An Evaluation of the Multi-Platform Efficiency of Lightweight Cryptographic Permutations

Luan Cardoso dos Santos and Johann Großschädl

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

Abstract. Permutation-based symmetric cryptography has become increasingly popular over the past ten years, especially in the lightweight domain. More than half of the 32 second-round candidates of NIST's lightweight cryptography standardization project are permutation-based designs or can be instantiated with a permutation. The performance of a permutation-based construction depends, among other aspects, on the rate (i.e. the number of bytes processed per call of the permutation function) and the execution time of the permutation. In this paper we analyze the execution time and code size of assembler implementations of the permutation of Ascon, Gimli, Schwaemm, and Xoodyak on an 8-bit AVR and a 32-bit ARM Cortex-M3 microcontroller. Our aim is to ascertain how well these four permutations perform on microcontrollers with very different architectural and micro-architectural characteristics such as the available register capacity or the latency of multi-bit shifts and rotations. We also determine the impact of flash wait states on the execution time of the permutations on Cortex-M3 development boards with 0, 2, and 4 wait states. Our results show that the throughput (in terms of permutation time divided by rate when the capacity is fixed to 256 bits) of the permutation of Ascon, Schwaemm, and Xoodyak is similar on ARM Cortex-M3 and lies in the range of 41.1 to 48.6 cycles per rate-byte. However, on an 8-bit AVR ATmega128, the permutation of Schwaemm outperforms its counterparts of Ascon and Xoodyak by a factor of 1.20 and 1.59, respectively.

1 Introduction

The term *Internet of Things (IoT)* describes the evolution of the Internet from a computer network to a network that connects various kinds of smart devices and enables them to communicate with each other or transmit data to central servers. This development started roughly 15 years ago, when more and more "everyday objects," ranging from household appliances over business machines to vehicles, became equipped with microcontrollers and transceivers for wireless communication (e.g. ZigBee, Bluetooth, WiFi). These devices differ greatly in terms of computing power, but also regarding their data transmission speeds and run-time memory capacities. At one end of the spectrum are e.g. modern cars, which are equipped with powerful processors, while e.g. battery-operated

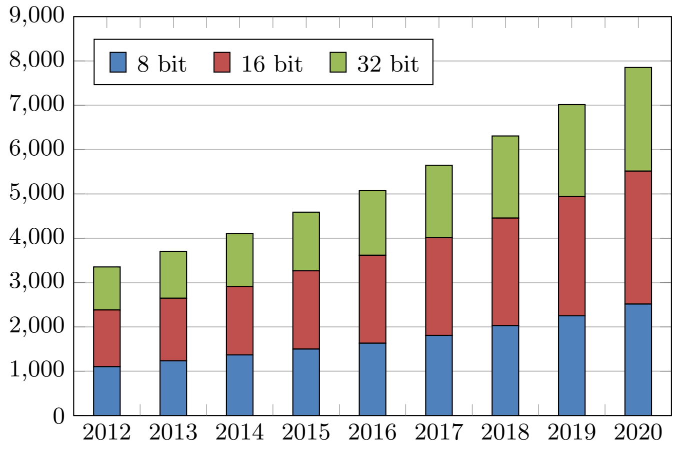

Fig. 1. North American microcontroller market by product (8-bit, 16-bit, 32-bit) in million units (source: Radiant Insights Inc. [27])

miniature sensor nodes at the opposite end of the spectrum commonly feature only a small 8-bit or 16-bit microcontroller. Already today, approximately twice as many "smart things" are connected to the Internet than ordinary computers like PCs or laptops, and this proportion will grow rapidly over the next couple of years [28]. Internet-enabled smart devices can be found in basically all areas of our life, from home automation over industrial production ("Industry 4.0") to transportation and logistics.

The IoT can be seen as a large ecosystem populated by highly diverse and heterogeneous devices, which come in all shapes and sizes. Therefore, it is little surprising that there exist dozens of different (and largely incompatible) microcontroller platforms, operating systems, and wireless communication standards for the IoT, many of which are optimized to serve a certain application domain with specific requirements and constraints. This heterogeneity of IoT devices is in stark contrast to the "monoculture" in the realm of classical computers like PCs or laptops, where the 64-bit Intel architecture has a market share of well over 90%. Nonetheless, 64-bit Intel processors represent only a small fraction of the IoT altogether, which is (quantitatively) dominated by microcontrollers with rather modest computational capabilities. Figure 1 shows a forecast of the development of the North American microcontroller market until 2020, split up in 8-bit, 16-bit, and 32-bit architectures [27]. The North American market was estimated to be over 3700 million units in 2013 and is expected to reach some 8000 million units in 2020, i.e. the compound annual growth rate is more than 11.2% in the period from 2014 to 2020. 32-bit microcontrollers constitute the fastest growing product segment over the forecast period, driven mainly by an increased demand for higher processing capabilities and the expected reduction in unit prices. Currently, the ARM architecture is the undisputed leader in the 32-bit segment, but it faces fierce competition by ESP32 and RISC-V. There is also a growing demand for 16-bit microcontrollers (e.g. MSP430, 68HC16) due to the need for high level of precision in embedded processing and the development of intelligent and real-time functions in the automotive domain [27]. The 8-bit platforms (e.g. AVR, PIC) are expected to retain their market share and continue to be widely used for automotive and industrial applications [24].

Since there is no single dominating microcontroller platform in the IoT, it is essential that a cryptographic algorithm delivers consistently high performance on a wide variety of 8, 16, and 32-bit architectures. This is far from trivial to achieve since, for example, a 32-bit ARM Cortex-M3 microcontroller has significantly different architectural and micro-architectural characteristics than an 8-bit AVR ATmega microcontroller. The Cortex-M3 has 16 registers, of which 14 are available for general use, i.e. the general-purpose register space amounts to 448 bits. AVR microcontrollers, on the other hand, have 32 general-purpose registers, but each of them can only store eight bits of data, yielding a usable register space of 256 bits. ARM and AVR also differ greatly in their ability to execute multi-bit shifts or rotations, which are performance-critical operations of various symmetric cryptosystems. The arithmetic/logic unit of a Cortex-M3 comes with a fast barrel shifter capable to shift or rotate a 32-bit word by an arbitrary number of bits in a single cycle. Furthermore, a shift or rotation can be combined with most data-processing instructions, in which case they become practically "free" [2]. More specifically, the second operand of most arithmetic or logical instructions can be shifted or rotated (before the actual operation is executed) without increasing the instruction latency. However, the situation is much different for 8 and 16-bit architectures, as most of them have only single-bit shift and rotate instructions, which means that shifting a register by n bits requires (at least) n clock cycles. This can make multi-bit shifts and rotations very costly, especially when the length of the operand to be shifted or rotated exceeds the capacity of a single register. For example, rotating a 32-bit word on an 8-bit AVR microcontroller (stored in four registers) can, depending on the rotation amount, require more than 20 clock cycles.

A *cryptographic permutation* is a bijective mapping within $\mathbb{Z}_2^b$, designed to behave as a random permutation, i.e. a permutation chosen randomly from the set of all possible permutations that operate on b bits. The width b of a cryptographic permutation is usually between 200 (for cryptosystems targeting the lightweight domain) and 1600 [12]. Permutation-based cryptography emerged approximately 15 years ago as a sub-area of research in the field of symmetric cryptography and started to attract particular interest when the hash function Keccak [11] and the stream cipher Salsa [6] became popular[1]. Permutations are extremely flexible and versatile primitives, similar to block ciphers, and can be used to construct e.g. hash functions, message authentication codes, pseudo-

[1] In October 2012, the U.S. National Institute of Standards and Technology (NIST) selected Keccak as winner of the SHA-3 hash competition [25]. Roughly 1.5 years later, in April 2014, Google announced that a TLS cipher suite using ChaCha20 (a variant of Salsa) for symmetric encryption will be their default option to secure HTTPS connections on devices without AES hardware acceleration [14].

random bit-sequence generators, stream ciphers, and authenticated encryption algorithms [9,12]. However, unlike a block cipher, a permutation does not have a key schedule and needs to be efficient only in one direction since the inverse permutation is normally never used. Past research in the area of permutation-based cryptography can be split into two main categories; the first is about the design (and security analysis) of permutation-based constructions and "modes of use" built on top of them, while the second category is concerned with the permutations themselves. Representative work in the former category includes besides the classical sponge [9] and duplex [10] construction also various kinds of constuctions/modes that aim to boost performance through a higher bitrate (e.g. full-state absorption [21], Beetle mode [15]) or via a parallel application of a sponge or a permutation (e.g. KangarooTwelve [13], Farfalle [8]), as well as modes with "built-in" countermeasures against certain physical attacks (e.g. Isap [18]). Research in the second category deals mainly with the design of permutations and their efficient (and side-channel resistant) implementation in hardware and/or software. The majority of the published permutations are either classical Addition-Rotation-XOR (ARX) designs, e.g. Salsa [6], or can be classified as "AndRX" variants, e.g. Keccak-f [11], Norx F^l [4].

Permutation-based cryptography is well suited for resource-limited devices (e.g. RFID tags, wireless sensor nodes, smart cards), which is evidenced by the fact that roughly half of the 32 second-round candidates of NIST's currently-ongoing standardization effort for lightweight cryptography use a permutation as underlying primitive [26]. However, despite a broad body of research in the area of permutation-based cryptography, surprisingly little is known about the performance of state-of-the-art permutations on small microcontrollers. There exist, of course, a lot of benchmarking results for the second-round candidates of NIST's lightweight crypto project[2], but these benchmarks specify only the execution time of the full authenticated encryption (resp. hash) algorithms and not that of the permutation alone. These timings are relatively poor indicators for the efficiency of the underlying permutation since they also include various "auxiliary" operations. For example, designs based on the Beetle mode, such as the NIST candidate Schwaemm [5], include a feedback function ρ through which data is injected into (and extracted from) the state. Furthermore, some optimized implementations of permutations that operate on 64-bit words, like Keccak-f[1600] and Ascon's p [19], adopt the bit-interleaving method [11] to speed up rotations on 32-bit ARM processors. This bit-interleaving makes the injection/extraction of data to/from the state more costly, whereby the actual penalty factor depends on how fast the permutation itself is. The benchmarks for full authenticated encryption or hash algorithms do not even allow one to reason about the *relative* efficiency of their permutations due to differences in the bit-rates. Unfortunately, the lack of detailed implementation results makes the design of new permutations a challenging task since it is not easily possible to compare the execution time and code size with the state-of-the-art.

[2] `http://github.com/usnistgov/Lightweight-Cryptography-Benchmarking/` (accessed 2021-09-10).

In this paper, we analyze and compare the multi-platform (resp. cross-platform) efficiency of four cryptographic permutations that are part of candidates of the current lightweight cryptography standardization project of the National Institute of Standards and Technology (NIST) [26]. These four candidates are ASCON [19], GIMLI [7], SCHWAEMM [5], and XOODYAK [16], all of which come with algorithms for Authenticated Encryption with Associated Data (AEAD) and hashing. In addition, they have in common that the permutation width is very similar (i.e. between 320 and 384 bits) and they all consist of only simple arithmetic/logic operations (SCHWAEMM is a classical ARX construction, while the other three can be classified as "AndRX" designs, i.e. they use the logical AND operation or OR operation as a source of non-linearity). We evaluate the execution time and code size of these four permutations with highly-optimized Assembler implementations for ARM Cortex-M3 and AVR ATmega128 microcontrollers, whereby we applied the same general optimization strategies and invested a similar amount of optimization effort for each implementation so as to ensure a fair evaluation. By focusing solely on the permutations, we aim to make their relative performance more transparent and generate new insights to their multi-platform efficiency, which are not immediately apparent when one compares the execution times collected by other benchmarking initiatives. We also assess how basic design decisions, e.g. shift/rotation amounts, impact the performance of the permutations on 32-bit ARM and 8-bit AVR platforms.

2 Overview of the Permutations

In this section, we briefly review the main properties of the four permutations we consider in this paper, which are the permutations of the NIST candidates ASCON, GIMLI, SCHWAEMM, and XOODYAK. Except for GIMLI, they all made it to the final round of the evaluation process [26]. GIMLI was eliminated in the second round, but we still include it in our study since its permutation is well known and has inspired a number of other designs.

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 [19]. It is based on the so-called Monkey Duplex mode [12] 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, and (iii) a permutation 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 permutation layer performs an operation of the form $x = x \oplus (x \ggg n_1) \oplus (x \ggg n_2)$ on each word x of the state with $n_1 \in \{1, 7, 10, 19, 61\}$ and $n_2 \in \{6, 17, 28, 39, 41\}$ [19].

Gimli. The second-round NIST candidate Gimli consists of the AEAD algorithm Gimli-Cipher and the hash function Gimli-Hash. Both are claimed to provide 128 bits of security against all known attacks, and Gimli-Cipher even uses a 256-bit key to "reduce concerns about multi-target attacks and quantum attacks" [7]. The underlying 384-bit permutation is called Gimli-24 and was presented at CHES 2017. Gimli-Cipher is a conventional duplex construction with a capacity of 256 bits, i.e. the rate is 128 bits. On the other hand, Gimli-Hash is an ordinary sponge and also uses a rate of 128 bits. Unfortunately, the permutation has weak diffusion, which makes it possible to build a full-round distinguisher of relatively low complexity [20]. Though this distinguisher on the permutation does not immediately threaten the security of Gimli-Cipher and Gimli-Hash, the NIST decided to not promote Gimli to the final round.

The Gimli-24 permutation was designed to reach high performance across a broad range of platforms, from high-end 64-bit CPUs with vector extensions to small 8-bit microcontrollers, as well as FPGAs and ASICs. Its 384-bit state is represented as a 3×4 matrix of 32-bit words. Each of the 24 rounds consists of three operations: (i) a non-linear layer in the form of a 96-bit SP-box that is applied to each column of the matrix, (ii) a linear mixing layer in every second round, and (iii) a constant addition in every fourth round. The SP-box itself is inspired by Norx and can be efficiently implemented using logical operations (32-bit AND, OR, and XOR), left shifts by 1, 2 and 3 bits, as well as rotations by 9 and 24 bits. On the other hand, the linear layer performs swap operations on row 0 of the matrix: a small-swap every fourth round (starting from round 1), and a big-swap also every fourth round (starting from round 3).

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 [15], 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 [5] 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 [5]. 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.

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 [17]. 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 [16]. 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 is inspired by both Keccak and Gimli 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 [11]. 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.

3 Implementation and Evaluation

To ensure a fair and consistent evaluation of the four permutations, we applied the same implementation and optimization strategy to each permutation, and we put a similar effort into optimizing each implementation. This section gives an overview of our optimization strategy for ARM and AVR, and presents some insights into the implementations and the benchmarking. In total, we evaluated eight implementations (four for ARM and also four for AVR), half of which we developed from scratch, namely the ARM implementation of Sparkle384 and the AVR implementations of Ascon, Sparkle384, and Xoodoo, whereas the remaining four are based on Assembler source code provided by the designers (with minor modifications to ensure a fair and consistent evaluation).

Target Platforms. The two concrete microcontrollers on which we evaluate the execution time and binary code size of the cryptographic permutations are a 32-bit ARM Cortex-M3 [1] and an 8-bit AVR ATmega128 [22]. They possess very different (micro-)architectural properties and features, making them good targets for an assessment of the multi-platform efficiency of permutations.

The Cortex-M3 is described in [1] as a "low-power processor" that combines low hardware cost with high code density and is intended for deeply embedded applications. It features a 3-stage pipeline with branch speculation and is based on a modified Harvard memory structure, which means data memory (SRAM) and instruction memory (usually flash) are connected through separate buses with the core, but the address space is unified. All Cortex-M3 microcontrollers implement the ARMv7-M architecture profile [2] and are, therefore, capable to execute the Thumb-2 instruction set. Thumb-2 is is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions, whereby instructions of different length (i.e. 16 bit, 32 bit) can be intermixed freely. There are 16 registers in total (`r0` to `r15`), of which up to 14 are available to the programmer and can serve as general-purpose registers for data operations. Arithmetic/logical instructions generally have a 3-operand format (i.e. two source registers and one destination register), though there are some restrictions for 16-bit Thumb-2 instructions, e.g. only the lower registers `r0` to `r7` can be used or the destination register and one of the source registers must be identical. On the other hand, many 32-bit data processing instructions support a "flexible" second operand, which means the second operand can be a 12-bit immediate value or a register with an optional shift or rotate [2]. This makes it possible to execute a shift or rotation as part of another instruction in a single clock cycle; for example, `add rd, rs1, rs2, ror #17` first rotates the content of `rs2` 17 bits left before adding it to `rs1` and assigning the sum to register `rd`. Thumb-2 allows for conditional execution of up to four instructions that immediately follow an "if-then" construct (i.e. an `it` instruction) and are suffixed with appropriate condition codes (see [2] for details).

The ATmega128 [22], like other members of the AVR family of 8-bit microcontrollers, is based on a modified Harvard architecture, which means it comes with separate memories and buses for program and data in order to maximize performance and parallelism. However, in contrast to Cortex-M3, ATmega128 microcontrollers have also separate address spaces for program and data (the ATmega128 only qualifies as a *modified* Harvard architecture in the weak sense that it provides dedicated instructions to read and write program memory as data, e.g. `lpm` and `spm`). In total, there are 133 instructions, which are encoded to be either 16 or 32 bits wide; most of them are executed in only one or two clock cycles. The ATmega128 features a 2-stage pipeline, making it possible to execute an instruction while the next instruction is fetched from the program memory. In addition, it comes with a large register file consisting of 32 general-purpose registers (`r0` to `r31`) of 8-bit width. Six registers can be used as three 16-bit pointers (`X`, `Y`, and `Z`) to access the data memory. The arithmetic/logical instructions have a 2-operand format, which allows them to read two operands from two (arbitrary) registers and write the result of the operation back to the first register [23]. The ATmega128, like most other 8-bit microcontrollers, can shift or rotate the content of a register by only one bit at a time; this implies that shifting an 8-bit operand by e.g. three bits takes three clock cycles. The cost increases accordingly for 32 or 64-bit operands. For example, the rotation of a 32-bit operand (stored in four registers) to the left by one bit requires the execution of five instructions (one `lsl`, three `rol`, and one `adc`) and takes five clock cycles. A 1-bit right-rotation of a 32-bit operand is even more expensive since it involves six instructions (one `bst`, four `ror`, and one `bld`).

Optimization Strategies. The evaluated assembly implementations for the Cortex-M3 platform are purely speed-optimized, which means whenever there was a trade-off to be made between execution time and code size, the decision was always in favor of the optimization that led to the best performance. This implies, for example, the full unrolling of the main loop of each permutation to eliminate the loop overhead and facilitate some other optimizations. Round constants are not kept in tables in flash or RAM, but loaded into registers on the fly via `movw` and `movt` (to reduce the impact of wait states) or, if they are short enough, directly encoded into an instruction as an immediate value. Such speed-optimized implementations have been developed by the designer teams of Ascon, Gimli, and Xoodoo; we used these assembly implementations as starting point but made small modifications to increase the readability of the source code (e.g. by using macros) and to ensure that they all adhere to the specifications of the ARM Application Binary Interface (ABI). We translated the assembly source code of Gimli from the GNU assembler (GAS) syntax to the syntax of Keil MicroVision such that its execution time can be determined with Keil's cycle-accurate simulator and by execution on development boards using the GNU toolchain for ARM. The original 32-bit ARM implementation of Ascon provided by its designers contained "inlined" assembly code for the permutation. We converted this implementation into a pure assembly function (with a separate file) to ensure consistency across all permutations. Finally, the fourth permutation, i.e. Sparkle384, was implemented by us from scratch.

Our assembly implementations of the permutations for the 8-bit AVR platform [23] aim for small (binary) code size instead of high speed. Therefore, we refrained from code-size increasing optimizations like (full) loop unrolling as otherwise the code size may grow unreasonably large. This can be exemplified using the AVR assembler implementations of Gimli (provided by its designers) as case study. One of these implementations is size-optimized and, thus, quite small (less than 800 B), while the other is speed-optimized (with fully unrolled main loop) and has a code size of over 19 kB [7]. For comparison, the code size of the fully-unrolled ARM implementation is less than 4 kB. However, it has to be taken into account that the flash capacity for storing program code is, in general, more restricted on small and cheap devices that are equipped with an 8-bit microcontroller than on devices with a more powerful 32-bit ARM microcontroller. We implemented the assembly code for Ascon, Sparkle384, and Xoodoo from scratch since, at the time we started with our evaluation of the permutations, no size-optimized AVR implementations were available. On the other hand, we took over the small-size AVR version of the Gimli permutation developed by the designer team since it aligns very well with our optimization strategy for AVR. We put a similar effort into optimizing each implementation of the permutations to ensure a fair and consistent evaluation.

Implementation Details. Optimizing the permutations for the 32-bit ARM Cortex-M3 microcontroller is fairly straightforward. All four permutations have in common that the full state can be kept in the register file, which still leaves either two (Gimli, Sparkle384, Xoodoo) or four (Ascon) registers available for the computations. Gimli, Sparkle384, and Xoodoo organize their state in 32-bit words and can, therefore, take advantage of implicit shifts/rotations folded into data processing instructions. All implementations we evaluate make extensive use of such "free" shifts or rotations so as to minimize the execution time. As already mentioned above, round constants are either directly encoded into an instruction as immediate value (if they are short enough) or loaded to registers via `movw` and `movt`. The permutation of Ascon is a special case since it operates on 64-bit words. In order to still be able to exploit "free" shifts and rotations of 32-bit operands, the designer team of Ascon adopted a so-called *bit-interleaving* technique [11,19], which is, in essence, a special representation of a 64-bit word as two 32-bit words, one containing all bits at even positions and the other all bits at odd positions. In this way, Ascon can take advantage of implicitly-performed rotations in the linear layer, though this comes at the expense of conversions between the normal representation and bit-interleaved representation. More concretely, data that is injected into the state has to be converted from normal to bit-interleaved form, while an extraction of data from the state requires a conversion in the opposite direction.

The 8-bit AVR architecture requires significantly different implementation and optimization techniques than ARM. First and foremost, the register space of an 8-bit AVR microcontroller is not large enough to accommodate the entire state of any of the four permutations, which means the state has to be kept in

RAM and parts of the state are loaded into registers when an operation is to be carried out on them. Therefore, the main optimization problem for AVR is to find a good register allocation strategy, which includes to decide when to load state-words from RAM to registers and when to write them back to RAM so that the overall number of memory accesses (i.e. `ld`, `st` instructions) becomes minimal. Ascon is well suited for platforms with small register space because each of the two main layers needs, at any time, only a part of the 320-bit state (but never the full state) in registers. Our AVR implementation processes the substitution layer in 16-bit slices (i.e. a 16-bit part of each state-word is loaded and stored) and the permutation layer one state-word at a time, which means each byte of the state gets loaded/stored twice per round. This is also the case for Sparkle384, but requires moving the computation of the temporary values t_x and t_y from the linear layer to the ARX-box layer. Our AVR implementation of Xoodoo integrates parts of the plane-shifting layers ρ_{west} and ρ_{east} into the mixing layer θ and the non-linear layer χ, respectively, to minimize the overall number of memory accesses. Nonetheless, each byte of the 384-bit state has to be loaded from RAM and stored to RAM on average 2.66 times per round.

Rotations of 32-bit (resp. 64-bit) words can be optimized on AVR by taking advantage of the fact that rotating by a multiple of 8 bits is cheap (i.e. can be executed by `mov` instructions) or even free (e.g. when combined with XOR).

Benchmarking. We evaluated the execution time of both the ARM and the AVR implementation of the permutations via simulation with a cycle-accurate instruction set simulator, namely the simulator of Keil MicroVision 5.26 and Atmel Studio 7, respectively. Execution times obtained by simulation with the latter are, in general, very close to the timings on a real AVR device. Unfortunately, this is often not the case for simulation results for ARM since, as stated on Keil's website[3], the simulator assumes ideal conditions for memory accesses and "does not simulate wait states for data or code fetches." Thus, the timings obtained with this simulator should be seen as lower bounds of the execution times one will observe on a real Cortex-M3 device. In order to get more precise performance figures, we also measured the execution time of the permutations on three development boards with a different number of flash wait states. The first board is a STM32 VL Discovery, which is equipped with a STM32F100RB Cortex-M3 microcontroller clocked at a nominal frequency of 24 MHz. Due to this relatively low clock frequency, the microcontroller can access flash with no wait states at all. Our second board is also a STM32, but a more sophisticated one, namely the Nucleo-64. It comes with a STM32F103RB Cortex-M3 clocked with a frequency of 72 MHz. At this clock frequency, flash accesses require two wait states. Finally, the third board is an Arduino Due, which houses an Atmel SAM3X8E Cortex-M3 clocked at 84 MHz. When operated using the standard configuration, flash accesses require four wait states. However, the performance impact of this high number of wait states is partly mitigated by a 128-bit wide memory interface and a system of 2×128-bit buffers (see [3, Sect. 18]).

[3] `http://www2.keil.com/mdk5/simulation/` (accessed 2021-09-14).

4 Experimental Results

Table 1 presents the code size and execution time of speed-optimized (i.e. with fully-unrolled loops) ARM assembly implementations of the four permutations Ascon, Gimli, Sparkle384, and Xoodoo. All these execution times are the result of simulations using the cycle-accurate instruction set simulator of Keil MicroVision 5.26 using a generic Cortex-M3 model as target device. The times range from 387 clock cycles (Ascon) to 1041 cycles (Gimli). However, when comparing symmetric cryptosystems, the throughput (in cycles per byte) is, in general, more meaningful than raw execution times. For example, in the case of block ciphers, the throughput obtained by dividing the execution time by the block size allows one to take into account that different algorithms may have different block sizes. Similarly, when comparing permutations, one can obtain throughput figures by dividing the computation time by either the width of the permutation or the rate of the associated AEAD algorithm. The AEAD rates that are relevant for our four permutations are all different, namely eight bytes for Ascon-128, 16 bytes for Gimli-Cipher, 24 bytes for Xoodyak, and even 32 bytes for Schwaemm256-128. However, when using the rate of the related AEAD algorithm to determine the throughput, the resulting values take into account the efficiency of the permutation *and* the efficiency of the mode of the AEAD algorithm. Since our aim is to analyze the efficiency of the permutation alone, we decided to calculate the throughput under the assumption that each permutation is used to instantiate one and the same mode (namely a classical sponge) with one and the same capacity (namely 256 bits, which corresponds to 128 bits of security). Consequently, Ascon has a rate of eight bytes, and the three other permutations a rate of 16 bytes.

Table 1. Code size, execution time, and throughput of speed-optimized ARMv7-M assembly implementations of the four permutations on a Cortex-M3 microcontroller.

Permutation	Code size (bytes)	Exec. time (clock cycles)	Throughput (cc/rate-byte)
Ascon-128 (6 rounds)	1364	387	48.38
Gimli (24 rounds)	3950	1041	65.06
Sparkle384 (7 steps)	2810	778	48.63
Xoodoo (12 rounds)	2376	657	41.06

The last column of Table 1 gives the throughput (in cycles per byte) of the permutations calculated in this way, i.e. by dividing the execution time by the rate under the assumption that the permutation is used to instantiate a sponge with a capacity of 256 bits. Xoodoo requires only 41 cycles per rate-byte and reaches the best throughput, followed by Ascon and Sparkle384, which are nearly identical. However, the results for Ascon do not include the conversion to and from bit-interleaved form. Gimli has the by far worst throughput of all four permutations. In terms of code size, Ascon is the clear winner.

Table 2. Code size, execution time, and throughput of size-optimized AVR assembly implementations of the four permutations on an ATmega128 microcontroller.

Permutation	Code size (bytes)	Exec. time (clock cycles)	Throughput (cc/rate-byte)
Ascon-128 (6 rounds)	836	4484	560.50
Gimli (24 rounds)	778	23699	1481.19
Sparkle384 (7 steps)	844	7460	466.25
Xoodoo (12 rounds)	756	11849	740.56

Table 2 lists the code size, execution time, and throughput (in terms of the permutation time divided by the rate, assuming a capacity of 256 bits) of code-size-optimized AVR assembly implementations of the four permutations on an ATmega128 microcontroller [22]. The execution times were simulated using the cycle-accurate instruction set simulator of Atmel Studio 7. Apparently, all the AVR timings are significantly worse (by at least one order of magnitude) than the execution times of the permutations on ARM. This enormous performance penalty can be explained by different optimization goals (i.e. size versus speed) and, more importantly, the completely different characteristics of the AVR and ARM architecture (e.g. register space, latency of multi-bit rotations). In terms of throughput, Sparkle384 is now the clear winner, followed by Ascon and Xoodoo. While on ARM the three fastest permutations were throughput-wise relatively close to each other, we see a significant difference on AVR since the throughput of Ascon is 20% worse than the throughput of Sparkle384, and the throughput of Xoodoo is even 59% worse. Even though we optimized the permutations for small code size, they compare very well with speed-optimized AVR implementations. For example, the AVR assembler implementation of the permutation of Ascon developed by Rhys Weatherley[4] has an execution time of 4693 cycles and a code size of 1418 bytes, which means our implementation is not only much smaller but also a bit faster. The AVR implementation of the Xoodoo permutation provided by its designers[5] needs 11009 clock cycles for 12 rounds and has a code size of 1656 bytes, making it more than twice as big as our implementation, but also 840 clock cycles (or 7.6%) faster.

As mentioned in the last section, the simulation results obtained with Keil MicroVision can differ from the execution time on "real" Cortex-M3 hardware because the simulator does not take flash wait states into account. The purpose of such flash wait states is to compensate the difference between the maximum clock frequency with which the microcontroller core and the flash memory can be operated. Modern Cortex-M3 microcontrollers can be clocked with frequencies of over 200 MHz, which is far above the maximum frequency of conventional flash memory (usually between 20 and 30 MHz). Thus, it makes sense to assess

[4] `ascon_permute` from `http://github.com/rweather/lwc-finalists/blob/master/src/individual/ASCON/internal-ascon-avr.S` (accessed 2021-09-21).

[5] `Xoodoo_Permute_Nrounds` from `http://github.com/XKCP/XKCP/blob/master/lib/low/Xoodoo/AVR8/Xoodoo-avr8-u1.s` (accessed 2021-09-21).

Table 3. Execution time of the four permutations as determined by simulation with Keil MicroVision using a generic Cortex-M3 model and measurement on Cortex-M3 development boards with 0, 2, and 4 flash wait states (values in parentheses are the performance penalties versus the VL Discovery board, which has 0 wait states).

Permutation	Keil μVision (simulation)	VL Discovery 0 wait states	Nucleo-64 2 wait states	Arduino Due 4 wait states
Ascon-128 (6 rounds)	387	389	601 (1.54)	472 (1.21)
Gimli (24 rounds)	1041	1043	1656 (1.59)	1287 (1.23)
Sparkle384 (7 steps)	778	780	1196 (1.53)	936 (1.20)
Xoodoo (12 rounds)	657	659	1014 (1.54)	795 (1.21)

the impact of flash wait states on the actual performance of the permutations by measuring their execution time on the three Cortex-M3 development boards mentioned in the previous section, namely an STM32 VL Discovery (which has no flash wait states), an STM32 Nucleo-64 (two wait states), and an Arduino Due (four wait states). However, the Atmel SAM3X8E microcontroller on the Due board performs fetches from flash through a 128-bit wide bus and comes with a 2×128-bit buffer, which mitigates to a certain extent the impact of the wait states. Table 3 shows the (measured) execution times of the permutations on these boards. The timings on the VL Discovery are almost the same as the ones obtained through simulation with Keil MicroVision; this confirms that the Keil simulator is indeed cycle-accurate. On the other hand, the execution times on the Nucleo-64 board are significantly worse (by factors of between 1.53 and 1.59) than the results on the Discovery board and the timings reported by the simulator. The timings on the Arduino Due are better than the timings on the Nucleo-64, despite the two times larger number of wait states, which is because of the afore-mentioned 128-bit wide flash access and the 2×128-bit buffer.

5 Conclusions

Since there is no single dominating microcontroller architecture in the IoT, the designers of (lightweight) symmetric algorithms have to aim for multi-platform efficiency, i.e. efficiency on a wide range of microcontrollers with highly diverse (and divergent) characteristics. In this paper, we analyzed how well the permutations of the AEAD algorithms Ascon-128, Gimli, Schwaemm256-128, and Xoodyak achieve this goal, whereby we used a 32-bit ARM Cortex-M3 and an 8-bit AVR microcontroller as target platforms. We evaluated speed-optimized assembler implementations for ARM, based primarily on source code from the designer teams, and size-optimized assembler implementations for AVR, which we mainly developed from scratch. Our results indicate that the throughput (in terms of permutation time divided by the rate when the capacity is fixed to 256 bits) of Ascon, Sparkle384 and Xoodoo is very similar on ARM and differs by just a few cycles per rate-byte. On the other hand, on AVR, Sparkle384 is significantly more efficient than all its competitors; for example, it outperforms

Ascon and Xoodoo by a factor of 1.20 and 1.59, respectively. A major reason for the difference between ARM and AVR results is the cost of multi-bit shifts and rotations on the latter platform. Many of the rotation amounts of the five linear functions of Ascon are not particularly AVR-friendly, which makes the linear layer relatively inefficient. The performance of Xoodoo on AVR is also hampered by rotation amounts that are "unfriendly" to small microcontrollers and further suffers from a relatively large number of memory accesses compared to e.g. Ascon. On a more positive note, the results for Sparkle demonstrate that it is possible to design a permutation for multi-platform efficiency.

Acknowledgements. The second author was supported, in part, by the Fonds National de la Recherche Luxembourg under CORE grant C19/IS/13641232.

References

1. Arm Limited. ARM Cortex-M3 Processor Technical Reference Manual, Revision r2p1. `http://developer.arm.com/documentation/100165/latest`, 2016.
2. Arm Limited. ARMv7-M Architecture Reference Manual, Issue E.e. `http://developer.arm.com/documentation/ddi0403/latest`, 2021.
3. Atmel Corporation. SAM3X/SAM3A Series Atmel SMART ARM-based MCU. Data sheet, `http://www.microchip.com/en-us/product/ATSAM3X8E`, 2015.
4. J.-P. Aumasson, P. Jovanovic, and S. Neves. NORX v3.0. Specification, `http://github.com/norx/resources/raw/master/specs/norxv30.pdf`, 2016.
5. C. Beierle, A. Biryukov, L. Cardoso dos Santos, et al. Lightweight AEAD and hashing using the Sparkle permutation family. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, June 2020.
6. D. J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs – The eSTREAM Finalists*, vol. 4986 of *Lecture Notes in Computer Science*, pp. 84–97. Springer, 2008.
7. D. J. Bernstein, S. Kölbl, S. Lucks, et al. Gimli: A cross-platform permutation. In *Cryptographic Hardware and Embedded Systems — CHES 2017*, vol. 10529 of *Lecture Notes in Computer Science*, pp. 299–320. Springer, 2017.
8. G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Farfalle: Parallel permutation-based cryptography. *IACR Transactions on Symmetric Cryptology*, 2017(4):1–38, Dec. 2017.
9. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. `http://keccak.team/files/CSF-0.1.pdf`, 2011.
10. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography — SAC 2011*, vol. 7118 of *Lecture Notes in Computer Science*, pp. 320–337. Springer, 2011.
11. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, version 3.0. `http://keccak.team/files/Keccak-reference-3.0.pdf`, 2011.
12. 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)*, pp. 159–170, 2012.

13. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier. KangarooTwelve: Fast hashing based on Keccak-p. In *Applied Cryptography and Network Security — ACNS 2018*, vol. 10892 of *Lecture Notes in Computer Science*, pp. 400–418. Springer, 2018.
14. E. Bursztein. Speeding up and strengthening HTTPS connections for Chrome on Android. Google Security Blog, `http://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html`, 2014.
15. 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.
16. 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.
17. 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.
18. C. Dobraunig, M. Eichlseder, S. Mangard, et al. Isap v2.0. *IACR Transactions on Symmetric Cryptology*, 2020(S1):390–416, June 2020.
19. 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.
20. A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. Internal symmetries and linear properties: Full-permutation distinguishers and improved collisions on Gimli. *Journal of Cryptology*, 34(4):45, Oct. 2021.
21. B. Mennink, R. Reyhanitabar, and D. Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In *Advances in Cryptology — ASIACRYPT 2015*, vol. 9453 of *Lecture Notes in Computer Science*, pp. 465–489. Springer, 2015.
22. Microchip Technology Inc. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash: ATmega128, ATmega128L. `http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf`, 2011.
23. Microchip Technology Inc. AVR Instruction Set Manual. `http://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf`, 2020.
24. Mordor Intelligence, Inc. 8-bit Microcontroller Market – Growth, Trends, and Forecast (2020–2025). `http://www.mordorintelligence.com/industry-reports/8-bit-microcontroller-market-industry`, 2020.
25. National Institute of Standards and Technology (NIST). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS Publication 202, `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`, Aug. 2015.
26. National Institute of Standards and Technology (NIST). Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process. Internal Report 8369, `http://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf`, July 2021.
27. Radiant Insights, Inc. Microcontroller Market Size, Share, Analysis Report 2020. `http://www.radiantinsights.com/research/microcontroller-market/`, 2015.
28. Telefonaktiebolaget LM Ericsson. Ericsson Mobility Report November 2017. `http://www.ericsson.com/assets/local/mobility-report/documents/2017/ericsson-mobility-report-november-2017.pdf`, 2017.