 2007; National Institute of Standards and Technology [NIST], 2023) and ChaCha20-Poly1305 (Nir & Langley, 2018). In August 2025 NIST published Special Publication 800-232, which standardizes Ascon-AEAD128 as the lightweight AEAD for constrained devices (Sönmez Turan et al., 2025). Ascon was selected at the end of a multi-year public competition in which software and hardware efficiency on small devices was an explicit criterion (Sönmez Turan et al., 2023). The standard gives engineers of IoT nodes a new option, and with it a practical question: on a given microcontroller, is Ascon actually cheaper than the algorithms that are already available in common libraries, and by how much?

Significance and problem statement:

Published comparisons answer this question only in part. Many studies measure older Ascon versions or the reference code on 8-bit Arduino boards (Radhakrishnan et al., 2024; Sarasa Laborda et al., 2025), or compare ciphers without authentication, or exclude AES because the chip has a hardware accelerator (Sorescu et al., 2025). Others report a single message length, mix different key and nonce sizes without comment, or omit the memory footprint and the cost of key setup. A recent protocol for the comparison of Ascon-AEAD128, AES-128-GCM and ChaCha20-Poly1305 on IoT nodes explicitly leaves the measurements to future work (Atul et al., 2026). In addition, the choice among several implementations of the same algorithm, for example reference C code, portable optimized C or assembly for one core, can change the result as much as the choice of algorithm. An engineer who reads that “Ascon is faster than AES” therefore does not know whether this holds for the core, the library and the packet size of his own design.

Objectives and contributions:

This study compares the three AEAD families on the two most common IoT cores, the Arm Cortex-M4 and the Cortex-M0+, with a fully reproducible method. Its contributions are:

1. A benchmark of nine AEAD implementations, built from public, unmodified source code: five Ascon-AEAD128 implementations from the designers’ reference package, AES-128-GCM with two table options from Mbed TLS, and ChaCha20-Poly1305 from Mbed TLS and from Monocypher. They were compiled for the Cortex-M4 and, where the code supports it, the Cortex-M0+, each at two optimization levels, which gives 30 firmware images.
2. A measurement method based on instruction-accurate emulation of the actual firmware binaries. It gives exact instruction counts for key setup, encryption and decryption, a cycle estimates from the published instruction timing of each core, the peak stack depth, and the flash and static RAM footprint. Every output was checked against official test vectors or an independent reference library.
3. Results for message lengths from 0 to 1024 bytes, an IoT packet scenario, a forgery (tampered tag) test, a check of data-independent execution and a sensitivity analysis of the cycle model, followed

by practical recommendations for the choice of algorithm and implementation.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the implementations, the build, the emulation and the metrics. Section 4 presents and discusses the results, and Section 5 gives the conclusions and recommendations.

2. Literature Review

Lightweight cryptography has been studied for more than a decade. Surveys list dozens of lightweight block ciphers designed for small hardware and software footprints (Hatzivasilis et al., 2018) and compare their cost and security for IoT devices (Thakor et al., 2021). Most of these ciphers provide only confidentiality, so a separate mode and message authentication code are needed to protect a packet. The NIST lightweight cryptography project therefore asked for complete AEAD schemes. Its final report describes how the ten finalists were benchmarked in software and hardware and why Ascon was chosen (Sönmez Turan et al., 2023). Ascon is built on a 320-bit permutation that uses only bitwise operations and rotations; it absorbs 128 bits of data per permutation call in Ascon-AEAD128 and needs no tables (Dobraunig et al., 2021). SP 800-232 fixes the final parameters, including a 128-bit key, a 128-bit nonce and a 128-bit tag, and changes some details of the earlier version 1.2, so that older test vectors and implementations do not match the standard (Sönmez Turan et al., 2025).

Software benchmarks of the LWC candidates on microcontrollers were collected during the competition. Renner et al. (2020) built a hardware-in-the-loop suite that measured the speed and memory of the candidates on several 8-bit and 32-bit boards, and showed that the implementation and the compiler setting change the ranking. Fotovvat et al. (2021) compared lightweight algorithms on sensor-node boards in terms of execution time, memory and energy. Later studies focused on Ascon and on small boards. Radhakrishnan et al. (2024) compared AES-128, SPECK and Ascon on two AVR-based Arduino boards and found AES faster in execution time and Ascon the most economical in RAM. Sarasa Laborda et al. (2025) measured the Ascon v1.2 reference code on four Arduino boards from 8-bit AVR to 32-bit ESP32. Sorescu et al. (2025) evaluated Ascon and several stream ciphers on a Cortex-M33 board in a Bluetooth mesh scenario, but excluded AES because the chip has an AES accelerator. Ascon is also used inside larger constructions; Shehzadi et al. (2025), for example, replaced the hash and XOF functions of a post-quantum key encapsulation with Ascon on IoT devices. Atul et al. (2026) proposed a detailed benchmark protocol for Ascon-AEAD128, AES-128-GCM and ChaCha20-Poly1305, with fixed workloads and acceptance criteria, but reported no measurements.

The two established algorithms have been optimized for the same class of processors. Schwabe and Stoffelen (2017) gave table-based and bitsliced AES implementations for the Cortex-M3 and M4 and discussed the cost of resistance to timing attacks. Table-based AES is fast on processors with large caches, but its memory accesses depend on the key and the data, which enables cache-timing attacks on processors with caches (Osvik et al., 2006). The GHASH part of GCM is a multiplication in a binary field, and without a carry-less multiply instruction it is usually computed with precomputed

tables, which costs memory and time (Käsper & Schwabe, 2009). ChaCha20 uses only 32-bit additions, rotations and exclusive-OR, and the Poly1305 authenticator needs 32×32-bit multiplications (Bernstein, 2005; Nir & Langley, 2018), which the Cortex-M4 executes in one cycle.

Three gaps follow from this review. First, few studies compare the standardized Ascon-AEAD128 (not version 1.2) with both AES-GCM and ChaCha20-Poly1305 on the same core, with authentication included in every case. Second, results are usually given for one implementation of each algorithm and one message length, although both the implementation and the packet size change the ranking. Third, the memory footprint, the key-setup cost, the behavior when a forged packet is received, and the timing behavior are seldom reported together with the speed. The present study addresses these gaps.

3. Methodology

3.1 Algorithms and Implementations:

Three AEAD families were compared (Table 1). Ascon-AEAD128 follows SP 800-232 (Sönmez Turan et al., 2025) and uses a 128-bit key, a 128-bit nonce and a 128-bit tag. AES-128-GCM uses a 128-bit key, the recommended 96-bit nonce and a 128-bit tag (Dworkin, 2007). ChaCha20-Poly1305 follows RFC 8439 with a 256-bit key, a 96-bit nonce and a 128-bit tag (Nir & Langley, 2018). The tag length, and therefore the packet expansion, is the same 16 bytes for all three. The key lengths are not equal: ChaCha20 has only a 256-bit key, so its nominal security level is higher than that of the two 128-bit schemes, which is noted when results are compared.

All implementations were taken unchanged from public sources. For Ascon, the designers' reference package ascon-c (commit 446347f, January 2026) provides several implementations of the same algorithm for different goals. Five were used on the Cortex-M4: the plain reference code (ref), portable 32-bit optimized C (opt32), and three Cortex-M4 variants with inline assembly for the permutation, called armv7m (unrolled rounds, for speed), armv7m_small (a round loop) and armv7m_lowsize (smallest code). On the Cortex-M0+ the ref and opt32 variants were used; the package's assembly for this core is written for the GNU assembler's syntax and could not be built with the compiler used here (Section 4.7). AES-128-GCM was taken from Mbed TLS 3.6.2, the cryptography library of the Trusted Firmware project, with the AES tables placed in flash (gcm-mbed) and with the option that keeps only one of the four tables (gcm-mbed-small). ChaCha20-Poly1305 was taken from Mbed TLS 3.6.2 (ccp-mbed) and from Monocypher 4.0.3, a small single-file library (ccp-mono). The Mbed TLS configuration enabled only the modules each scheme needs.

Table (1). Implementations evaluated. All use a 16-byte tag; "M4" and "M0+" mark the cores for which each was built

Label	Algorithm	Source (version)	Key/nonce (bytes)	Technique	Cores
Ascon ref	Ascon-AEAD128	ascon-c (commit 446347f)	16 / 16	Portable reference C	M4, M0+
Ascon opt32	Ascon-AEAD128	ascon-c (commit 446347f)	16 / 16	Portable 32-bit C, inlined and unrolled permutation	M4, M0+
Ascon armv7m	Ascon-AEAD128	ascon-c (commit 446347f)	16 / 16	Inline assembly, unrolled rounds	M4
Ascon armv7m_small	Ascon-AEAD128	ascon-c (commit 446347f)	16 / 16	Inline assembly, round loop	M4
Ascon armv7m_lowsize	Ascon-AEAD128	ascon-c (commit 446347f)	16 / 16	Inline assembly, round loop, minimal code	M4
AES-GCM Mbed	AES-128-GCM	Mbed TLS 3.6.2	16 / 12	Four AES tables in flash; 4-bit GHASH table	M4, M0+
AES-GCM Mbed small	AES-128-GCM	Mbed TLS 3.6.2	16 / 12	One AES table with rotations; 4-bit GHASH table	M4, M0+
ChaCha20-Poly1305 Mbed	ChaCha20-Poly1305	Mbed TLS 3.6.2	32 / 12	Portable C	M4, M0+
ChaCha20-Poly1305 Monocypher	ChaCha20-Poly1305	Monocypher 4.0.3	32 / 12	Portable C, single-file library	M4, M0+

3.2 Firmware and Build:

Each implementation was linked into the same minimal bare-metal firmware. The firmware has a vector table, a start-up routine that initializes RAM, and a main function that runs three phases in order: key setup, authenticated encryption of one message, and verified decryption of the resulting ciphertext. A global phase variable is written before each phase, so that the emulator can attribute every executed instruction to a phase. The key, nonce, associated data (AD) and message are placed in a RAM section that the start-up code does not initialize, and the emulator writes them there before each run. An optional flag inverts one bit of the tag before decryption, to test the handling of a forged packet. A small C library supplies byte-wise memcpy, memset and related functions, as found in size-optimized embedded C libraries, together with the ARM run-time helpers for 64-bit shifts and multiplications that the Cortex-M0+ needs.

The firmware was compiled with Clang 18.1.3 for the Cortex-M4 (Armv7E-M, software floating point, unaligned access enabled) and for the Cortex-M0+ (Armv6-M), at the optimization levels -Os (size) and -O2 (speed), with per-function sections and garbage collection at link time. Built-in expansion of small fixed-size copies was enabled, as in a normal embedded tool chain. The image without any cryptographic code (null) was also built, and its sizes were subtracted, so that the reported footprints contain only the AEAD code, its constant tables and its state.

3.3 Emulation and Cycle Model:

The firmware images were executed in the Unicorn CPU emulator (version 2.0.1), which interprets the Arm Thumb-2 instruction set exactly, and every executed instruction was decoded with Capstone. This gives the exact number of instructions retired in each phase. Version 2.1.4 of the emulator was also tried, but it mis-executed one conditional-instruction sequence in the armv7m_lowsize images, so version 2.0.1 was used for all runs; for all other images the two versions gave identical results. The emulator does not report instructions inside an IT (if-then) block whose condition fails. On the Cortex-M4 such an instruction still occupies one cycle, so the instruction addresses of every IT block were decoded and each skipped instruction was counted as one instruction and one cycle, without a pipeline refill.

Emulation does not model the processor pipeline, so the execution time was estimated with a cycle model built from the instruction timing tables of the two cores (ARM Limited, 2010, 2012). Each instruction i receives a base cost c_i: one cycle for data-processing instructions and single-cycle multiplications, two cycles for loads and stores, $1 + N$ cycles for loads and stores of N registers, three for 64-bit loads and stores, and two for multiply-accumulate on the Cortex-M4. A taken branch, or any other write to the program counter, adds a pipeline refill of P cycles, with $P = 2$ on the three-stage Cortex-M4 and $P = 1$ on the two-stage Cortex-M0+. On the Cortex-M4, a load or store that directly follows another load or store is pipelined and saves one cycle. The cycle estimate of a phase is therefore.

$$C = \sum_i c_i + P\, N_{\mathrm{br}} - N_{\mathrm{pipe}} \tag{1}$$

where N_{br} is the number of non-sequential instruction fetches and N_{pipe} the number of pipelined memory instructions. Memory with zero wait states is assumed, which corresponds to running from flash at a low clock frequency or from a flash accelerator with a high hit rate. Because the model is an approximation, its sensitivity to P and to the pipelining rule was tested (Section 4.6).

The peak stack depth was obtained from the lowest stack address written in each phase; the depth of the null image was subtracted. Code size (flash) is the size of the text and read-only data sections, and static RAM is the size of the initialized and zero-initialized data sections, both relative to the null image.

3.4 Energy Estimate:

The energy of a computation on a microcontroller at a fixed supply voltage is close to proportional to its cycle count. With a run-mode supply current of k µA per MHz of clock frequency and a supply voltage V, the energy of C cycles is

$$E = C\, k\, V \times 10^{-12}\ \mathrm{J} \tag{2}$$

independently of the clock frequency. A value of $k = 100$ µA/MHz at $V = 3.0$ V, typical of low-power Cortex-M4 microcontrollers, was used, which gives 0.3 nJ per cycle. The energy figures are

therefore a linear rescaling of the cycle counts. Their purpose is to show the order of magnitude of the cost per packet, and they should be rescaled with the data-sheet values of a specific device.

3.5 Verification and Workload:

Correctness was checked for every image before any cost was recorded. The Ascon images were run on 33 entries of the official known-answer test file of the ascon-c package, which covers messages and AD of 0 to 32 bytes, and the ciphertext and tag had to match exactly. The AES-GCM and ChaCha20-Poly1305 images were compared with the Python *cryptography* library (version 50.0.1), which is independent of both C libraries. In every run, the decrypted message had to equal the original and the tag had to verify.

The main workload used message lengths of 0, 16, 32, 64, 128, 256, 512 and 1024 bytes, each with 0 and 16 bytes of AD, and random keys, nonces and data. The asymptotic cost per byte was computed from the difference between the 1024-byte and 512-byte runs, which removes the fixed cost of initialization and finalization. To test whether the execution depends on secret data, each image was also run with ten different random keys, nonces, AD and messages of fixed length (64-byte message, 16-byte AD), and the instruction and cycle counts were compared. Finally, each image decrypted a packet with one inverted tag bit; the result had to be a rejection, and the cost and the content of the output buffer were recorded.

As an application scenario, a sensor packet with a 16-byte header used as AD and a 32-byte payload was considered, which is typical of periodic telemetry over low-power radio links. Two key-management patterns were compared: one key setup for a long series of packets, and one key setup per packet, as happens when the key changes often or the device sleeps with its RAM switched off.

4. Results and Discussion

4.1 Correctness:

All 30 firmware images passed every correctness check. The five Ascon implementations reproduced the ciphertexts and tags of all 33 known-answer entries, which confirms that the ascon-c code used here implements the final SP 800-232 version of Ascon-AEAD128. The AES-GCM and ChaCha20-Poly1305 images agreed bit for bit with the independent Python library in all 16 random test cases each, and every image decrypted its own ciphertext to the original message. The cost figures below therefore refer to implementations whose output is known to be correct on the emulated cores.

4.2 Memory Footprint:

Table (2) lists the flash, static RAM and stack needed by each implementation. The flash footprint varied by a factor of 27 within Ascon alone: the assembly variant for the smallest code (armv7m_lowsize) needed only 1,266 bytes on the Cortex-M4, whereas the portable optimized C version (opt32) needed 34.8 KB, because it inlines and fully unrolls the permutation at every call site. The fast assembly variant (armv7m) needed 7.3 KB, and the armv7m_small variant 2.2 KB. Both

ChaCha20-Poly1305 libraries were compact (2.6 KB at -Os). AES-GCM was the largest of the established schemes: 14.0 KB with the full AES tables and 7.8 KB with the reduced-table option.

Static RAM depends mainly on the key material that must be kept between packets. Ascon has no key schedule, so only the 16-byte key is stored, and the 32 bytes of Monocypher are its stored ChaCha20 key. The Mbed TLS ChaCha20-Poly1305 context needs 232 bytes. The AES-GCM context needs 616 bytes, because it holds the expanded AES key and the 16-entry table for the GHASH multiplication. On a microcontroller with 8 or 16 KB of RAM, this difference is noticeable, in particular when one context per peer must be kept. The peak stack depth during a complete encryption and decryption was between 240 and 400 bytes for all implementations except Monocypher, which needed 560–590 bytes because it allocates its working buffers on the stack instead of in a static context.

Table (2). Memory footprint of the implementations: flash (code and constant tables), static RAM (state kept between calls) and peak stack depth, all relative to the firmware without cryptography. "–" marks configurations that were not built

Implementation	Flash M4 -Os (B)	Flash M4 -O2 (B)	Static RAM (B)	Stack M4 -Os (B)	Flash M0+ -Os (B)	Flash M0+ -O2 (B)	Stack M0+ -Os (B)
Ascon ref	10,808	14,260	16	240	10,762	13,786	308
Ascon opt32	34,826	34,804	16	280	38,360	38,514	336
Ascon armv7m	7,346	7,760	16	308	–	–	–
Ascon armv7m_small	2,194	2,368	16	284	–	–	–
Ascon armv7m_lowsize	1,266	1,632	16	332	–	–	–
AES-GCM Mbed	13,978	17,200	616	336	14,158	18,406	360
AES-GCM Mbed small	7,794	11,000	616	328	8,202	12,450	360
ChaCha20-Poly1305 Mbed	2,642	3,588	232	392	3,516	4,548	400
ChaCha20-Poly1305 Monocypher	2,593	5,516	32	580	3,014	5,790	560

4.3 Execution Time on the Cortex-M4:

Table (3) and Figure 1(a) give the cycle counts on the Cortex-M4. Three observations stand out. First, AES-GCM was the slowest scheme at all message lengths except the empty message: 240 cycles per byte at -Os and 135 at -O2, against about 40 cycles per byte for the best Ascon and ChaCha20-Poly1305 implementations. A profile of the AES-GCM image for a 1,024-byte message showed that 63% of the executed instructions were spent in the table-based GHASH multiplication and 25% in the AES block function. In addition, AES-GCM needed about 8,000 cycles of key setup at -Os, mostly to compute the GHASH table, while the key setup of Ascon (142 cycles) is only a copy of the key.

Second, Ascon was fastest for short messages. For a 16-byte message with 16 bytes of AD, the fast assembly variant needed 4,144 cycles, against 6,635 for ChaCha20-Poly1305 from Mbed TLS (60% more), 8,981 for Monocypher and 11,142 for AES-GCM (2.7 times more). The reason is the fixed

cost per message. Ascon needs one 12-round permutation for initialization and one for finalization, and it absorbs the AD and the message directly. ChaCha20-Poly1305 must generate one ChaCha20 block only to derive the Poly1305 key, pad the AD and the message to 16 bytes and authenticate a length block, and Monocypher at -Os adds the cost of a size-optimized block function.

Third, for long messages the advantage reversed slightly. The cost per byte of ChaCha20-Poly1305 was 38.5 cycles (Monocypher) and 39.3 cycles (Mbed TLS) at -Os, compared with 41.3 for the fast Ascon assembly. At 1,024 bytes the total cost of the two was almost equal (44,476 and 44,638 cycles), and at -O2 the Mbed TLS ChaCha20-Poly1305 reached 32.4 cycles per byte, 22% faster than Ascon. The Ascon assembly variants were insensitive to the optimization level, because their permutation is hand-written, while the speed of the C implementations depended strongly on it: the Ascon reference code improved from 95.2 to 64.4 cycles per byte between -Os and -O2, and AES-GCM from 240 to 135.

The implementation mattered as much as the algorithm. The three Ascon assembly variants were within 15% of each other (41.3 to 47.1 cycles per byte), but the portable reference code at -Os was 2.3 times slower than the fastest variant. A claim that "Ascon is faster than ChaCha20-Poly1305" is therefore true for short packets with assembly code, but false for long messages, and false for short messages if the reference C code is used at -Os (Table 3).

Table (3). Cortex-M4 cycle counts: key setup, encryption of the given message (without AD unless stated) at -Os, and the asymptotic cost per byte of encryption and decryption. The lowest value in each column is in bold

Implementation	Key setup	0 B	16 B + 16 B AD	64 B + 16 B AD	256 B	1024 B	Enc. c/B - Os	Dec. c/B -Os	Enc. c/B - O2
Ascon ref	142	3,881	7,680	12,252	28,262	101,414	95.2	95.2	64.4
Ascon opt32	142	2,709	5,915	9,050	19,440	69,600	65.3	65.9	63.8
Ascon armv7m	142	2,174	4,144	6,127	12,748	44,476	41.3	41.3	41.4
Ascon armv7m_small	142	2,291	4,440	6,597	13,794	48,306	44.9	44.9	44.9
Ascon armv7m_lowsize	142	2,498	4,795	7,054	14,548	50,692	47.1	47.3	47.1
AES-GCM Mbed	8,024	1,918	11,142	22,686	66,289	250,993	240.5	240.4	134.6
AES-GCM Mbed small	7,950	1,844	10,994	22,316	65,031	246,183	235.9	235.8	127.1
ChaCha20-Poly1305 Mbed	1,395	4,236	6,635	7,107	14,446	44,638	39.3	39.3	32.4
ChaCha20-Poly1305 Monocypher	270	6,186	8,981	9,102	16,140	45,672	38.5	38.5	37.8

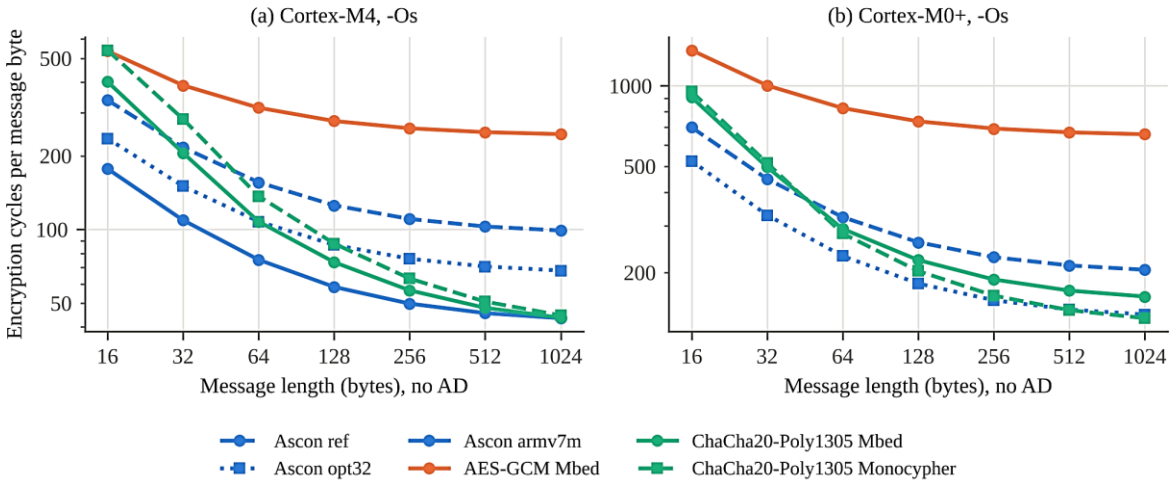


Figure (1). Encryption cost per message byte as a function of the message length (no AD) at -Os: (a) Cortex-M4; (b) Cortex-M0+. Both axes are logarithmic

4.4 Execution Time on the Cortex-M0+:

The Cortex-M0+ has only the 16-bit Thumb instruction set, eight easily usable registers, no barrel shifter in data-processing instructions and no 64-bit multiply. All implementations became slower (Table 4, Figure 1(b)). The cost per byte of Ascon and AES-GCM rose by a factor of about 2.0 to 2.7, that of ChaCha20-Poly1305 by 3.3 to 3.9. Poly1305 needs 32×32-bit multiplications with a 64-bit result, which the Cortex-M0+ must build from several 32-bit multiplications, whereas Ascon only needs its 64-bit rotations to be split into 32-bit shifts. The *opt32* variant was the fastest for messages of up to 256 bytes (12,521 cycles for 16 bytes with 16 bytes of AD), and Monocypher's ChaCha20-Poly1305 was the fastest for long messages (125.4 cycles per byte against 133.3). The choice of implementation again had a large effect: at -Os the Ascon reference code needed 197 cycles per byte, 48% more than *opt32*. However, the faster *opt32* variant costs 38 KB of flash on this core (Table 2), which can exceed the budget of a small Cortex-M0+ device; at -O2 the reference code reached the same speed (134.8 cycles per byte) with 13.8 KB. AES-GCM was 2.4 times slower than Ascon for a 16-byte packet and about five times slower per byte, and it needed about 12,000 cycles of key setup.

On this core the one-table AES variant was slightly slower than the four-table variant (661 against 647 cycles per byte), whereas on the Cortex-M4 it was slightly faster (235.9 against 240.5). The one-table variant replaces three table look-ups by rotations of the looked-up word. The Cortex-M4 performs a rotation for free as part of an exclusive-OR, so the smaller table also saves memory accesses, but the Cortex-M0+ needs a separate instruction for each rotation.

Table (4). Cortex-M0+ cycle counts, arranged as in Table 3

Implementation	Key setup	0 B	16 B + 16 B AD	64 B + 16 B AD	256 B	1024 B	Enc. c/B - Os	Dec. c/ B -Os	Enc. c/B - O2
Ascon ref	**141**	8,036	15,854	25,313	58,482	209,826	197.1	201.2	134.8
Ascon opt32	**141**	6,228	**12,521**	**18,920**	**40,366**	142,750	133.3	133.4	134.4
AES-GCM Mbed	11,991	**2,820**	29,859	60,939	177,147	674,427	647.5	647.4	448.6
AES-GCM Mbed small	12,214	3,043	30,305	62,054	180,938	688,922	661.4	661.4	462.5
ChaCha20-Poly1305 Mbed	1,530	8,648	15,829	19,983	48,258	166,638	154.1	154.1	149.2
ChaCha20-Poly1305 Monocypher	269	9,761	16,397	19,093	42,000	**138,300**	**125.4**	**125.4**	**124.5**

4.5 Code Size against Speed and Energy per Packet:

Figure 2 places every Cortex-M4 build in the plane of flash footprint and cost of a typical sensor packet (32-byte payload, 16-byte header as AD). The three Ascon assembly variants lie on the lower left edge of the plot: no other implementation was both smaller and faster than any of them. The Mbed TLS ChaCha20-Poly1305 was a close alternative (2.6 KB, 6,788 cycles). AES-GCM was slower than every other implementation, and larger than all of them except the two portable Ascon builds. The portable Ascon *opt32* build shows that unrolling and inlining in C buys little speed on this core at a very high cost in flash.

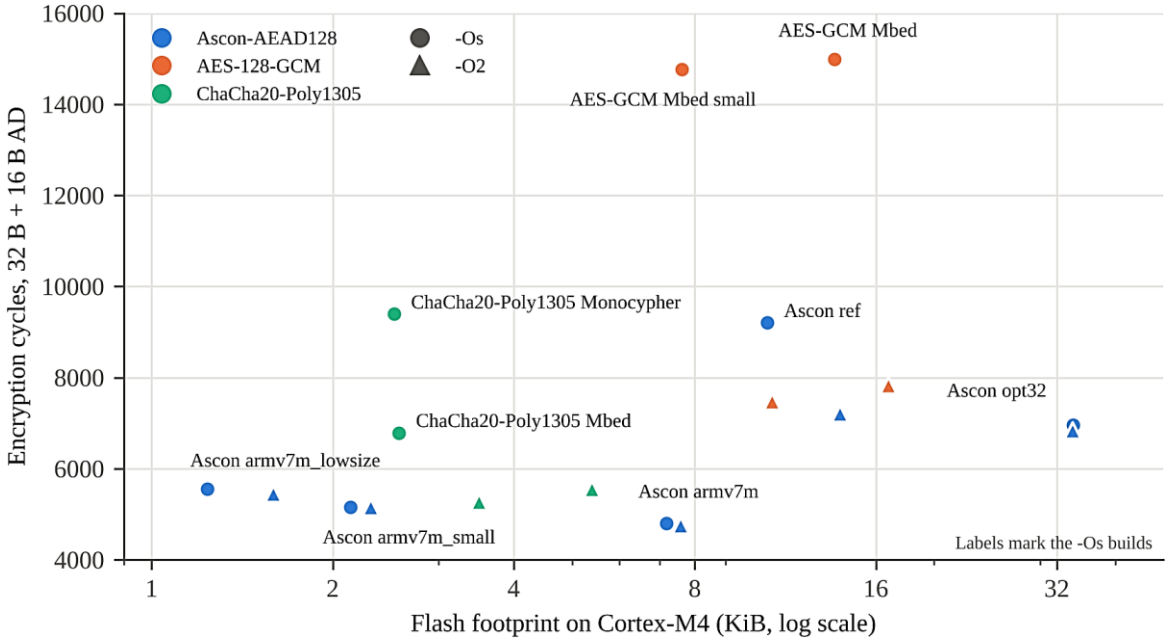


Figure (2). Flash footprint against the encryption cost of a 32-byte packet with 16 bytes of AD on the Cortex-M4. Circles: -Os; triangles: -O2. The color identifies the algorithm

Table (5) and Figure (3) convert the packet costs into energy with the model of Section 3.4 (0.3 nJ per cycle). On the Cortex-M4 the encryption of one packet cost 1.44 µJ with the fast Ascon assembly, 2.04 µJ with ChaCha20-Poly1305 from Mbed TLS and 4.50 µJ with AES-GCM. If a key setup is needed for every packet, the AES-GCM cost rises to 6.90 µJ, or 4.7 times that of Ascon, while Ascon is almost unaffected. On the Cortex-M0+ the corresponding values were 4.40 µJ for Ascon *opt32*, 5.16 µJ for ChaCha20-Poly1305 and 12.07 µJ (15.66 µJ with key setup) for AES-GCM. At a clock frequency of 16 MHz the fastest Cortex-M4 encryption of such a packet takes 300 µs. These energies are small compared with the energy that a low-power radio needs to transmit a packet, so the choice of AEAD matters most for devices that process many packets per transmission, for example gateways and relays, for devices that run from energy harvesting with very small energy budgets, and for the processor time that remains available to the application.

Table (5). Cost of a sensor packet with a 32-byte payload and a 16-byte header as AD at -Os, with energy at 0.3 nJ per cycle (100 µA/MHz at 3.0 V). The decryption cost includes verification of the tag

Implementation	Core	Enc. (cycles)	Dec. (cycles)	Key setup (cycles)	Enc. energy, key reused (µJ)	Enc. energy, key setup per packet (µJ)	Enc. time at 16 MHz, key reused (µs)
Ascon ref	M4	9,204	9,385	142	2.76	2.80	575
Ascon opt32	M4	6,960	6,979	142	2.09	2.13	435
Ascon armv7m	M4	4,805	4,807	142	1.44	1.48	300
Ascon armv7m_small	M4	5,159	5,160	142	1.55	1.59	322
Ascon armv7m_lowsize	M4	5,548	5,587	142	1.66	1.71	347
AES-GCM Mbed	M4	14,990	15,114	8,024	4.50	6.90	937
AES-GCM Mbed small	M4	14,768	14,892	7,950	4.43	6.82	923
ChaCha20-Poly1305 Mbed	M4	6,788	6,873	1,395	2.04	2.45	424
ChaCha20-Poly1305 Monocypher	M4	9,391	9,508	270	2.82	2.90	587
Ascon ref	M0+	19,007	19,352	141	5.70	5.74	1188
Ascon opt32	M0+	14,654	14,782	141	4.40	4.44	916
AES-GCM Mbed	M0+	40,219	40,473	11,991	12.07	15.66	2514
AES-GCM Mbed small	M0+	40,888	41,142	12,214	12.27	15.93	2556
ChaCha20-Poly1305 Mbed	M0+	17,208	17,424	1,530	5.16	5.62	1076
ChaCha20-Poly1305 Monocypher	M0+	17,617	17,829	269	5.29	5.37	1101

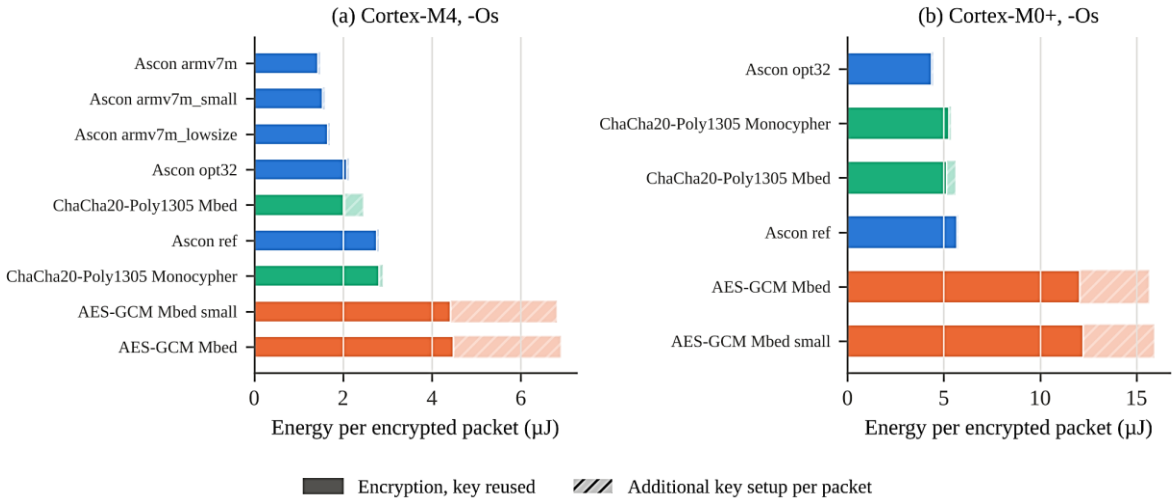


Figure (3). Energy to encrypt one 32-byte packet with 16 bytes of AD at -Os, with the key set up once (solid) and with an additional key setup per packet (hatched): (a) Cortex-M4; (b) Cortex-M0+

4.6 Data Independence, Forged Packets and Robustness of the Cycle Model:

For every image, ten encryptions and decryptions with different random keys, nonces, AD and messages of the same length gave exactly the same instruction and cycle counts (Table 6), so no implementation contains branches that depend on secret data. This result is necessary but not sufficient for timing safety. The AES implementations look up tables at addresses that depend on the key and the data. The emulation assumes memory with a uniform access time, so it cannot show the variations that a data cache or a flash cache would cause; such variations are the basis of cache-timing attacks on table-based AES (Osvik et al., 2006). Ascon and ChaCha20-Poly1305 use no secret-dependent addresses, and therefore do not have this risk on any memory system (Schwabe & Stoffelen, 2017).

All images rejected a packet with one inverted tag bit and left no plaintext in the output buffer. Ascon and the Mbed TLS implementations decrypt the whole message before they compare the tag, and Mbed TLS then clears the output, which adds about 410 cycles to a rejection. Monocypher checks the tag before it decrypts, so rejecting a forged 64-byte packet cost 31% fewer cycles than accepting a valid one on the Cortex-M4 (28% on the Cortex-M0+). This is useful against attackers who send many forged packets to drain the battery, and the difference does not leak secret information, because whether a tag is valid is also revealed by the rejection itself.

Table (6). Data independence and handling of a forged packet (64-byte message, 16-byte AD, -Os). The second column gives the number of distinct cycle counts over ten random inputs

Implementation	Core	Distinct cycle counts over 10 random inputs	Dec. valid (cycles)	Dec. forged (cycles)	Forgery rejected	Output cleared
Ascon ref	M4	1	12,431	12,431	yes	yes
Ascon opt32	M4	1	9,089	9,089	yes	yes
Ascon armv7m	M4	1	6,129	6,129	yes	yes
Ascon armv7m_small	M4	1	6,598	6,598	yes	yes
Ascon armv7m_lowsize	M4	1	7,101	7,101	yes	yes
AES-GCM Mbed	M4	1	22,808	23,221	yes	yes
AES-GCM Mbed small	M4	1	22,438	22,851	yes	yes
ChaCha20-Poly1305 Mbed	M4	1	7,192	7,605	yes	yes
ChaCha20-Poly1305 Monocypher	M4	1	9,219	6,354	yes	yes
Ascon ref	M0+	1	25,792	25,792	yes	yes
Ascon opt32	M0+	1	19,052	19,052	yes	yes
AES-GCM Mbed	M0+	1	61,191	61,605	yes	yes
AES-GCM Mbed small	M0+	1	62,306	62,720	yes	yes
ChaCha20-Poly1305 Mbed	M0+	1	20,199	20,613	yes	yes
ChaCha20-Poly1305 Monocypher	M0+	1	19,305	13,906	yes	yes

Because the cycle counts come from a model, Table (7) repeats the Cortex-M4 measurements with three changes to it: no pipelining of neighbouring loads and stores, and a pipeline refill of one or three cycles instead of two. The counts changed by at most 7%, and the change was largest for AES-GCM and ChaCha20-Poly1305, which use more memory accesses and branches. For 64-byte messages the ranking of the nine implementations was the same under all four models, except that with a one-cycle refill ChaCha20-Poly1305 from Mbed TLS moved ahead of the smallest Ascon variant. For 1,024-byte messages the fast Ascon assembly and the two ChaCha20-Poly1305 libraries, which differ by less than 3%, changed places between models, so the study does not claim a winner among these three for long messages. AES-GCM was last under every model, and its distance to the other schemes was far larger than the model uncertainty.

Table (7). Sensitivity of the Cortex-M4 encryption cycle counts to the cycle model (-Os; 64-byte message with 16-byte AD / 1,024-byte message without AD)

Implementation	Baseline (P = 2, pipelining)	No load/store pipelining	P = 1	P = 3
Ascon ref	12,252 / 101,414	12,343 / 102,026	12,056 / 99,376	12,448 / 103,452
Ascon opt32	9,050 / 69,600	9,216 / 71,043	9,035 / 69,525	9,065 / 69,675
Ascon armv7m	6,127 / 44,476	6,159 / 44,746	6,093 / 44,267	6,161 / 44,685
Ascon armv7m_small	6,597 / 48,306	6,644 / 48,650	6,503 / 47,631	6,691 / 48,981
Ascon armv7m_lowsize	7,054 / 50,692	7,139 / 51,367	6,920 / 49,865	7,188 / 51,519
AES-GCM Mbed	22,686 / 250,993	23,868 / 264,645	21,631 / 239,910	23,741 / 262,076
AES-GCM Mbed small	22,316 / 246,183	23,383 / 258,340	21,261 / 235,100	23,371 / 257,266
ChaCha20-Poly1305 Mbed	7,107 / 44,638	7,432 / 47,333	6,615 / 42,455	7,599 / 46,821
ChaCha20-Poly1305 Monocypher	9,102 / 45,672	9,443 / 48,361	8,503 / 44,582	9,701 / 46,762

4.7 Discussion:

The results give a consistent picture. For the short packets that dominate IoT traffic, the standardized Ascon-AEAD128 was the cheapest AEAD on both cores, in time and in energy, provided that a suitable implementation is used. On the Cortex-M4 its assembly variants also had the smallest code, and the *armv7m_small* variant (2.2 KB, 44.9 cycles per byte) is a good default because it is within 10% of the fastest variant at less than a third of its size. ChaCha20-Poly1305 was a strong second: it was slightly faster for messages longer than about 1 KB, it is small, and it is available in many libraries, but it needs a 256-bit key and has a higher fixed cost per message. AES-GCM in portable software was the least suitable choice. It was three to six times slower than the alternatives on both cores, it had the largest code and RAM context, its key setup was expensive, and its table look-ups are a timing risk on devices with caches. These results agree with the selection criteria of the NIST process (Sönmez Turan et al., 2023) and extend the Arduino-based results of Radhakrishnan et al. (2024), who found AES faster than Ascon in execution time on 8-bit AVR boards: on the 32-bit Arm cores, where Ascon's 64-bit operations map well to pairs of 32-bit registers, the ranking is the opposite.

The study also shows how strongly the results depend on factors other than the algorithm. A change of the Ascon implementation changed the cost by up to 2.3 times and the flash footprint by 27 times, and a change of the optimization level almost halved the cost of AES-GCM. Two technical details of the tool chain also changed the results during this study: with unaligned memory access disabled (the default of the compiler for this bare-metal target), the Mbed TLS GHASH routine falls back to byte-wise operations, and without built-in expansion of small fixed-size copies every 32-bit load in Mbed TLS becomes a function call. Benchmarks that do not report the implementation, the compiler and these options are therefore difficult to compare.

The study has limitations. The results come from emulation with a cycle model, not from measurements on a physical board. The model assumes memory without wait states and ignores flash accelerators and caches, which favour code that fits into a small cache and penalize large unrolled code at high clock frequencies. Only one compiler (Clang 18) was used, and GCC may produce faster or slower code. Many current microcontrollers contain an AES hardware accelerator, which would make AES-GCM much faster; the results apply to devices without one, or to designs that want a portable software solution. The energy estimate assumes a constant energy per cycle. Resistance to power-analysis attacks, for which protected Ascon implementations exist, was not evaluated. The Ascon assembly variants for the Cortex-M0+ could not be built because they use the syntax of the GNU assembler; they would probably make Ascon faster on that core as well.

5. Conclusions and Recommendations

This study compared the NIST lightweight standard Ascon-AEAD128 with AES-128-GCM and ChaCha20-Poly1305 on the Arm Cortex-M4 and Cortex-M0+, using nine public implementations, 30 firmware images and an instruction-accurate emulation of the real binaries. All implementations were

verified against official test vectors or an independent library before their cost was measured. The main conclusions are:

1. For short packets, Ascon-AEAD128 was the most efficient AEAD on both cores. For a 32-byte payload with a 16-byte header, the fastest Ascon implementation needed 4,805 cycles on the Cortex-M4, 29% fewer than the best ChaCha20-Poly1305 and 68% fewer than AES-GCM, and its estimated energy was 1.44 µJ against 2.04 µJ and 4.50 µJ.
2. For long messages, ChaCha20-Poly1305 and assembly Ascon were equivalent within the uncertainty of the cycle model (about 39–41 cycles per byte at -Os on the Cortex-M4), and ChaCha20-Poly1305 was faster at -O2 and on the Cortex-M0+.
3. Software AES-GCM was three to six times slower than the alternatives, needed the largest code (7.8–14 KB) and RAM context (616 bytes), and required a key setup of 8,000–12,000 cycles.
4. The implementation and the compiler options affected the results as much as the choice of algorithm, with differences of up to 2.3 times in speed and 27 times in code size among implementations of Ascon alone.
5. All implementations showed data-independent instruction counts and correctly rejected forged packets; only the table-based AES has secret-dependent memory addresses.

For engineers of IoT nodes without an AES accelerator, the study recommends Ascon-AEAD128 as the default AEAD, using a core-specific implementation such as armv7m_small on the Cortex-M4 and avoiding the reference code at -Os. ChaCha20-Poly1305 is recommended where a device mostly handles long messages or where compatibility with existing protocols requires it, and AES-GCM in software only where interoperability demands it. Benchmarks of lightweight cryptography should report the implementation, compiler version and options, and the message lengths, together with the memory footprint and key-setup cost. Future work will validate the cycle model with measurements on physical boards, include microcontrollers with AES accelerators and caches, port the Cortex-M0+ assembly of Ascon, and evaluate protected implementations against power analysis.

Declarations

Data availability: The firmware sources, build scripts, emulator and cycle model, and the raw results that support the findings of this study are available from the corresponding author on reasonable request. The cryptographic libraries are publicly available: ascon-c (https://github.com/ascon/ascon-c), Mbed TLS (https://github.com/Mbed-TLS/mbedtls) and Monocypher (https://monocypher.org).

Conflict of interest: The author declare no conflict of interest.

Funding: This research received no external funding.

References

[1] ARM Limited. (2010). Cortex-M4 technical reference manual (Revision r0p1, ARM DDI 0439B). ARM Limited.

[2] ARM Limited. (2012). Cortex-M0+ technical reference manual (Revision r0p1, ARM DDI 0484C). ARM Limited.

[3] Atul, Shukla, V., Das, M. K., Shukla, V., & Mishra, D. (2026). A reproducible security–performance benchmark protocol for lightweight AEAD on resource-constrained IoT nodes (Cryptology ePrint Archive, Paper 2026/1882). International Association for Cryptologic Research. https://eprint.iacr.org/2026/1882.

[4] Bernstein, D. J. (2005). The Poly1305-AES message-authentication code. In H. Gilbert & H. Handschuh (Eds.), Fast software encryption (Lecture Notes in Computer Science, Vol. 3557, pp. 32–49). Springer. https://doi.org/10.1007/11502760_3.

[5] Dobraunig, C., Eichlseder, M., Mendel, F., & Schläffer, M. (2021). Ascon v1.2: Lightweight authenticated encryption and hashing. Journal of Cryptology, 34(3), Article 33. https://doi.org/10.1007/s00145-021-09398-9.

[6] Dworkin, M. (2007). Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC (NIST Special Publication 800-38D). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-38D.

[7] Fotovvat, A., Rahman, G. M. E., Vedaei, S. S., & Wahid, K. A. (2021). Comparative performance analysis of lightweight cryptography algorithms for IoT sensor nodes. IEEE Internet of Things Journal, 8(10), 8279–8290. https://doi.org/10.1109/JIOT.2020.3044526.

[8] Hatzivasilis, G., Fysarakis, K., Papaefstathiou, I., & Manifavas, C. (2018). A review of lightweight block ciphers. Journal of Cryptographic Engineering, 8(2), 141–184. https://doi.org/10.1007/s13389-017-0160-y.

[9] Käsper, E., & Schwabe, P. (2009). Faster and timing-attack resistant AES-GCM. In C. Clavier & K. Gaj (Eds.), Cryptographic hardware and embedded systems – CHES 2009 (Lecture Notes in Computer Science, Vol. 5747, pp. 1–17). Springer. https://doi.org/10.1007/978-3-642-04138-9_1.

[10] National Institute of Standards and Technology. (2023). Advanced Encryption Standard (AES) (Federal Information Processing Standards Publication 197, updated May 9, 2023). https://doi.org/10.6028/NIST.FIPS.197-upd1.

[11] Nir, Y., & Langley, A. (2018). ChaCha20 and Poly1305 for IETF protocols (RFC 8439). Internet Engineering Task Force. https://doi.org/10.17487/RFC8439.

[12] Osvik, D. A., Shamir, A., & Tromer, E. (2006). Cache attacks and countermeasures: The case of AES. In D. Pointcheval (Ed.), Topics in cryptology – CT-RSA 2006 (Lecture Notes in Computer Science, Vol. 3860, pp. 1–20). Springer. https://doi.org/10.1007/11605805_1.

[13] Radhakrishnan, I., Jadon, S., & Honnavalli, P. B. (2024). Efficiency and security evaluation of lightweight cryptographic algorithms for resource-constrained IoT devices. Sensors, 24(12), Article 4008. https://doi.org/10.3390/s24124008.

[14] Renner, S., Pozzobon, E., & Mottok, J. (2020). A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In W. Meng, D. Gollmann, C. D. Jensen, & J. Zhou (Eds.), Information and communications security – ICICS 2020 (Lecture Notes in Computer Science, Vol. 12282, pp. 495–509). Springer. https://doi.org/10.1007/978-3-030-61078-4_28.

[15] Sarasa Laborda, V., Hernández-Álvarez, L., Hernández Encinas, L., Sánchez García, J. I., & Queiruga-Dios, A. (2025). Study about the performance of Ascon in Arduino devices. Applied Sciences, 15(7), Article 4071. https://doi.org/10.3390/app15074071.

[16] Schwabe, P., & Stoffelen, K. (2017). All the AES you need on Cortex-M3 and M4. In R. Avanzi & H. Heys (Eds.), Selected areas in cryptography – SAC 2016 (Lecture Notes in Computer Science, Vol. 10532, pp. 180–194). Springer. https://doi.org/10.1007/978-3-319-69453-5_10.

[17] Shehzadi, S., Whaley, N., Khalid, A., Ghafoor, A., Arshad, S., Islam, F. U., & O'Neill, M. (2025). Kyber-KEM-Ascon: Benchmarking a lightweight post-quantum KEM on IoT devices. In 2025 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE. https://doi.org/10.1109/ISCAS56072.2025.11043792.

[18] Sönmez Turan, M., McKay, K., Chang, D., Bassham, L., Kang, J., Waller, N., Kelsey, J., & Hong, D. (2023). Status report on the final round of the NIST lightweight cryptography standardization process (NIST Interagency Report 8454). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8454.

[19] Sönmez Turan, M., McKay, K., Kang, J., Kelsey, J., & Chang, D. (2025). Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions (NIST Special Publication 800-232). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-232.

[20] Sorescu, T.-G., Chiriac, V.-M., Stoica, M.-A., Comsa, C.-R., Soroaga, I.-G., & Contac, A. (2025). Comparative performance analysis of lightweight cryptographic algorithms on resource-constrained IoT platforms. Sensors, 25(18), Article 5887. https://doi.org/10.3390/s25185887.

[21] Thakor, V. A., Razzaque, M. A., & Khandaker, M. R. A. (2021). Lightweight cryptography algorithms for resource-constrained IoT devices: A review, comparison and research opportunities. IEEE Access, 9, 28177–28193. https://doi.org/10.1109/ACCESS.2021.3052867.