Lightweight Permutation-Based Cryptography for the Ultra-Low-Power Internet of Things

Malik Alsahli, Alex Borgognoni, Luan Cardoso dos Santos, Hao Cheng, Christian Franck, and Johann Großschädl

DCS and SnT, University of Luxembourg,
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{malik.alsahli.001,alex.borgognoni.001}@student.uni.lu
{luan.cardoso,hao.cheng,christian.franck,johann.groszschaedl}@uni.lu

Abstract. The U.S. National Institute of Standards and Technology is currently undertaking a process to evaluate and eventually standardize one or more "lightweight" algorithms for authenticated encryption and hashing that are suitable for resource-restricted devices. In addition to security, this process takes into account the efficiency of the candidate algorithms in various hardware environments (e.g. FPGAs, ASICs) and software platforms (e.g. 8, 16, 32-bit microcontrollers). However, while there exist numerous detailed benchmarking results for 8-bit AVR and 32-bit ARM/RISC-V/ESP32 microcontrollers, relatively little is known about the candidates' efficiency on 16-bit platforms. In order to fill this gap, we present a performance evaluation of the final-round candidates Ascon, Schwaemm, TinyJambu, and Xoodyak on the MSP430 series of ultra-low-power 16-bit microcontrollers from Texas Instruments. All four algorithms were explicitly designed to achieve high performance in software and have further in common that the underlying primitive is a permutation. We discuss how these permutations can be implemented efficiently in Assembly language and analyze how basic design decisions impact their execution time on the MSP430 architecture. Our results show that, overall, Schwaemm is the fastest algorithm across various lengths of data and associated data, respectively. Xoodyak has benefits when a large amount of associated data is to be authenticated, whereas TinyJambu is very efficient for the authentication of short messages.

1 Introduction

The emergence and rise of *cryptographic permutations* is widely seen one of the most exciting developments in the field of symmetric cryptography during the past 20 years. Formally, a cryptographic permutation is defined as a bijective mapping within $\mathbb{Z}_2^b$ (the bitstrings of length b), designed to behave as a random permutation, i.e. a permutation drawn uniformly at random from the set of all possible permutations that operate on b bits [5]. The width b of a permutation can range from 100 (e.g. Photon [17] and other cryptosystems that target the embedded domain) to 1600 (e.g. Keccak [6]). Permutations are highly flexible and universally-applicable primitives, similar to block ciphers, and can be used

to construct e.g. hash functions, message authentication codes, pseudo-random bit-sequence generators, stream ciphers, and even algorithms for authenticated encryption [3, 5, 7]. However, in contrast to a block cipher, a permutation is an unkeyed primitive, i.e. it does not use any key and, therefore, does not have to perform a key schedule. Another difference is that a cryptographic permutation is usually designed to be efficient only in the forward direction since the inverse permutation is (normally) not needed. In recent years, permutations have also served as building block for the design of "advanced modes" that cover the full functionality of the symmetric portion of modern security protocols. Examples for this relatively new line of research include Blinker [26], the Strobe protocol framework [18], and Stateful Hash Objects (SHO) [23].

Permutations are especially suitable for *lightweight cryptography*, which can be very generally defined as "cryptographic primitives, schemes, and protocols tailored to (extremely) constrained environments" [16]. Examples of such environments include RFID tags, miniature sensors and actuators, and numerous other kinds of devices that form part of the *Internet of Things (IoT)* [32]. The U.S. National Institute of Standards and Technology (NIST) is currently in the process of standardizing lightweight cryptosystems, in particular cryptographic hash functions and algorithms for Authenticated Encryption with Associated Data (AEAD) [20]. Permutation-based designs perform extremely well in this standardization, which is evidenced by the fact that 16 out of 32 second-round candidates, and four out of the ten candidates in the third and final round, use a permutation as low-level primitive [22]. The four permutation-based designs in the (currently still ongoing) final round of NIST's standardization effort are Ascon [15], Sparkle [2], TinyJambu [31], and Xoodyak [12]. However, the finalist TinyJambu is a special case since it uses a keyed permutation and can also be classified as a block-cipher-based design (like in [22]). The width of the permutations ranges from 128 bits (TinyJambu) over 320 bits (Ascon) up to 384 bits (Sparkle384, Xoodyak). Sparkle is a classical Addition-Rotation-XOR (ARX) design, while the other three permutations may be categorized as "AndRX" variants, i.e. they generate non-linearity via logical AND operations instead of modular additions.

The evaluation of candidates for NIST's lightweight cryptography standard takes into account a number of criteria, among which security and performance on software and hardware platforms are particularly important [22]. Regarding software performance, the official NIST document on submission requirements advised the algorithm designers to "consider a wide range of 8-bit, 16-bit, and 32-bit microcontroller architectures" [20, Sect. 3.4]. For most of the final-round candidates, optimized implementations with highly-tuned Assembly segments for the performance-critical parts have been developed for 8-bit and 32-bit platforms, most notably the AVR ATmega [19] and ARM Cortex-M3/M4 [1] series of microcontrollers. These Assembly implementations either come directly from the designers or have been contributed by other developers [30]. Hence, there exist now a large number of implementation results for these two platforms, in particular execution time and binary code size. Detailed benchmarking results

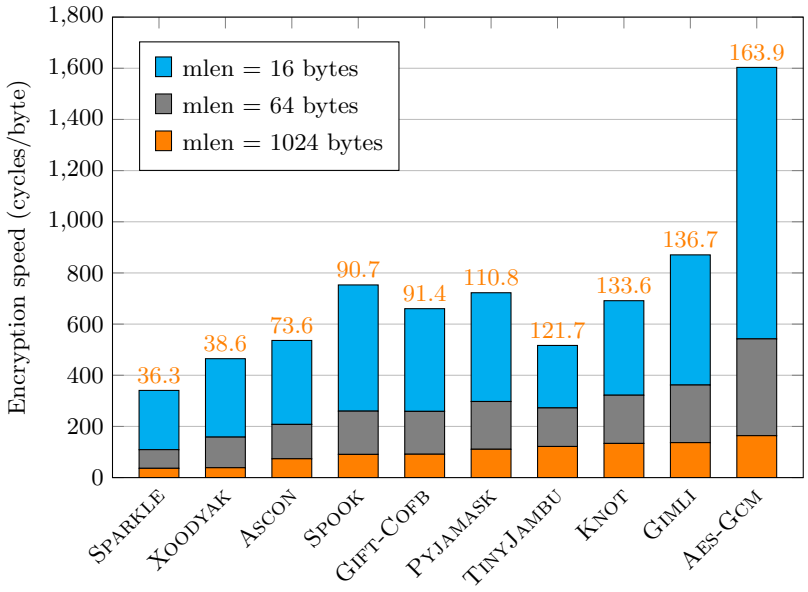

Fig. 1. Comparison of the ten fastest second-round AEAD candidates for encryption of a message with a length of 16, 64, and 1024 bytes (without associated data) on an ARM Cortex-M4F microcontroller. The value above each bar is the encryption speed (in cycles per byte) for a 1024-byte message. For each candidate, the implementation with the best encryption time for 1024 bytes was chosen.

have been published by the NIST lightweight cryptography team [21] and some academic research groups, see e.g. [24]. The four permutation-based algorithms are highly efficient in software; for example, Sparkle, Xoodyak, and Ascon take the top three positions on ARM Cortex-M4F according to NIST's official second-round benchmarking results[1], see Fig. 1. While the efficiency of the ten finalists on 8-bit and 32-bit architectures is well understood, relatively little is known about their performance and binary code size on 16-bit platforms. The only relevant paper we became aware of was published very recently by Blanc et al. [8], who benchmarked reference and optimized C implementations of the final-round candidates on a 16-bit MSP430F1611 microcontroller.

The 16-bit MSP430 platform from Texas Instruments is a particularly interesting target for the benchmarking of lightweight cryptosystems, mainly due to two reasons. First, MSP430 microcontrollers were from the ground up designed with the goal of low power dissipation, taking into account not only the active processing power, but also power in stand-by (resp. sleep) mode, which makes them ideal for many kinds of battery-operated devices, e.g. miniature wireless sensor nodes [14]. Recent members of the MSP430 family support up to seven

[1] At the time of writing this paper, the third (i.e. final) round of evaluation was still going on and NIST had not yet released the round-3 benchmarking results.

different *low-power modes* with fine-grain control over active components and instant wake-up thanks to a sophisticated clock system. Furthermore, MSP430 microcontrollers were among the first mass-market IoT platforms that became equipped with *Ferro-electric Random Access Memory (FRAM)*, a non-volatile form of memory combining properties of SRAM with properties of flash within a single memory space, which can be flexibly (re)configured to serve as storage for program or data [25]. More concretely, FRAM features relatively fast write accesses, low power consumption, and extremely high reliability and endurance (similar to SRAM), but is non-volatile and, thus, able to hold its content when being powered off. However, in contrast to flash and EEPROM, FRAM does not need high supply voltages for write operations, which is a major advantage for e.g. data-logging applications. Furthermore, FRAM makes it easy to switch from active to sleep mode and vice versa, thereby enabling energy savings even for short periods of inactivity. Texas Instruments markets the MSP430 line as "ultra-low-power" microcontrollers [29] to emphasize their potency for battery-operated devices. The fact that such devices are widely used in security-critical applications (e.g. sensors for medical monitoring) makes a strong case to assess the performance of the NIST finalists under ultra-low-power regimes.

A second reason as to why MSP430 microcontrollers are an interesting platform for the benchmarking of NIST's candidate algorithms relates to the basic characteristics of the underlying instruction set architecture. The MSP430 is, in essence, a CISC-like memory-to-memory architecture [27], whereas virtually all other benchmarking platforms (especially AVR and ARM) are more RISC-like and based on the load/store paradigm. All data processing instructions of the MSP430 architecture do not necessarily need to have the operands in registers but can also operate directly on data held in memory (without an intermediate register holding) [28]. This contrasts with RISC architectures, where operands have to be first loaded from memory to registers before an instruction can be executed on them. To a certain extent, the ability to directly process data in memory compensates for the (relatively) limited register space of the MSP430 architecture[2]. It is exactly these architectural differences that are interesting in the context of benchmarking. Namely, as argued in [4, 9], a lightweight cryptographic algorithm should be fast on a broad range of microcontroller platforms with highly diverse and even divergent characteristics. Collecting benchmarks on a (somewhat) CISC-based architecture like the MSP430 makes sense since the current portfolio of benchmarking platforms is solely RISC-based and does not represent the high diversity of microcontrollers in the IoT.

In this paper, we analyze and compare the performance of the permutation-based AEAD algorithms Ascon, Schwaemm, TinyJambu, and Xoodyak on a 16-bit MSP430F1611 microcontroller. However, in contrast to the recent work of Blanc et al. [8], we use carefully-optimized Assembly implementations of the underlying permutations for our evaluation. We developed all implementations

[2] Out of the total of 16 general-purpose registers, only 12 can actually be used by the programmer, which means the usable register space of MSP430 microcontrollers is even smaller than that of the 8-bit AVR architecture (192 vs. 256 bits).

from scratch, whereby we aimed for a reasonable trade-off between execution time and code size. Furthermore, we do not only report benchmarking results of the four algorithms for different lengths of associated data and data, but we also aim to analyze and explain *why* the algorithms perform differently on the MSP430 platform. More concretely, we study how basic design decisions of the underlying permutation, such as the *rotation distances* or the *locality* (i.e. the ability to operate on only a part of the state at a time[3]) affect their execution time. To this end, we developed a special tool that is able to simulate MSP430 instructions and gather detailed information about the execution profile of the permutations, e.g. the number of memory accesses. We use this information to compare the (relative) amount of register-to-register operations for each of the permutations, the proportions of clock cycles they spent for rotations and non-linear operations, as well as their throughput in terms of cycles per state-byte and per rate-byte, respectively. We observed significant differences in execution time, not only for the permutations but also for the full AEAD schemes. When taking different lengths of associated data and plaintext (resp. ciphertext) into account, Schwaemm is the best overall performer, mainly because it combines a well-optimizable permutation with an efficient mode of operation.

2 MSP430 Architecture

The MSP430 architecture uses the von-Neumann memory model, which means instructions (i.e. code) and data share a unified address space. There is a single address bus and a single data bus connecting the microcontroller core with the RAM, non-volatile memory (flash or FRAM), and peripheral modules. MSP430 microcontrollers have a total of 16 registers, each 16 bits wide, of which 12 are general working registers, and the remaining four serve a special purpose: `r0` is the program counter, `r1` is the stack pointer, `r2` is a status register, and `r3` is used to generate common constants like -1, 0, 1, 2, 4, 8. The instruction set is rather minimalist and consists of only 27 core instructions that can be divided into three categories: double-operand instructions (which overwrite one of the operands with the result), single-operand instructions, and jumps. Most of the instructions can not only operate on 16-bit operands, but also on bytes (more concretely, the lower bytes of 16-bit operands) when the instruction is suffixed by `.b`. The instruction set is orthogonal and supports seven addressing modes altogether, including modes for direct memory-to-memory transfers without an intermediate register holding [28]. Depending on the addressing mode(s), the latency of double-operand instructions can vary between one clock cycle (when both source and destination operand are held in registers) and six clock cycles (when operands and result are in RAM or non-volatile memory).

[3] As argued in [4], the ability to work locally (i.e. on a part of the state at a time) is an important design criterion to achieve good efficiency on microcontrollers whose register space is too small to store the full state (high locality reduces the need to move state-words between registers and RAM). However, efficiency desiderata like locality have to be carefully balanced with security desiderata like diffusion.

As explained in the last section, the MSP430 architecture is more CISC-like than e.g. AVR or ARM since it allows one to execute instructions on operands held in RAM or flash without intermediate register holding. For example, the instruction `add.w @r4+, 8(r5)` adds two 16-bit words, whereby register `r4` and `r5` contain the addresses of the operands (resp. result) instead of their actual values. More precisely, the first operand is accessed through the indirect auto-increment addressing mode, which means the value in `r4` is a pointer that gets automatically incremented by 2 after the 16-bit word at the target address has been fetched. On the other hand, the effective address of the second operand (and also of the result) is obtained using the indexed addressing mode, i.e. it is the sum of the base address contained in register `r5` and the offset of 8 (note that in MSP430 assembly language, the destination of an instruction is always on the right side). Consequently, two loads, an addition, and a store operation are combined into a single memory-to-memory instruction, which (potentially) saves not only code space but also execution time. On a RISC architecture like ARM, such a sequence of operations requires four separate instructions in the best case, and up to twice as much under register pressure. Namely, when all registers are occupied, two registers need to be spilled to free up space for the operands, which costs two push and two pop instructions. To some extent, the ability to execute memory-to-memory instructions compensates for the limited register capacity of the MSP430 architecture. However, since memory accesses generally increase the latency of instructions, finding a good register allocation is still very important to reach high performance.

Shifts or rotations of either 32-bit words or 64-bit words are essential operations of the four permutations we consider in this paper. However, contrary to their ARM counterparts, MSP430 microcontrollers do not feature a fast barrel shifter that would allow them to shift or rotate a 16-bit operand by several bits at a time. Therefore, multi-bit shifts/rotates have to be composed of the single-bit shift and rotate instructions supported by the MSP430 architecture; these are `rla.w` and `rra.w` for arithmetic shifts, and `rlc.w` and `rrc.w` for rotations via carry [27]. The execution time of shifts/rotations of 32-bit or 64-bit words depends heavily on the shift/rotation distance, whereby the best possible case is a distance of (a multiple of) 16 bits. Rotating a 32-bit or 64-bit word stored in registers by 16 bits is usually free since it only requires adapting the order in which the 16-bit parts are accessed in a subsequent operation. For example, an operation of the form $a = a \oplus (b \ggg 16)$, where a and b are 32-bit words in the register pairs `r4,r5` and `r6,r7`, respectively, takes only two `xor.w` instructions since the 16-bit rotation of b can be carried out *implicitly*: `xor.w r7, r4` and `xor.w r6, r5`. When a and b are 64-bit words, shifts or rotations by a multiple of 16 bits, i.e. 16, 32, and 48 bits, can be performed implicitly.

The second-fastest shift/rotation distances, after (multiples of) 16 bits, are the ones that are close to multiples of 16 bits, e.g. 1, 15, 17, and 31 bits for 32-bit words. Shifting a 32-bit word held in two registers by one of these distances requires two instructions and takes two cycles, independent of the direction. An additional instruction is necessary for a rotation, whereby again the direction

Listing 1. Macro for 1-bit left-rotation of a 32-bit word.

```
1: QROL macro a0, a1
2:     rla.w   a0
3:     rlc.w   a1
4:     adc.w   a0
5:     endm
```

Listing 2. Macro for 1-bit right-rotation of a 32-bit word.

```
1: QROR macro a0, a1
2:     bit.w   #1, a0
3:     rrc.w   a1
4:     rrc.w   a0
5:     endm
```

Listing 3. Macro for 8-bit left-rotation of a 32-bit word (`tr` is a scratch register).

```
1: QROL8 macro a0, a1
2:     swpb    a0
3:     swpb    a1
4:     mov.b   a0, tr
5:     xor.b   a1, tr
6:     xor.w   tr, a0
7:     xor.w   tr, a1
8:     endm
```

Listing 4. Macro for 8-bit right-rotation of a 32-bit word (`tr` is a scratch register).

```
1: QROR8 macro a0, a1
2:     mov.b   a0, tr
3:     xor.b   a1, tr
4:     xor.w   tr, a0
5:     xor.w   tr, a1
6:     swpb    a0
7:     swpb    a1
8:     endm
```

does not matter, i.e. a right-rotation needs the same number of cycles as a left-rotation. Listing 1 and 2 contain Assembly macros (based on directives of the IAR assembler) to rotate a 32-bit word held in registers one bit to the left and to the right, respectively. A rotation by a distance of more than one bit can be composed of these two macros, which confirms the importance of choosing the rotation distances carefully since e.g. a rotation by three bits already costs 12 cycles. However, thanks to the swap-byte instruction `swpb`, a "shortcut" exists for 8-bit left and right rotation as shown in Listing 3 and 4, respectively. These macros use byte-wise instructions with the `.b` suffix that only operate on the lower byte of a 16-bit register and set its upper byte to 0. Since the execution time of both macros is only six cycles, they can accelerate rotations by certain distances through a decomposition into 8-bit and 1-bit steps (e.g. a 7-bit right-rotation can be performed by first rotating eight bits right and then one bit to the left). Table 1 summarizes the execution time of optimized implementations of rotations by distances between 1 and 15 bits. As explained earlier, a rotation by a multiple of 16 bits is normally free (up to register-reordering). A rotation by distances of $n > 16$ bits can always be reduced to a (n mod 16)-bit rotation along with an implicit register-reordering in a subsequent operation.

Table 1. Execution time (in clock cycles) for a rotation of a 32-bit and a 64-bit word over a distance from 1 to 15 bits.

Rotation distance (bits)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Time to rotate a 32-bit word	3	6	9	12	15	12	9	6	9	12	15	12	9	6	3
Time to rotate a 64-bit word	5	10	15	20	25	26	21	16	21	26	25	20	15	10	5

3 Overview of the AEAD Algorithms

In this section, we overview the main properties of the four AEAD algorithms we consider in this paper, namely Ascon, Schwaemm, TinyJambu v2, and Xoodyak. They all reached the final round of NIST's lightweight cryptography standardization project [22] and are well suited for small microcontrollers.

ASCON. Ascon is not only one of the 10 finalists of NIST's standardization project in lightweight cryptography, but was also selected for the final portfolio of the CAESAR competition. The main AEAD instance of the Ascon suite is Ascon-128 and offers 128-bit security according to [15]. It is based on the so-called Monkey Duplex mode [7] with a stronger keyed initialization and keyed finalization function, respectively, which means the underlying permutation is carried out with an increased number of rounds. Said permutation operates on a 320-bit state (organized in five 64-bit words) by iteratively applying a round function p. The number of rounds is $a = 12$ in the initialization and finalization phase, and $b = 6$ otherwise; the corresponding permutations are referred to as p^a and p^b in the specification. Ascon-128 processes associated data as well as plaintext/ciphertext with a rate of $r = 64$ bits, i.e. the capacity is 256 bits. The hash function of the Ascon suite is a classical sponge construction.

Ascon's round function p is SPN-based and comprises three parts: (i) the addition of an 8-bit round constant c_r to a 64-bit state-word, (ii) a substitution layer that operates across the five words of the state and implements an affine equivalent of the S-box in the χ mapping of Keccak [6], and (iii) a diffusion layer consisting of linear functions that are similar to the Σ functions in SHA2 and performed on each state-word individually. The S-box maps five input bits to five output bits and is applied to each column of the state, whereby the five state-words are arranged upon each other. It is normally implemented in a bit-sliced fashion using logical ANDs and XORs. The diffusion layer performs an operation of the form $x = x \oplus (x \ggg n_1) \oplus (x \ggg n_2)$ on each word of the state with $n_1 \in \{1, 7, 10, 19, 61\}$ and $n_2 \in \{6, 17, 28, 39, 41\}$ [15].

SPARKLE. The Sparkle suite submitted to NIST consists of four instances of the AEAD algorithm Schwaemm, targeting security levels of 128, 192, and 256 bits, as well as two instances of the hash function Esch with digest lengths of 256 and 384 bits. All instances are built on top of the Sparkle permutation family, which consists of three members that differ by the width (i.e. the state size) and the number of steps they execute. Schwaemm is based on the highly-efficient Beetle mode of use [11], whereas Esch can be classified as a sponge construction. The main instance of Schwaemm uses the 384-bit variant of the Sparkle permutation, i.e. Sparkle384, with a rate of 256 bits. This variant is also used for Esch256, the main instance of the hash function Esch. Besides Sparkle384, there exists also a smaller and a larger version of the permutation with a width of 256 and 512 bits, respectively (see [2] for details).

Sparkle384 is a classical ARX design, optimized for high speed on a wide range of 8, 16, and 32-bit microcontrollers. The permutation is performed with a big number of steps, namely 11, for initialization, finalization, and separation between the processing of associated data and the secret message, while a slim (i.e. 7-step) version is used to update the intermediate state. From a high-level point of view, the permutation has an SPN structure and comprises three main parts: (i) a non-linear layer consisting of six parallel ARX-boxes, (ii) a simple linear diffusion layer, (iii) the addition of a step counter and round constant to the 384-bit state. The ARX-box is called Alzette and can be seen as a small 64-bit block cipher that operates on two 32-bit words and performs additions modulo 2^{32}, logical XORs, and rotations by 16, 17, 24, and 31 bits [2]. On the other hand, the linear layer is, in essence, a Feistel round with a linear Feistel function, followed by a swap of the left and right half of the state.

TinyJAMBU. TinyJambu is, in essence, a permutation-based variant of the AEAD algorithm Jambu, which was a candidate of the CAESAR competition but did not make it into the final portfolio. A distinguishing feature of TinyJambu is that it uses a keyed permutation and not a public (i.e. unkeyed) one like the other AEAD algorithms. However, according to [22], TinyJambu can also be viewed as a block-cipher-based design. In any case, the permutation has a very short width of only 128 bits. There is no key schedule, which means the key-bytes are directly added to the state. The specification [31] describes three variants of TinyJambu with key lengths of 128, 192, and 256 bits, whereby the main instance uses a 128-bit key with a 96-bit nonce. Its mode of operation is based on the duplex construction [5], but offers better security in nonce-misuse settings [31, Sect. 6]. Both the associated data and the plaintext/ciphertext are processed at a relatively low rate of 32 bits, i.e. four bytes.

The 128-bit permutation of TinyJambu is essentially a Nonlinear Feedback Shift Register (NFSR) whose feedback path consists of four bit-wise XOR and a bit-wise NAND operation. The latter is the only non-linear component of the whole permutation. Several rounds can be computed in parallel (e.g. 32 rounds when the target platform is a 32-bit microcontroller), which benefits software performance. The most costly part of the permutation are special shifts of the form $c = (a \gg n) \vee (b \ll (32 - n))$, where a, b, and c are 32-bit words and the shift distance $n \in \{6, 15, 21, 27\}$. These so-called *funnel shifts* concatenate two 32-bit words into a 64-bit value, shift this 64-bit value n bits left or right, and return the 32 most-significant (left shift) or least-significant (right shift) bits as result. Optimized software implementations combine 128 rounds (i.e. 128 state updates) into a step and execute several steps in a loop. TinyJambu processes associated data by iterating the step-loop five times (i.e. 640 rounds), whereas plaintext/ciphertex is processed with eight iterations (i.e. 1024 rounds).

Xoodoo. Xoodyak is a highly versatile cryptographic scheme that is suitable for a wide range of symmetric-key functions including hashing, pseudo-random bit generation, authentication, encryption, and authenticated encryption. At its

heart is Xoodoo, a lightweight 384-bit permutation [13]. The Xoodyak suite submitted to the NIST lightweight crypto project includes an AEAD algorithm and a hash function; both are built on the Cyclist mode of operation [12]. To perform authenticated encryption, Cyclist has to be initialized in keyed mode with a 128-bit key and nonce, respectively, after which associated data can be absorbed at a rate of 352 bits (i.e. 44 bytes), whereas plaintext/ciphertext gets processed at a rate of 192 bits. On the other hand, when Cyclist is operated in hash mode, the rate is 128 bits (i.e. 256 bits of capacity).

Xoodoo was inspired by Keccak [6] and Gimli [4] in the sense that the state has the same size and is represented in the same way as in Gimli, though the round function is similar to Keccak. Consequently, the state has the form of a 3×4 matrix of 32-bit words, which can be visualized via three horizontal 128-bit planes (one above the other), each consisting of four 32-bit lanes. It is also possible to view the 384-bit state as 128 columns of three bits lying upon another (i.e. each bit belongs to a different plane). The Xoodoo permutation executes 12 iterations of a round function of five steps: a column-parity mixing layer θ, a non-linear layer χ, two plane-shifting layers (ρ_{west} and ρ_{east}) between them, and a round-constant addition. Both ρ layers move bits horizontally and perform lane-wise rotations of planes as well as rotations of lanes by 11, 1, and 8 bits to the left. On the other hand, in the parity-computation part of θ and in the χ layer, state-bits interact only vertically, i.e. within 3-bit columns. The θ layer mainly executes XORs and left-rotations by 5 and 14 bits. Finally, the non-linear layer χ applies a 3-bit S-box to each column of the state, which can be computed using logical ANDs, XORs, and bitwise complements.

4 Implementation Details

We developed optimized implementations of the four AEAD algorithms for the purpose of benchmarking and performance analysis using a combination of C and MSP430 Assembly language. More concretely, the underlying permutation is the Assembly component, while the surrounding mode of operation (or mode of use) is written in C. Most of the C source code is based on either reference or optimized implementations provided by the designer teams, but we adapted them to adhere to the low-level benchmarking API introduced in [10] to ensure a consistent evaluation. The MSP430 Assembly code of the four permutations (which we developed from scratch) is based on a common set of special macros for load/store operations (using different addressing modes), arithmetic/logical operations, and shifts/rotations of both 32-bit and 64-bit operands. Our main optimization goal for the permutations was to achieve a good trade-off between execution time and (binary) code size, and therefore we refrained from certain optimization techniques like full loop unrolling, which in the case of MSP430 often only achieve a modest reduction in execution time at the expense of an enormous increase in code size. We devoted a similar amount of optimization time and effort to each of the four permutations to guarantee a fair evaluation and comparison of the performance of the AEAD algorithms.

The rotations performed by the four permutations are composed of macros for 1-bit and 8-bit rotation. As mentioned in Sect. 2, a rotation by a distance of $n > 16$ bits can be split up into a rotation by $k = n \bmod 16$ bits (taking into account that a k-bit rotation in one direction equals a $(16 - k)$-bit rotation in the other direction), followed by a rotation by a multiple of 16 bits, which can usually be performed implicitly (i.e. as part of a subsequent arithmetic/logical or store operation) and is, therefore, free. Since all four permutations use the same set of macros for rotations and other operations on 32/64-bit words, the optimization effort essentially boiled down to finding a good register allocation strategy in order to minimize the number of memory accesses. This includes both explicit accesses in the form of loads and stores, but also implicit accesses that take place when executing instructions where one or both operands reside in memory. A good register allocation is crucial for Ascon, Sparkle384, and Xoodyak since the size of their state is too big for the register space of the MSP430, which means the state has to be kept in RAM and parts of the state are loaded to registers to reduce the latency of arithmetic/logical instructions executed on them. However, TinyJambu's 128-bit state can be entirely kept in the register file throughout the computation of the permutation, in which case still four registers remain available for e.g. storing intermediate results.

As mentioned before, the C implementations of the mode of operation/use of the algorithms are largely based on source codes from the designers, but we modified them to comply with the low-level API given in [10]. The high-level API for authenticated encryption and decryption specified in [20, Sect. 3.5] can be implemented as simple wrappers around the low-level functions. This high-level API represents the plaintext, ciphertext, associated data, key, and nonce as arrays of bytes, i.e. arrays of type `unsigned char`, while the permutations operate on 32-bit or 64-bit words. It is, therefore, tempting to cast a pointer to a byte-array to a pointer to an array of unsigned 32/64-bit integers, e.g. when injecting a block of plaintext (or associated data) into the state. However, the ISO C standard only permits such upcasting of an unsigned-char pointer to an unsigned-integer pointer if the former meets the alignment requirements of the latter (which are more strict), otherwise the result of the cast is undefined. In the case of the MSP430 architecture, a 32-bit or 64-bit integer in memory has to be 2-byte aligned, i.e. its address must be even [27]. As a consequence, the casting of a pointer to a byte-array to a pointer to an unsigned-integer-array is only allowed when the start address of the byte-array is even. If this condition is not satisfied, the plaintext (resp. associated data) blocks have to be copied to an aligned buffer. Alternatively, it is, of course, always possible to process the blocks of plaintext and associated data in a byte-wise way. In the following, we briefly outline how we implemented and optimized the four AEAD algorithms and their permutations for the MSP430 architecture.

ASCON. Ascon is well suited for platforms with small register space because each of the two layers of the permutation needs, at any time, only a part of the state (but never the complete state) in registers. Our MSP430 implementation

processes the substitution layer in 16-bit slices, i.e. a 16-bit part of each state-word is loaded, processed, and stored, and these steps are repeated four times in a simple loop. The linear diffusion layer is implemented in a straightforward fashion, i.e. one state-word at a time. In summary, each of the five state-words loaded from (and stored to) RAM twice per round, which means Ascon has relatively high locality. As stated in the last section, the diffusion layer consists of operations of the form $x = x \oplus (x \ggg n_1) \oplus (x \ggg n_2)$; we tried alternative implementation options, e.g. $x = x \oplus ((x \oplus (x \ggg (n_2 - n_1))) \ggg n_1)$, with the goal of minimizing the execution time of the rotations.

Ascon's mode of operation is fairly straightforward to implement on basis of the low-level API from [10]. A peculiarity of Ascon is the byte-order of the five state-words, which is big endian, while MSP430 and most other embedded microcontrollers process and store 32-bit and 64-bit integers using little endian representation. Therefore, the byte-order of 64-bit words that are injected into (or extracted from) the state has to be reversed. Our implementation performs the injection/extraction of words (including endianness conversion) in a byte-by-byte fashion, which has the advantage that we do not need to pay attention to the alignment of the byte-arrays in which the inputs/outputs are stored.

SPARKLE. Sparkle384, which is the permutation of the primary instance of the Schwaemm family, has relatively high locality (though not as high as Ascon) and can, therefore, be well optimized for MSP430. Our implementation of the permutation processes the non-linear layer in a loop and evaluates one ARX-box at a time. An ARX-box computation requires ten registers: four to store two 32-bit state words, two for a 32-bit round constant, further two for an intermediate result, and one each for a pointer to the round-constant and state array, respectively. We integrated the computation of the two temporary values t_x and t_y into the ARX-box layer to reduce the number of memory accesses in the subsequent linear layer. In this way, each 32-bit word of the state is loaded and stored twice per round (similar to Ascon); once in the ARX-box layer and then a second time in the linear layer. However, some further memory accesses are necessary for the round constants and the temporary value t_y, which has to be stored on the stack due to the lack of free registers.

Schwaemm's mode of operation uses apart from the permutation also two auxiliary functions: a feedback function ρ and a rate-whitening function $\mathcal{W}$. We merge both functions into a single loop to reduce their execution time. Our C implementation of the mode also optimizes the processing of plaintext, ciphertext, and associated data, which are stored in byte-arrays. We check at runtime whether the pointers to these arrays are sufficiently aligned for an upcasting to `uint32_t` pointers; when this is the case we directly process the byte-arrays as integer-arrays, otherwise we copy them first to an aligned buffer via `memcpy`.

TinyJAMBU. TinyJambu has the highest locality among all four permutations since the full state can be kept in registers during the computation of the permutation. Nonetheless, some memory accesses are still required to load the

key-words in each round. Due to the permutation's high locality, the execution time is dominated by the funnel shifts, which extract a 32-bit word at a certain position within two concatenated 32-bit words (i.e. a 64-bit word). The source code provided by the designers implements these funnel shifts as normal right-shift operations of two concatenated state-words by distances of 6, 15, 21, and 27 bits. However, in MSP430 Assembly language, the four funnel shifts can be performed more efficiently by a 1-bit right-shift-through-carry of a 32-bit word (three instructions), a 1-bit left-shift-through-carry of a 32-bit word (also three instructions), an ordinary 5-bit left-shift of a 48-bit word (15 instructions), and an ordinary 5-bit right-shift of a 64-bit word (20 instructions).

TinyJambu processes plaintext/ciphertext and associated data with a rate of four bytes. The low-level encryption/decryption functions check whether the pointers to the byte-arrays containing these inputs are properly aligned for an upcasting to `uint32_t` pointers; when this is not the case the four bytes to be processed are copied into an aligned buffer, similar to Schwaemm. But unlike Schwaemm, the input blocks are copied byte by byte using plain C statements since calling `memcpy` would introduce a significant overhead for four bytes.

Xoodyak. Similar to Ascon and Sparkle, the state of the Xoodoo permutation is too big for the register file of a MSP430 microcontroller and, thus, has to be stored in RAM. A straightforward implementation of the five steps of the permutation, one step after another, would require a large number of load and store operations. In order to reduce the number of memory accesses, we tried to integrate (parts of) the plane-shifting layers ρ_{west} and ρ_{east} into the mixing layer θ and non-linear layer χ, respectively. Unfortunately, a full integration is not possible due to the limited register space (at least not when the goal is to achieve a good trade-off between performance and code size), which means the lane-wise rotations within a plane that form part of ρ_{west} and ρ_{east} still have to be implemented as separate steps with their own load and store operations. As a consequence, four state-words are loaded and stored twice per round, and the remaining eight words three times per round. This large number of load/store operations makes Xoodoo the permutation with the lowest locality.

Our low-level functions for Xoodyak's Cyclist mode of operation deal with unaligned byte-arrays for associated data and plaintext/ciphertext in the same way as the Schwaemm implementation: we first check at runtime whether the pointers to these arrays can be casted to `uint32_t` pointers and use `memcpy` to copy the bytes block-wise into an aligned buffer if this is not the case.

5 Performance Evaluation and Comparison

We compiled and assembled the source code of the four AEAD algorithms with version 7.2 of IAR Embedded Workbench for MSP430[4] and used its integrated

[4] `http://www.iar.com/products/architectures/iar-embedded-workbench-for-msp430` (accessed on 2022-12-14)

Table 2. Main characteristics and implementation results of the four permutations.

Characteristic/result	Ascon	Sparkle	TinyJambu	Xoodoo
Performance characteristics				
Execution time (cycles)	3510	5946	2454	8985
Number of executed instr.	2369	3811	2134	5191
Average cycles/instruction	1.48	1.56	1.15	1.73
Memory characteristics				
RAM consumption (bytes)	56	76	54	66
– of which is stack (bytes)	16	28	22	18
Code size (bytes)	708	618	652	570
Instruction-type characteristics				
Branching instructions	30	63	8	96
Memory-to-Memory (M2M)	0	21	8	12
Memory-to-Register (M2R)	261	491	146	884
Register-to-Memory (R2M)	254	493	19	789
Register-to-Register (R2R)	1824	2729	1953	3410
Percentage of R2R instr.	77.0%	71.6%	91.5%	65.7%

cycle-accurate instruction set simulator to determine the execution time of the permutations alone and the high-level encryption functions. Our target device was a MSP430F1611 microcontroller, which comes with 10 kB SRAM and has a flash capacity of 48 kB. In order to be able to examine our implementations of the permutations in more detail, we also developed a tool that emulates the execution of MSP430 instructions step by step and collects information via the execution trace. The tool works with snapshots of registers and memory (since they can be exported from IAR Workbench) and is able to emulate all 27 core instructions of the MSP430 with the supported addressing modes [28]. While the instructions are executed, information about the instruction type, the used addressing mode(s), the number of memory accesses, and so on is recorded.

Table 2 shows various results we obtained for performance, RAM and flash consumption, and the type of instructions executed by each permutation. The execution time (in cycles) covers all instructions contained in the Assembly file of the permutation, but does not include the generation or passing of function arguments like a pointer to the state or the number of rounds. We can observe that TinyJambu has the fastest permutation with just 2454 clock cycles, while Xoodoo is by far the worst in terms of execution time. TinyJambu's small Cycles-per-Instruction (CPI) ratio of 1.15 means that most of its instructions execute in one cycle, which is only possible when the operands and result are read from and written to registers instead of a location in memory. Indeed, as shown in Table 2, the percentage of Register-to-Register (R2R) instructions in TinyJambu's permutation is very high, namely above 91%. Both the CPI and ratio of R2R instructions confirms that TinyJambu has high locality. At the opposite end of the spectrum is Xoodoo, which has the lowest locality of the four evaluated permutations (evidenced by a CPI of 1.73 and only 65.7% R2R

Table 3. Detailed execution-time and throughput analysis of the permutations.

Characteristic/result	Ascon	Sparkle	TinyJambu	Xoodoo
State size (bytes)	40	48	16	48
Encryption rate (bytes)	8	32	4	24
Authentication rate (bytes)	8	32	4	44
Number of rounds or steps	6	7	8 (5)	12
Execution-time analysis of single round/step				
Cycles per round/step	577 (100%)	844 (100%)	302 (100%)	746 (100%)
– of which are rotations	160 (27.7%)	150 (17.8%)	172 (57.0%)	153 (20.5%)
– of which are non-lin. ops.	20 (3.5%)	48 (5.7%)	8 (2.6%)	24 (3.2%)
Execution-time analysis of full permutation				
Cycles for full permutation	3510 (100%)	5946 (100%)	2454 (100%)	8985 (100%)
– of which are rotations	960 (27.4%)	1050 (17.7%)	1376 (56.1%)	1836 (20.4%)
– of which are non-lin. ops.	120 (3.4%)	336 (5.7%)	64 (2.6%)	288 (3.2%)
Throughput analysis of full permutation				
Cycles per state-byte	87.75	123.88	153.38	187.19
Cycles per rate-byte (enc.)	438.75	185.82	613.50	374.38
Cycles per rate-byte (auth.)	438.75	185.82	387.00	204.20

instructions). Ascon has the second-best locality, and Sparkle is locality-wise approximately in the middle between Ascon and Xoodoo.

The RAM footprint (including stack usage) of the four permutations is relatively small and ranges from 54 bytes (TinyJambu) to 76 bytes (Sparkle). In essence, RAM is occupied for the state and, in the case of TinyJambu, for the key, while the stack is mainly used for the preservation of callee-saved registers and to store infrequently-used local variables like loop counters. Also the code size of the permutations is relatively similar since the smallest one (Xoodoo) and biggest one (Ascon) differ by only 138 bytes, which is roughly 24% of the code size of the former.

Table 3 provides more-detailed information about the execution time of the permutations, including an analysis of the cycles spent for shifts/rotations and non-linear operations (i.e. addition in the case of Sparkle, logical AND for the other three permutations). The table also summarizes the main characteristics of the permutations, e.g. the size of the state, the rate used for authentication and for encryption, and the number of rounds or steps. We analyzed a single round or step of each permutation and determined the overall cycle count, the number of cycles spent for shifts/rotations, and the number of cycles for non-linear operations. The latter was evaluated with help of the specification of the permutation and does not include any `add.w` or `and.w` instruction that has no impact on non-linearity, e.g. the `adc.w` at line 4 of the `QROL` rotation macro in Listing 1. According to the per-round/step results in Table 3, the rotations are more costly than the non-linear operations, and this holds true for each of the four permutations. However, the relative computational cost of rotations versus non-linear operations is not only determined by the design of the permutation

but also by the features of the target architecture. For example, SPARKLE and XOODOO were designed such that, when implemented for a 32-bit ARM microcontroller, each rotation can be "folded" into an arithmetic/logical instruction and both together executed within a single cycle, which makes these rotations basically free. Therefore, when 32-bit ARM is the target architecture, the nonlinear operations contribute more cycles to the overall execution time than the rotations, while the opposite is the case for MSP430. To be more concrete, the rotations make up between 17.7% and 56.1% of the overall cycle counts of the permutations on an MSP430F1611 microcontroller. These results underline the importance of choosing the rotation (resp. shift) distances carefully, taking into account both security and efficiency aspects.

As explained in Sect. 2, a shift/rotation of a 32 or 64-bit word by a distance of d bits is fast on MSP430 if either (i) d is a multiple of 16, (ii) d is close to a multiple of 16 (e.g. 1, 2, 14, 15, 17, 18, ...), or (iii) d is a multiple of 8. The SPARKLE permutation performs seven rotations of 32-bit words in each of its ARX-boxes; the distances are 31, 24, 17, 17, 31, 24, and 16 bits. Each distance meets the above requirements, which makes the rotations relatively fast (one is completely free, one takes six cycles, and the other five rotations require three cycles). Overall, the rotations contribute roughly 17.8% to the execution time of SPARKLE. The distances of the rotations carried out by XOODYAK include three that are relatively fast (namely by 1 and 8 bits in ρ_{east} and by 14 bits in θ), but also two slow ones (by 5 bits in θ and 11 bits in ρ_{west}). In summary, the rotations account for 20.4% of the execution time of XOODYAK. The diffusion layer of ASCON includes ten rotations (executed on 64-bit words) by distances of 19, 28, 61, 39, 1, 6, 10, 17, 7, and 41 bits. Only two out of this total of ten distances, namely 1 and 17 bits, can be considered fast according to the above requirements. Though some optimizations are possible (see Sect. 5), our overall verdict is that the rotation distances of ASCON are not particularly "MSP430-friendly," which explains why the rotations consume 27.4% of the permutation cycles. Finally, TINYJAMBU is a special case because it performs funnel shifts instead of actual rotations. As explained in Sect. 5, these funnel shifts can be implemented by two 1-bit shift-through-carry operations on a 32-bit word and two 5-bit shifts (carried out on a 48-bit and a 64-bit word, respectively). The former two are fast but the latter two extremely slow. In summary, the funnel shifts make up 56.1% of TINYJAMBU's overall permutation cycles.

The impact of the rotations (resp. funnel shifts) on the total execution time of the four permutations should not be viewed as completely independent from other efficiency aspects like locality. TINYJAMBU has very high locality and, as a consequence, wastes only few cycles for memory accesses (this is one of the reasons for its relatively fast execution time). Therefore, it is natural that the funnel shifts constitute a large fraction of the execution time, which makes the designers' choice of shift distances appear worse (in relation to the other three permutations) than they are in reality. The opposite is the case for XOODYAK's permutation. Namely, the long execution time of XOODOO (which is partly due to poor locality) makes the rotation distances look less costly than they are.

Table 4. Execution time (in cycles) of the AEAD algorithms for authentication only (dlen = 0), encryption only (adlen = 0), and authenticated encryption (adlen = dlen).

adlen	dlen	Ascon	Schwaemm	TinyJambu	Xoodyak
16	0	25567	20311	18539	28225
128	0	75729	38777	63952	47091
1024	0	477025	214421	427280	243385
0	16	22109	20704	22191	28273
0	128	72957	39618	93168	74865
0	1024	479707	221080	661008	420299
16	16	32834	30748	28680	28377
128	128	133842	68126	145073	93838
1024	1024	941924	425268	1076241	635566

Since the state size of three of the four permutations differs, it makes sense to analyze the throughput in terms of execution time divided by the state-size in bytes. The results at the bottom of Table 3 show that Ascon wins in this category with a throughput of approximately 87.75 cycles per state-byte. Also contained at the bottom of this table are the throughput figures per rate-byte for encryption and authentication, respectively. The cycles per rate-byte serve as a good benchmark for the efficiency of both the permutation and the mode of operation/use of the corresponding AEAD algorithm. Schwaemm employs the Beetle mode of operation, which allows it to process associated data and plaintext/ciphertext at a rate of 32 bytes. The resulting throughput of 185.82 cycles per rate-byte is the best among the four evaluated AEAD schemes. Also Xoodyak profits from a fairly high rate, namely 24 bytes for encryption, and achieves a throughput of 374.38 cycles per rate-byte. Even though Ascon and TinyJambu have fast permutations, their throughput is relatively poor due to a small rate. Note that the throughput of both TinyJambu and Xoodyak is much higher for authentication than for encryption; in the former case because of a smaller number of steps and in the latter case due to a higher rate.

Table 4 shows the execution time of the four AEAD algorithms for authentication only (i.e. no plaintext is processed), encryption only (i.e. no associated data is processed) and authenticated encryption (both the associated data and plaintext have the same length). For each scenario, we evaluated the execution time for inputs of three different lengths: short (i.e. 16 bytes), medium (i.e. 128 bytes), and long (i.e. 1024 bytes). The timings in Table 4 are closely correlated with the throughput values at the bottom of Table 3, in particular for medium and long inputs. Therefore, it is not surprising that, overall, Schwaemm is the best performer across different lengths of associated data and plaintext. When the inputs are short (i.e. 16 bytes), the execution times of the four algorithms are relatively similar and depend not only on the throughput, but also on the efficiency of operations like initialization, finalization, and computation of the authentication tag. However, for medium-size inputs, Schwaemm outperforms Xoodyak, which is (overall) the second-best algorithm, by a factor of 1.89 in

the encryption-only case and a factor of approximately 1.45 for authenticated encryption. Finally, in the authentication-only scenario, the speed-up factor is smaller, namely about 1.21 for associated data of medium length and a bit less for longer lengths. Ascon and TinyJambu are around two times slower than Schwaemm for both medium and long inputs.

6 Conclusions

In this paper, presented a performance analysis of the four AEAD algorithms Ascon, TinyJambu, Schwaemm, and Xoodyak on a 16-bit MSP430 microcontroller. We developed carefully-optimized Assembler implementations of the underlying permutations, whereby we aimed for a reasonable trade-off between execution time and (binary) code size. Our results show that the shift/rotation distances and the locality have a significant impact on the performance of the permutations. TinyJambu's permutation has very high locality since its entire state can be kept in registers. The permutation of Ascon and Sparkle have the second and third-bast locality; each word of their state needs to be loaded from RAM and written back to RAM twice per round or step. Xoodoo shows the worst locality of the four permutations. On the other hand, when it comes to rotation distances, Sparkle is the winner since the majority of its rotations can be executed in only three clock cycles. Xoodoo and TinyJambu perform a mix of fast and slow rotations (resp. shifts), while almost all of the rotation distances of Ascon's permutation are not well-suited for MSP430. The actual performance of each of the four AEAD algorithms does not only depend on the permutation, but also the rate for encryption and authentication. Our results show that Schwaemm is clearly the best overall performer across different use cases (authentication only, encryption only, and authenticated encryption) and input lengths. When encrypting a 128-byte plaintext, Schwaemm is 1.89 times faster than Xoodyak and outperforms Ascon by a factor of 1.84. Xoodyak is more competitive when a large amount of associated data is processed, whereas TinyJambu is particularly efficient for the authentication of very short blocks of associated data (up to approximately 16 bytes).

Acknowledgements. The last author was supported by the Fonds National de la Recherche (FNR) Luxembourg under CORE grant C19/IS/13641232. The source code is available online at `http://github.com/johgrolux/aead430`.

References

1. Arm Limited. ARM Cortex-M3 Processor Technical Reference Manual, Revision r2p1. Available for download at `http://developer.arm.com/documentation/100165/latest`, 2016.
2. C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Lightweight AEAD and hashing using the Sparkle permutation family. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, June 2020.

3. D. J. Bernstein. The Salsa20 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 84–97. Springer Verlag, 2008.
4. D. J. Bernstein, S. Kölbl, S. Lucks, P. M. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier. Gimli: A cross-platform permutation. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems — CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer Verlag, 2017.
5. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. Available for download at `http://keccak.team/files/CSF-0.1.pdf`, 2011.
6. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, version 3.0. Available for download at `http://keccak.team/files/Keccak-reference-3.0.pdf`, 2011.
7. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Permutation-based encryption, authentication and authenticated encryption. In *Record of the 1st ECRYPT II Workshop on New Directions in Authenticated Encryption (DIAC 2012)*, pages 159–170, 2012.
8. 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.
9. 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.
10. L. Cardoso dos Santos, J. Großschädl, and A. Biryukov. FELICS-AEAD: Benchmarking of lightweight authenticated encryption algorithms. In S. Belaïd and T. Güneysu, editors, *Smart Card Research and Advanced Applications — CARDIS 2019*, volume 11833 of *Lecture Notes in Computer Science*, pages 216–233. Springer Verlag, 2019.
11. A. Chakraborti, N. Datta, M. Nandi, and K. Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):218–241, May 2018.
12. J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology*, 2020(S1):60–87, June 2020.
13. J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, Dec. 2018.
14. D. Dang, M. Plant, and M. Poole. Wireless connectivity for the Internet of Things (IoT) with MSP430 microcontrollers (MCUs). Texas Instruments white paper, available for download at `http://www.ti.com/lit/wp/slay028/slay028.pdf`, Mar. 2014.
15. 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.
16. V. D. Gligor. Light-weight cryptography – How light is light? Keynote presentation at the Information Security Summer School, Florida State University. Slide deck available online at `http://www.sait.fsu.edu/conferences/2005/is3/resources/slides/gligorv-cryptolite.ppt`, May 2005.

17. 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.
18. M. Hamburg. The STROBE protocol framework. Cryptology ePrint Archive, Report 2017/003, available for download at `http://eprint.iacr.org/2017/003`, 2017.
19. Microchip Technology Inc. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash: ATmega128, ATmega128L. Available for download at `http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf`, 2011.
20. 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`, Aug. 2018.
21. National Institute of Standards and Technology (NIST). Benchmarking of lightweight cryptographic algorithms on microcontrollers. Available online at `http://github.com/usnistgov/Lightweight-Cryptography-Benchmarking`, 2020.
22. National Institute of Standards and Technology (NIST). Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process. Internal Report 8369, available for download at `http://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf`, July 2021.
23. T. Perrin. Stateful hash objects: API and constructions. Specification, available online at `http://github.com/noiseprotocol/sho_spec`, 2018.
24. S. Renner, E. Pozzobon, and J. Mottok. NIST LWC software performance benchmarks on microcontrollers. Available online at `http://lwc.las3.de`, 2020.
25. V. Rzehak. Low-power FRAM microcontrollers and their applications. Texas Instruments white paper, available for download at `http://www.ti.com/lit/wp/slaa502/slaa502.pdf`, July 2019.
26. M.-J. O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In J. Benaloh, editor, *Topics in Cryptology — CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 270–285. Springer Verlag, 2014.
27. 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.
28. 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.
29. Texas Instruments Inc. MSP430 Ultra-Low-Power Microcontrollers. Product bulletin, available for download at `http://www.ti.com/lit/sg/slab034w/slab034w.pdf`, 2013.
30. R. Weatherley. Lightweight cryptography primitives documentation. Available online at `http://rweather.github.io/lwc-finalists/index.html`, 2021.
31. H. Wu and T. Huang. TinyJAMBU: A family of lightweight authenticated encryption algorithms (Version 2). Specification, available online at `http://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf`, 2021.
32. L. Yan, Y. Zhang, L. T. Yang, and H. Ning. *The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems*. Auerbach Publications, 2008.