Efficient Implementation of Authenticated Encryption on 16-bit MSP430 Microcontrollers

Christian Franck and Johann Großschädl

DCS and SnT, University of Luxembourg,
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{christian.franck,johann.groszschaedl}@uni.lu

Abstract. Algorithms for Authenticated Encryption with Associated Data (AEAD) extend the normal functionality of authenticated encryption schemes by the ability to process data that is only authenticated but not encrypted. Such algorithms have attracted much interest in the past few years, especially the question of how they can be designed and implemented efficiently to perform well in resource-constrained devices like miniature sensor nodes or RFID tags. In this paper, we analyze the performance of the lightweight AEAD schemes Elephant v2, Grain-128AEADv2, ISAP v2.0, PHOTON-Beetle, and Romulus v1.3 on the MSP430 family of 16-bit ultra-low-power microcontrollers. All five have in common that they offer large security margins and made it into the last round of the Lightweight Cryptography (LWC) standardization project of the U.S. National Institute of Standards and Technology. We describe how these AEAD algorithms can be implemented efficiently in software and introduce Assembly-level optimization techniques for the underlying primitives, which include three permutations, one tweakable block cipher, and one stream cipher. Furthermore, we present numerous detailed benchmarking results (i.e., execution time and code size) for the primitives as well as for the full AEAD algorithms for different lengths of plaintext and associated data. Our benchmarks clearly show that all five AEAD algorithms are much more efficient (up to almost two orders of magnitude) on MSP430 than indicated by results in the literature.

Keywords: Lightweight cryptography · AEAD algorithm · Permutation · Block cipher · MSP430 architecture

1 Introduction

Algorithms for authenticated encryption do not only produce ciphertext from plaintext (or vice versa), but also generate or verify a so-called authentication tag using the secret key. In this way, they are capable to simultaneously ensure the confidentiality and authenticity/integrity of data. *Authenticated Encryption with Associated Data (AEAD)* is a special variant of authenticated encryption that supports auxiliary, non-confidential data, which is authenticated but does not get encrypted. The canonical use case for AEAD schemes is the protection of network packets where the payload is encrypted and authenticated, but the

header containing the destination address must remain in the clear (yet should be authenticated) to enable routers to forward the packet. Historically, authenticated encryption schemes were based on generic composition of a symmetric encryption algorithm, commonly a block cipher, and a message-authentication algorithm such as HMAC or CBC-MAC [28]. However, generic composition is not ideal in terms of efficiency because the obtained authenticated encryption schemes need two different algorithms and make two separate passes over the data. Furthermore, generically-composed authenticated encryption is prone to (accidental) misuse since, for example, it can be extremely difficult to analyze whether MAC-then-Encrypt is secure in a certain context. These shortcomings have led to the construction of single-pass combined schemes for authenticated encryption, which are based on a single cryptographic primitive (in most cases either a block cipher or an unkeyed permutation).

AEAD schemes attracted a lot of attention in the past five years due to the Lightweight Cryptography (LWC) standardization project of the U.S. National Institute of Standards and Technology (NIST) [21]. This project had the form of a public, competition-like process with the goal of developing new standards for AEAD as well as hashing that perform "significantly better in constrained environments" compared to existing NIST standards, most notably AES-GCM and SHA-2 [24]. In August 2018, the NIST released a request for nominations for lightweight cryptographic schemes and also published detailed submission requirements [23]. A total of 57 candidates were submitted by the (extended) deadline of March 25, 2019, of which 56 satisfied the acceptance criteria and were considered as "proper and complete." These 56 proposals entered a multi-round evaluation procedure that took almost four years altogether. By the end of the first round the number of candidates was reduced to 32 and then, in the second round, an additional 22 algorithms were eliminated. The remaining ten candidates made it to the third and final round, in which they were scrutinized for security and efficiency over a period of 1.5 years. On February 7, 2023, the NIST declared the candidate Ascon [10] as the sole winner of the competition and to become the new standard for lightweight cryptography.

Besides security, the evaluation of submitted algorithm proposals took into account the efficiency in restricted environments, e.g., hardware and embedded software platforms. The NIST document describing evaluation criteria for the submissions mentions that performance on "a wide range of 8-bit, 16-bit, and 32-bit microcontroller architectures" is desirable [24]. NIST's evaluation of the software efficiency (i.e., execution time and binary code size) of the candidates largely relied on three major benchmarking initiatives. First and foremost, the NIST LWC team developed a benchmarking toolsuite and collected results on five 8-bit and 32-bit architectures: AVR, ARMv6-M (Cortex-M0+), ARMv7-M (Cortex-M3/M4), MIPS32 (PIC32), and Xtensa (LX6) [25]. The benchmarked implementations were contributed by the designer teams or by volunteers and independent researchers. The majority of designer teams developed optimized implementations for (at least) AVR and ARMv7-M, whereby most final-round submissions even included carefully-tuned Assembly code for the performance-

critical parts (e.g., the underlying primitive of the AEAD algorithm). A second source of benchmarking results were the implementations and execution times of the final-round candidates contributed by Rhys Weatherley [32]. He focused on AVR, ARM Cortex-M3, and ESP32 microcontrollers, the latter of which are based on the 32-bit Xtensa LX6 architecture (i.e., his evaluation platforms are a subset of the NIST platforms). Finally, also a team of researchers from OTH Regensburg collected benchmarking results of the AEAD candidates on various microcontrollers [26, 27]. They developed their own benchmarking toolset from scratch, which is capable to determine the execution time, code size, and RAM footprint of AEAD algorithms (i.e., hash functions are not supported). Besides AVR, Cortex-M3, Cortex-M7, and ESP32, they also generated benchmarks on a development board equipped with a 64-bit RISC-V processor [26].

NIST's recommendation for algorithm designers to consider "a wide range of 8-bit, 16-bit, and 32-bit microcontroller architectures" [24] hints at a highly-desirable property of lightweight cryptosystems, namely that of multi-platform (resp., cross-platform) efficiency. However, achieving this property is not trivial because the architectural and micro-architectural characteristics and features of modern microcontrollers differ significantly, even among architectures of the same word size. Consider, for example, Ascon, the winner of the NIST LWC competition and future standard for lightweight AEAD and hashing. Ascon is a permutation-based design that performs a lot of rotations of 64-bit words in its linear diffusion layer, whereby the rotation distances are fixed and have the following values: 1, 6, 7, 10, 17, 19, 28, 39, 41, 61 (see [10] for details). All these rotations can be executed in a single clock cycle on a 64-bit ARM processor; in many cases they can even be combined with some other arithmetic or logical instruction, making them basically free. On the other hand, the rotations take three instructions on a simple 64-bit RISC-V (i.e., RV64I) processor since the base instruction set is relatively minimalist and does not include explicit rotate instructions, i.e., a rotation has to be composed of two shifts and a XOR. The cycle count of 64-bit rotations further increases on 32-bit microcontrollers like the ARM Cortex-M3, though they support rotate instructions. However, these instructions can only rotate a 32-bit register but not a 64-bit word, i.e., a 64-bit rotation has to be emulated by four 32-bit shifts and two other instructions (costing six clock cycles in total). The situation becomes even worse on small 8/16-bit microcontrollers since most of them lack a barrel shifter, which means a multi-bit shift or multi-bit rotation has to be performed through a sequence of 1-bit shift (resp., 1-bit rotate) instructions [2, 8]. Depending on the rotation distance, rotating a 64-bit word can take more than 30 clock cycles.

Multi-platform efficiency is particularly important for lightweight cryptosystems targeting the Internet of Things (IoT) and its billions of resource-limited devices. These devices are highly diverse and heterogeneous, and they come in all shapes and sizes. Consequently, it is not surprising that there exist dozens of different microcontroller platforms, operating systems, and communication standards for the IoT, many of which are optimized to serve a niche domain with specific requirements and constraints. This heterogeneity of IoT devices is

Table 1. Basic properties of the AEAD algorithms (according to [29, Table 5]).

AEAD algorithm	Primitive	Mode	Key size	N. size	Tag size
Elephant v2	Spongent-π[160]	Enc-then-MAC	128	96	64
Grain-128AEADv2	Grain-128A	Enc-and-MAC	128	96	64
ISAP v2.0	Ascon P	Enc-then-MAC	128	128	128
PHOTON-Beetle	PHOTON_{256}	Sponge/COFB	128	128	128
Romulus v1.3	Skinny-128-384+	COFB	128	128	128

the main reason why all benchmarking initiatives for lightweight cryptosystems carried out during the NIST LWC competition collected results (e.g., execution time, binary code size) on a number of different microcontrollers architectures instead of one. However, when looking at the target architectures summarized above, one could argue that 32-bit platforms are over-represented, 8-bit platforms under-represented, and 16-bit platforms completely lacking. Indeed, none of the three main benchmarking toolsuites (NIST, Weatherley, OTH) supports a 16-bit microcontroller. Nonetheless, there exist a few benchmarking results of second and third-round LWC candidates on 16-bit MSP430 microcontrollers from Texas Instruments. For example, Blanc et al [6] analyze and compare the execution time of 18 of the 32 second-round candidates (which includes all ten finalists) on a MSP430F1611 microcontroller [31]. However, their results stem from C implementations (in most cases the reference C code of the designers) and lack Assembly-level optimizations for the performance-critical components (e.g., the underlying primitive). Alsahli et al [2] implemented the permutation of four permutation-based LWC finalists, which are Ascon, Sparkle, Tiny-Jambu, and Xoodyak, in MSP430 Assembly and also evaluated the execution time of the permutations and full AEAD algorithms using a MSP430F1611 as target platform. In an extended version of [2] (not yet publicly available), the final-round candidate Gift-COFB was included in the evaluation.

This paper introduces Assembly-optimized MSP430 implementations of the five final-round candidates Elephant v2 [5], Grain-128AEADv2 [15], ISAP v2.0 [9], PHOTON-Beetle [3], and Romulus v1.3 [12]. Some of the basic characteristics of these five AEAD algorithms, including the primitive they are based on, are summarized in Table 1. We developed Assembly implementations of the underlying primitive of each algorithm, while the rest (i.e., the mode) is written in portable C. Our optimized implementations shed some new light on the efficiency of these five AEAD algorithms on the MSP430 platform since all execution times reported in the literature were determined without Assembly-level optimizations. Our work aims to contribute to a better understanding on how lightweight AEAD algorithms can be optimized for small microcontrollers and what performance they can reach on the MSP430 platform, all of which is important for the progress of the field of lightweight cryptography. Finally, we point out that (variants of) four of the primitives we implemented, namely the stream cipher Grain-128a, the (tweakable) block cipher Skinny, and the two permutations Photon and Spongent, are ISO standards [17, 19, 18].

2 MSP430 Architecture and Development Tools

MSP430 is a popular architecture for 16-bit microcontrollers designed by Texas Instruments with the goal of very low power dissipation [30]. Modern MSP430 microcontrollers have up to seven low-power modes and draw less than 1 µA in standby mode, making them an excellent choice for IoT applications that need high energy-efficiency. Even though the MSP430 is described as a 16-bit RISC architecture in [31], it provides some features that are more CISC-like, such as memory-to-memory operations without an intermediate register holding of the operand(s). The MSP430 architecture has 16 registers, each of which is 16 bits long, whereby registers `r0`–`r3` serve a special purpose (program counter, stack pointer, status register, constant generator) and the remaining 12 are available for general use. The instruction set includes 27 core instructions and additional 24 emulated instructions to facilitate Assembly programming. There are seven addressing modes in total; depending on the used mode(s), one and the same instruction can be either 2, 4, or 6 bytes long. The instruction set is orthogonal and allows every instruction to be used with every combination of source and destination addressing mode(s). Memory management is relatively simple since the MSP430 is a "von-Neumann" architecture, i.e., instructions and data share the same address space and are accessed through a single bus. This means, in particular, that every memory read access anywhere in the address space takes one cycle, regardless of whether it targets flash or RAM. Consequently, static look-up tables do not necessarily need to be in RAM to reach top performance (this contrasts with, e.g., Harvard-based ARM microcontrollers where tables in RAM can be accessed faster than tables in flash, even with 0 wait states).

The latency of MSP430 instructions depends on the instruction format and the used addressing mode(s). More concretely, execution time of an instruction is determined by the number of memory accesses needed to fetch the complete instruction and to subsequently read/write the data being processed from/to flash or RAM. For example, an instruction that operates only on registers (and requires only one single memory access to read the instruction itself), such as `mov r5, r8`, executes in a single cycle. An instruction of a length of four bytes whose operands require a read and a write access to memory needs four cycles to execute. For example, `xor r8, 0x1234` takes one cycle to fetch the opcode for xor, one cycle to read the value 0x1234, a further cycle to read the operand stored at the address 0x1234, and finally one cycle to write the result back to the address 0x1234. In general, the number of cycles corresponds to the overall number of memory accesses to be performed. There are only a few exceptions to this simple rule; for example using the `dadd` instruction or using the program counter as destination register costs an additional cycle.

IAR Embedded Workbench for MSP430 is a (proprietary) Integrated Development Environment (IDE) for which a free trial version exists that is limited to a code size of 8 kB [16]. Besides a modern graphical user interface, it comes with a range of tools, including a C++ compiler, a feature-rich debugger, and a cycle-accurate instruction-set simulator. We used the latter to optimize the Assembly code of the underlying primitives and to evaluate the execution time

of the complete AEAD algorithms on a MSP430F1611 microcontroller [31]. In addition, we developed a special simulation script to gain further insights into code execution on our target device. By using snapshots of the memory of the MSP430 and its register state at two different breakpoints, both exported from the Embedded Workbench debugger, our script generates statistics about the instructions that are executed in between those breakpoints. The output of the script provides, for instance, detailed information on:

- Memory usage, e.g., how many different RAM or flash memory addresses have been accessed overall, or how many read accesses and how many write accesses have been performed;
- Instruction execution, e.g., how many times a certain instruction has been executed, how often each addressing mode has been used, how many of the executed instructions are register-to-register, register-to-memory, etc.
- A number of other events, like how many times a certain macro has been executed, or how much stack memory has been used.

This enables us to analyze our implementations of the primitives and to assess to what extent computations are performed locally (i.e., all operands are held in registers), how much memory is occupied by static look-up tables, how many accesses are made to these look-up tables, and so on.

3 Overview of Algorithms and their Implementation

In this section, we describe our implementation of Elephant v2 (the instance Dumbo [5]), Grain128-AEADv2 [15], ISAP v2.0 (concretely Isap-A-128a) [9], the 128-bit instance of PHOTON-Beetle [3], and Romulus-N [12].

3.1 Elephant v2 (Dumbo)

Elephant [5] is a family of Encrypt-then-MAC constructions whose members use a permutation masked with a Linear-Feedback Shift Register (LFSR) in an Even-Mansour-like way [11] instead of a block cipher. Encryption/decryption is performed via counter mode, while message authentication was originally done with a Wegman-Carter-Shoup MAC. However, in the final round, the designers decided to replace this MAC by a variant of the protected counter-sum MAC function, which has the advantage of guaranteeing authenticity in nonce-reuse scenarios. All three Elephant members have a small state size of between 160 and 200 bits, thereby enabling lightweight hardware implementations, and are parallelizable by design. The primary instance for the NIST LWC competition was Dumbo; it is based on the Spongent-π permutation [7], provides 112 bits of security, and is particularly well-suited for hardware implementation.

The Spongent variant used by Dumbo has state of 160 bits and executes 80 rounds. In each round, the following three operations are carried out [7].

- *lCounter:* a 7-bit LFSR defined by the polynomial $p(x) = x^7 + x^6 + 1$.

bit 0 → bit 0	bit 1 → bit 40	bit 2 → bit 80	bit 3 → bit 120
bit 4 → bit 1	bit 5 → bit 41	bit 6 → bit 81	bit 7 → bit 121
bit 8 → bit 2	bit 9 → bit 42	bit 10 → bit 82	bit 11 → bit 122
...	...	...	...
bit 156 → bit 39	bit 157 → bit 79	bit 158 → bit 119	bit 159 → bit 159

Fig. 1. Illustration of the *pLayer* permutation of Spongent.

- *sBoxLayer:* a 4-bit S-box applied 40 times in parallel.
- *pLayer:* a fixed bit-level permutation over the 160-bit state.

To reduce the execution time, we merged the S-box and fixed permutation into a single Assembly macro, so that the values this macro operates on can be kept in a register-set as long as possible and unnecessary memory accesses are avoided. There are several different steps that all require the same operations with respect to rotation and S-box, but there are also some operations that are specific to each step. As illustrated in Fig. 1, the *pLayer* permutation is quite regular, making it possible to sequentially go through the 160 bits and fill up four blocks of 40 bits. This operation is expensive to perform in C, but can be implemented very efficiently in Assembly language, since one can roll the bits one by one from the source register to the carry flag and then successively roll them into four destination registers. In order to keep the binary code size small without compromising performance by using subroutines that are expensive to call (because of operands placed on the stack), we used a technique of dynamic branching, i.e., the execution of the program is controlled by successive loadings of address values directly from a table into the program-counter register `r0`.

Despite our optimization efforts, SPONGENT is the by far slowest of the five considered primitives (see Sect. 4), partly because its design is everything else than "software-friendly" and partly due to the large number of rounds.

3.2 Grain-128AEADv2

GRAIN-128AEAD was the only candidate of the NIST competition that used a stream cipher, namely GRAIN-128A, as its low-level primitive [15]. The initial design was submitted to the eStream project[1] and then tweaked to prevent an attack based on linear approximation (GRAIN V1). Furthermore, the designers proposed a variant (GRAIN-128) with an extended key size of 128 instead of 80 bits and an initialization vector of 96 bits. The GRAIN family made it into the final eStream portfolio for Profile 2 ("stream ciphers for hardware applications with restricted resources") [14]. After the end of the eStream project, it turned out that the 128-bit version has a security flaw, which was fixed in a revision (known as GRAIN-128A) that is not only more secure, but features also native support for authentication with tag sizes up to 32 bits [1].

[1] The eStream project was run by the ECRYPT Network of Excellence from 2004 to 2008 with the goal to identify new stream ciphers suitable for widespread adoption.

The stream cipher on which Grain-128AEAD is based on is (essentially) Grain-128a with some small modifications to allow for a larger authentication tag and to support AEAD. Due to a weakness discovered in the second round of the NIST competition, the initialization of the stream cipher was modified (resulting in Grain-128AEADv2) to better protect against key recovery from a known internal state. This stream cipher is bit-oriented and consists of two major components, namely a so-called *Pre-Output Generator (POGen)* and an *Authenticator Generator (AUGen)*. The former itself is composed of: (i) a 128-bit Linear Feedback Shift Register (LFSR) whose (linear) feedback polynomial and state at time t is denoted $f(x)$ and $S_t = [s_0^t, s_1^t, \ldots, s_{127}^t]$, respectively, (ii) a 128-bit Non-linear Feedback Shift Register (NFSR) with polynomial $g(x)$ and state at time t referred to as $B_t = [b_0^t, b_1^t, \ldots, b_{127}^t]$, and (iii) a simple Boolean function $h(x)$ that takes as input seven bits from the LFSR and two bits from the NFSR. Using these components, the *POGen* generates es a pseudo-random stream of bits through the pre-output function $y_t = h(x) + s_{93}^t + \sum b_j^t$, where $j = \{2, 15, 36, 45, 64, 73, 89\}$. On the other hand, the *AUGen* computes the 64-bit authentication tag and comprises a simple 64-bit shift Register (R) and an Accumulator register (A) of the same length. After initialization, register R is shifted one bit per clock and always contains the 64 most recent *odd bits* from the pre-output bit-stream (more details below). Furthermore, in each clock, the 64 bits of R are ANDed with a single message bit m_i and the 64-bit result is XORed to the content of the accumulator register A.

We developed a variant of the *POGen* in Assembly language that produces 16 bits of pre-output stream following the approach proposed in [20]. Both the LFSR and NFSR are shifted 16 bits in 1-bit steps since the shift instructions of the MSP430 do not support any distances beyond a single bit. We first load the 128-bit LFSR into eight registers and compute (intermediate results of) the terms that form the feedback functions $f(x)$, $g(x)$, and the Boolean function $h(x)$. Thereafter, we load the NFSR and again compute terms of $f(x)$, $g(x)$, as well as $h(x)$. Due to the limited register space, some of the intermediate results have to be stored in RAM, but thanks to the multitude of addressing modes they can be updated very efficiently. While the execution time of the *POGen* is relatively small (just 589 cycles), it has to be considered that only eight of the 16 pre-output bits, namely the eight *even bits*, are actually used for encryption or decryption since the other eight bits are fed into the *AUGen*.

3.3 Isap v2.0

The Isap family of AEAD algorithms [9] is somewhat special amongst the ten LWC finalists in the sense that the main design goal was not efficiency in hardware or software, but to offer strong protection against various implementation attacks such as Differential Power Analysis (DPA) and fault analysis. Isap can be characterized as a family of permutation-based MAC-then-Encrypt schemes consisting of four instances. The primary instance is Isap-A-128a and uses the Ascon-p permutation [10] with 1, 6, or 12 rounds as underlying primitive. All Isap instances feature a mode-level protection against implementation attacks

based on the *fresh re-keying* technique [22], which means encryption/decryption and authentication do not directly process the provided secret key but are performed with session keys that are unique for distinct input data. These session keys are derived from the provided secret key and some public data (either the nonce or a hash of the nonce, associated data, and ciphertext) by a duplex-like re-keying function named *IsapRk*. After initialization with the secret key, the *IsapRk* function absorbs the public data at a rate of only one bit, which limits the number of possible inputs to the permutation's inner part to two per call of the permutation. This, in turn, limits the input data complexity exploitable in DPA such that classical DPA attacks become impractical. Since *IsapRk* uses a very small rate, the number of rounds of Ascon-p was reduced to one. The other two main functions of Isap, namely *IsapEnc* (for encryption/decryption) and *IsapMac* (authentication), operate with higher rates than *IsapRk*, but also execute a larger number of rounds (either six or 12).

An Assembly implementation of the Ascon-p permutation for the MSP430 platform was presented in our previous paper [2]. We used this implementation to determine the execution times of Isap-A-128a given in Sect. 4. A detailed description of our optimizations for Ascon-p can be found in [2].

3.4 Photon-Beetle

PHOTON-Beetle is a family of AEAD and hash algorithms whose members are all based on PHOTON_{256}, a 256-bit permutation designed for efficiency in hardware [3]. The main AEAD instance uses the so-called Beetle mode and operates on top of this permutation with a rate of 128 bits, i.e., the capacity is also 128 bits. In contrast to a conventional duplex construction, where the rate-part of the next input for the permutation is a block of ciphertext (obtained as the direct result of XORing a plaintext-block to the rate), the Beetle mode produces the next input "indirectly" by a COmbined FeedBack (COFB) of the current state and ciphertext-block. More concretely, PHOTON-Beetle uses a linear function ρ, performing a 1-bit right-rotation of a 128-bit word and an XOR, to update the state and encrypt a block of plaintext [3]. In this way, the Beetle mode achieves a security level of $c - \log_2(r)$ bits where c and r denote the size of capacity and rate, respectively (i.e., the security level is only a little below the capacity in bits). Thus, the primary instance of PHOTON-Beetle offers 121-bit security for plaintext confidentiality and ciphertext integrity.

The Photon permutation, introduced at CRYPTO 2011 [13], shows some similarities with the AES, though its columns mixing layer can be computed in a serial way, thereby reducing area cost in hardware. Photon_{256} represents its internal state in the form of 64 elements of four bits, which are arranged as an 8×8 matrix of cells. Each of the 12 rounds performs four basic operations:

- *AddConstants* adds a fixed constant to each cell of the first column.
- *SubCells* applies a 4-bit S-box to each cell of the state.
- *ShiftRows* rotates the cells within each row.
- *MixColumnsSerial* mixes the columns using matrix multiplication.

We used a number of techniques to optimize our Assembly implementation of PHOTON$_{256}$ for the MSP430. A naive implementation of the four operations is quite slow, but as the designer's reference implementation already shows, the three expensive computations *SubCells*, *ShiftRows*, and *MixColumnsSerial* can be significantly accelerated using a pre-computed look-up table (referred to as SCShRMCS table). We use a table of $8 \times 12 = 96$ bytes for the round constants and a table of $16 \times 8 \times 8 = 1024$ bytes for the other three operations. Since the look-ups into the SCShRMCS table require a multiplication of the state values by 8, it is possible to achieve a speed-up by pre-multiplying the state by 8 and adapting the operations accordingly (e.g., all the round constants are also pre-multiplied by 8 and so on). Only at the very end of the permutation, the state values are divided by 8 so as to bring them back to the expected range. At the beginning of each of the 12 rounds, we copy the state to a temporary area in RAM, and we combine this copying with the round-key addition to reduce the number of loads and stores. Our general optimization strategy was to minimize the number of memory accesses by keeping state-values as much as possible in the 12 (unrestricted) MSP430 registers. During the execution of the main loop of the permutation, one line of the matrix (containing eight values) is kept in four registers, which are successively updated. We use another four registers to store pointers to the tables and the state matrix, respectively. Two out of the four remaining registers serve as counters and as temporary variable, while the other two hold constants that are frequently used.

3.5 Romulus v1.3

ROMULUS [12] is a family of AEAD algorithms based on the Tweakable Block Cipher (TBC) SKINNY, which was introduced at CRYPTO 2016 [4]. The main instance, ROMULUS-N, uses a rate-1 TBC-based COmbined FeedBack (COFB) mode of operation. It encrypts a 16-byte block of plaintext with one invocation of the TBC and also needs just one TBC-call for a 32-byte block of associated data. However, the processing of associated data involves a relatively expensive tweakey-schedule per block. The main benefits of ROMULUS-N are efficiency in hardware and extremely low overhead for short messages since there is no pre-processing TBC-call. For example, authenticating 16 bytes associated data and encrypting/decrypting 16 bytes plaintext/ciphertext can be done with just two TBC-calls altogether. Plaintext is added to the cipher-state via a state-update function ρ performing a multiplication by a binary matrix, which boils down to a 1-bit right-rotation of each state-byte and an XOR. The computation of the tweak mainly consists of updating a 56-bit counter (initially set to 0) by means of an LFSR with polynomial $f(x) = x^{56} + x^7 + x^4 + x^2 + 1$.

The SKINNY variant used by ROMULUS-N is SKINNY-128-384+; it has, as indicated by its name, a block size of 128 bits, a tweakey size of 384 bits, and iterates its round function 40 times. This round function represents the cipher-state as a 4×4 array of 8-bit cells and performs five relatively simple AES-like operations: *SubCells* applies an 8-bit S-box to every cell, *AddConstants* XORs round-dependent constants (generated using a 6-bit LFSR) to the first column

Table 2. Implementation results for the underlying primitives of (in this order) Elephant (Dumbo), Grain-128AEADv2, ISAP v2.0, PHOTON-Beetle and Romulus-N.

Underlying primitive in Assembly	Asm function(s)		Throughput		Static table look-ups	
	Size (bytes)	Time (cycles)	Encr. bytes	Cycles per byte	Table size (bytes)	Look-ups (cycles)
Spongent-π[160]	822	40495	20	2025	428 (52%)	2562 (6%)
Grain-128A	916+144	584+287	1	871	0	0
Ascon-p^6	710	3520	8	440	0	0
PHOTON$_{256}$	1612	17034	16	1065	1120 (69%)	6240 (37%)
Skinny-128-384+	782	5490	16	343	256 (33%)	1280 (23%)

of the state-array, *AddTweakey* XORs the first two rows of the tweakey arrays to the state and then updates the tweakeys (see below), *ShiftRows* rotates the second, third, and fourth row of the state cell-wise, and *MixColumns* multiplies each column of the state-array by a binary matrix [4]. SKINNY-128-384+ uses three 128-bit tweakeys, which are represented through three 4×4 arrays of cells (referred to as *TK1*, *TK2*, and *TK3*). The three tweakey-states are updated in every round via a cell-permutation (i.e., the 16 cells of the tweakey-matrix are re-ordered) and, thereafter, the cells of the first two rows of *TK2* and *TK3* are individually manipulated, treating them as 8-bit LFSRs.

The designer team provided some optimized implementations of ROMULUS-N, among these is a version using a 256-byte S-box table and one based on the so-called fixed-slicing method [12]. Our implementation for MSP430 follows the former approach and comprises four functions altogether; three to pre-compute the round-tweakeyes, and one for the encryption of a 16-byte block, which gets the three expanded tweakeys as input. We developed optimized Assembly code for the block-encryption and the tweakey schedule for *TK2*, the latter to speed up the processing of associated data. We did not implement the *TK1* schedule in Assembly since it is very light (the last 64 bits of *TK1* are always 0 and the tweakeys repeat after 16 rounds) and relatively fast in C. Also the *TK3* schedule exists only in C because it is executed just once. Our Assembly implementation of the encryption function keeps the full 16-byte state in registers and merges *SubCells* and *ShiftRows* to a single macro, taking advantage of the swap-bytes (`swpb`) instruction. On the other hand, our *TK2*-schedule function utilizes the bit-test (`bit`) instruction to efficiently execute the 8-bit LFSR.

4 Performance Evaluation and Comparison

As explained in Sect. 2, we used IAR Embedded Workbench 7.2 to develop the Assembly implementations of the underlying primitives of the AEAD schemes and determined execution times (and other results) using IAR's cycle-accurate instruction set simulator and our specifically developed Python script.

Table 2 summarizes some basic size and performance characteristics of the Assembly functions of the five primitives, assuming that they are used for the

Table 3. Execution time (in cycles) of the AEAD algorithms for authentication only (dlen = 0), encryption only (adlen = 0), and authenticated encryption (adlen = dlen).

adlen	dlen	Elephant	Grain	ISAP	PHOTON	Romulus
16	0	166795	36044	157330	35494	25793
128	0	417230	142203	269556	157254	70171
1024	0	2254717	984501	1167124	1131094	381894
0	16	167203	36140	277555	36212	26064
0	128	668652	142007	448833	162830	71706
0	1024	4430480	988727	1818817	1175534	436601
16	16	208684	51165	293583	53676	26127
128	128	958893	263164	577058	302025	116118
1024	1024	6545252	1952182	2844610	2288569	792736

processing of plaintext, i.e., encryption. The Assembly component of the three permutation-based AEAD schemes (Elephant, Isap, PHOTON-Beetle) is simply the function for permuting the state. Isap is somewhat special because it executes Ascon-p with either one, six, or even 12 rounds, but *IsapEnc* calls the permutation-function with six rounds only. Grain-128a has two functions written in Assembly language, namely the *POGen* and *AUGen*. The execution time of both is given in Table 2, whereby the cycle-count for the former refers to the generation of a 16-bit pre-output stream and the cycles for the latter to the authentication of eight bits. Finally, for Skinny, the table lists the results of the function for encrypting a 16-byte block, which receives the three round-tweakeys as input. This function, as well as the *TK2*-schedule, are written in Assembly language, but the latter is only used for the processing of associated data and, therefore, omitted from Table 2. Also not included in the table is the *TK1*-schedule, even though it is part of the main encryption loop, since it takes only 521 clock cycles in C and so we did not bother to write it in Assembly.

The results in Table 2 show that four of the five implemented primitives are very compact and have a binary code size of only about 1 kB (including static look-up tables). PHOTON$_{256}$ is a bit larger, but roughly two thirds of its code space are occupied by static tables. The execution time of the primitives varies by a much greater extent than their code size. Grain-128a has an execution time of less than 1000 clock cycles (sum of the cycle counts of the *POGen* and *AUGen*), whereas Spongent, which is at the other end of the spectrum, needs more than 40000 cycles for a single permutation. However, the number of bytes processed by each primitive while encrypting plaintext varies significantly too and ranges from one byte to 20 bytes. Consequently, it makes sense to compare the (inverse) throughput values, obtained by dividing each cycle-count by the corresponding number of bytes. The resulting cycles per byte, included in the table, indicate that Romulus is the fastest AEAD scheme for encryption since its underlying primitive has the best (i.e., smallest) cycle-per-byte value.

Table 3 contains the execution times of the five AEAD algorithms in three different case studies: (i) authentication of associated data (without encryption

Table 4. Execution times of the reference and optimized C implementations from [6] and our mixed C-and-Assembly implementations of the five AEAD algorithms for 16 bytes of associated data and 16 bytes of plaintext.

AEAD Algorithm	Ref. C [6] (cycles)	Opt. C [6] (cycles)	Our impl. (cycles)	Speed-up factor for our impl.
Elephant (Dumbo)	17073111	–	208684	81.81×
Grain-128AEAD	3601170	244766	51165	70.38× / 4.78×
ISAP-A-128A	8109430	–	293583	27.62×
PHOTON-Beetle	1125425	–	53676	20.97×
Romulus-N	187504	–	26127	7.17×

of plaintext, i.e., dlen = 0), (ii) encryption of plaintext (without authentication of associated data, i.e., adlen = 0), (iii) authenticated encryption (associated data and plaintext of the same size are processed). For each use case, we give the execution time for short (i.e., 16-byte), medium (i.e., 128-byte) and large (i.e., 1024-byte) amounts of associated data and plaintext, respectively. When inspecting the results of each algorithm individually, it turns out that Grain-128AEAD and PHOTON-Beetle have roughly the same execution times in the "authentication-only" and "encryption-only" scenario, respectively. On the other hand, Elephant and ISAP-A-128A process plaintext much slower than associated data, which is due to the Encrypt-then-MAC mode. Namely, when using this mode, a plaintext block requires two executions of the primitive, one to produce the ciphertext block and the other to authenticate it. Romulus is slightly slower in the "encryption-only" scenario since it needs one invocation of the block-encryption function for 16 plaintext bytes, while authenticating 32 bytes of associated data requires also one call of the block-encryption function and one call of the *TK2*-schedule, the latter being 670 cycles faster. When we compare the five schemes with each other, Romulus is the clear winner since it reaches the best execution times in all three use cases and across all operand lengths, which is not surprising given the throughput of Skinny. Romulus is even faster than the LWC winner Ascon; for example, when adlen = dlen = 1024, Ascon has, according to [2], an execution time of 941924 clock cycles on a MSP430F1611, i.e., almost 150000 cycles more than Romulus. Elephant is the worst of the five schemes, partly because of the poor software performance of Spongent and partly due to its relatively inefficient mode.

Table 4 compares the execution time of our five Assembly-optimized AEAD schemes with that of the "plain" C implementations benchmarked in [6]. These timings are for an authenticated encryption of 16 bytes of associated data and 16 bytes of plaintext. Even though we optimized only the underlying primitive at the Assembly level, all five AEAD algorithms experienced a massive speed-up (e.g., nearly two orders of magnitude for Elephant). These results confirm that reference C implementations are not suitable to evaluate the performance of an AEAD algorithm. They also show that the performance of the algorithms depends heavily on the efficient implementation of the underlying primitive.

5 Concluding Remarks

The recently concluded LWC standardization project of the NIST has spurred a large body of research on "lightweight" AEAD algorithms that are suitable for resource-restricted devices. However, since the NIST focussed their software benchmarking primarily on 8 and 32-bit architectures, many questions related to the efficiency of the final-round candidates on 16-bit microcontrollers have remained largely unexplored. We contributed to fill this gap by developing and benchmarking optimized implementations of the final-round LWC candidates Elephant (i.e., Dumbo), Grain-128AEAD, ISAP, PHOTON-Beetle, and Romulus for 16-bit MSP430 microcontrollers. More concretely, we developed highly-tuned MSP430 Assembly implementations of the underlying primitives of these five AEAD schemes and analyzed their efficiency with a cycle-accurate simulator as well as a special Python tool. Our results show that Skinny-128-384+ (part of Romulus-N) has the best throughput of all five primitives. The outstanding efficiency of Skinny contributes to the superior execution times of Romulus-N, which outperforms its competitors by a factor of at least two when the length of the input(s) exceeds 128 bytes. Remarkably, Romulus-N is even faster than the LWC winner Ascon, despite the fact that software speed was not the main objective of the designers. Our work also shows that the five AEAD algorithms are much faster (by a factor of between 4.78 and 81.81) on MSP430 microcontrollers than indicated by previously-published results.

Acknowledgments. The second author was supported, in part, by the Luxembourg National Research Fund (FNR) under CORE grant C19/IS/13641232 (APLICA).

References

1. M. Ågren, M. Hell, T. Johansson, and W. Meier. A new version of Grain-128 with authentication. In G. Leander and S. S. Thomsen, editors, *Proceedings of the 6th ECRYPT Workshop on Symmetric Encryption (SKEW 2011)*. Available for download at `http://skew2011.mat.dtu.dk/proceedings/A%20New%20Version%20of%20Grain-128%20with%20Authentication.pdf`, 2011.
2. M. Alsahli, A. Borgognoni, L. Cardoso dos Santos, H. Cheng, C. Franck, and J. Großschädl. Lightweight permutation-based cryptography for the ultra-low-power Internet of things. In G. Bella, M. Doinea, and H. Janicke, editors, *Innovative Security Solutions for Information Technology and Communications — SecITC 2022*, volume 13809 of *Lecture Notes in Computer Science*, pages 17–36. Springer Verlag, 2022.
3. Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. PHOTON-Beetle authenticated encryption and hash family. Specification, available for download at `http://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf`, 2021.
4. C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In M. Robshaw and J. Katz, editors, *Advances in Cryptology —*

CRYPTO 2016, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer Verlag, 2016.

5. T. Beyne, Y. L. Chen, C. Dobraunig, and B. Mennink. Dumbo, Jumbo, and Delirium: Parallel authenticated encryption for the lightweight circus. *IACR Transactions on Symmetric Cryptology*, 2020(S1):5–30, June 2020.
6. S. Blanc, A. Lahmadi, K. Le Gouguec, M. Minier, and L. Sleem. Benchmarking of lightweight cryptographic algorithms for wireless IoT networks. *Wireless Networks*, 28(8):3453–3476, Nov. 2022.
7. A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede. SPONGENT: A lightweight hash function. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 312–325. Springer Verlag, 2011.
8. L. Cardoso dos Santos and J. Großschädl. An evaluation of the multi-platform efficiency of lightweight cryptographic permutations. In P. Y. A. Ryan and C. Toma, editors, *Innovative Security Solutions for Information Technology and Communications — SecITC 2021*, volume 13195 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 2022.
9. C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. Isap v2.0. *IACR Transactions on Symmetric Cryptology*, 2020(S1):390–416, June 2020.
10. C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, July 2021.
11. R. Granger, P. Jovanovic, B. Mennink, and S. Neves. Improved masking for tweakable blockciphers with applications to authenticated encryption. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology — EUROCRYPT 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 263–293. Springer Verlag, 2016.
12. C. Guo, T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin. Romulus v1.3. Specification, available for download at `http://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf`, 2021.
13. J. Guo, T. Peyrin, and A. Poschmann. The PHOTON family of lightweight hash functions. In P. Rogaway, editor, *Advances in Cryptology — CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer Verlag, 2011.
14. M. Hell, T. Johansson, A. Maximov, and W. Meier. The Grain family of stream ciphers. In M. J. Robshaw and O. Billet, editors, *New Stream Cipher Designs – The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 179–190. Springer Verlag, 2008.
15. M. Hell, T. Johansson, A. Maximov, W. Meier, J. Sönnerup, and H. Yoshida. Grain-128AEADv2 – A lightweight AEAD stream cipher. Specification, available for download at `http://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf`, 2021.
16. IAR Systems AB. IAR Embedded Workbench for MSP430. Product description, available online at `http://www.iar.com/iar-embedded-workbench/msp430`, 2023.
17. International Organization for Standardization (ISO). ISO/IEC 29167–13:2015 Information technology — Automatic identification and data capture techniques — Part 13: Crypto suite Grain-128A security services for air interface communications, 2015.

18. International Organization for Standardization (ISO). ISO/IEC 29192–5:2016 Information technology — Security techniques Lightweight cryptography — Part 5: Hash-functions, 2016.
19. International Organization for Standardization (ISO). ISO/IEC 18033–7:2022 Information technology — Encryption algorithms — Part 7: Tweakable block ciphers, 2022.
20. A. Maximov and M. Hell. Software evaluation of Grain-128AEAD for embedded platforms. Cryptology ePrint Archive, Report 2020/659, 2020. Available for download at `http://eprint.iacr.org`.
21. K. A. McKay, L. Bassham, M. Sönmez Turan, and N. Mouha. Report on lightweight cryptography. Technical Report IR 8114, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2017. Available for download at `http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf`.
22. M. Medwed, F.-X. Standaert, J. Großschädl, and F. Regazzoni. Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In D. J. Bernstein and T. Lange, editors, *Progress in Cryptology — AFRICACRYPT 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer Verlag, 2010.
23. National Institute of Standards and Technology (NIST). Announcing request for nominations for lightweight cryptographic algorithms. Federal register notice, available online at `http://csrc.nist.gov/news/2018/requesting-nominations-for-lightweight-crypto-algs`, 2018.
24. National Institute of Standards and Technology (NIST). Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. Available for download at `http://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf`, 2018.
25. National Institute of Standards and Technology (NIST). Benchmarking of lightweight cryptographic algorithms on microcontrollers. Available online at `http://github.com/usnistgov/Lightweight-Cryptography-Benchmarking`, 2023.
26. S. Renner, E. Pozzobon, and J. Mottok. The final round: Benchmarking NIST LWC ciphers on microcontrollers. In W. Li, S. Furnell, and W. Meng, editors, *Attacks and Defenses for the Internet-of-Things — ADIoT 2022*, volume 13745 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, 2022.
27. S. Renner, E. Pozzobon, and J. Mottok. NIST LWC software performance benchmarks on microcontrollers. Available online at `http://lwc.las3.de`, 2022.
28. P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, pages 98–107. ACM Press, 2002.
29. M. Sönmez Turan, K. A. McKay, D. Chang, L. E. Bassham, J. Kang, N. D. Waller, J. M. Kelsey, and D. Hong. Status report on the final round of the NIST lightweight cryptography standardization process. Internal Report IR 8454, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2023. Available for download at `http://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8454.pdf`.
30. Texas Instruments Inc. MSP430 Family Architecture Guide and Module Library. TI literature number SLAUE10B, available for download at `http://www.ti.com/sc/docs/products/micro/msp430/userguid/ag_01.pdf`, 1996.
31. Texas Instruments, Inc. MSP430x1xx Family User's Guide (Rev. F). Manual, available for download at `http://www.ti.com/lit/ug/slau049f/slau049f.pdf`, Feb. 2006.
32. R. Weatherley. Lightweight cryptography primitives documentation. Available online at `http://rweather.github.io/lwc-finalists/index.html`, 2021.