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 5 wait states. Our results show that the throughput (in terms of permutation time divided by encryption rate) of the permutation of Ascon, Schwaemm, and Xoodyak is similar on ARM and lies in the range of between 24.4 and 29.1 cycles per rate-byte. However, on AVR, the permutation of Schwaemm clearly outperforms Ascon and Xoodyak by a factor of 1.55 and 2.10, 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 miniature sensor nodes at the opposite end of the spectrum commonly feature

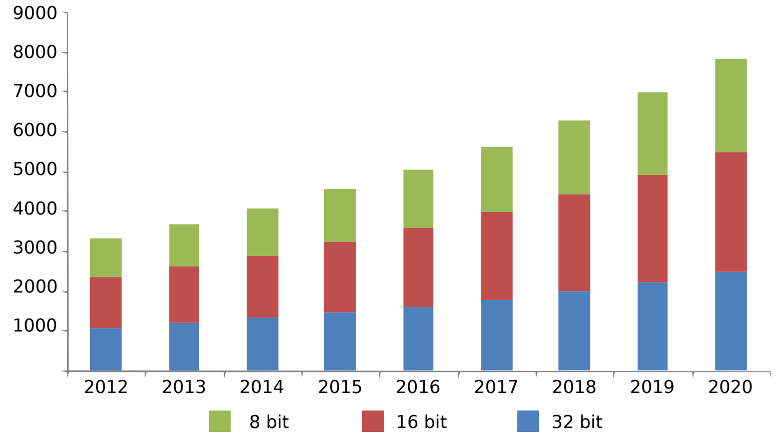

Fig. 1. North American microcontroller market by product (8-bit, 16-bit, 32-bit) in million units (source: Radiant Insights Inc. [7])

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 [5]. Internet-enabled devices can nowadays be found in nearly 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) micro-controller 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 [7]. 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 a high level of precision in embedded processing and the development of intelligent and real-time functions in the automotive industry [7]. 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 [6].

Since there is no single dominating microcontroller platform in the IoT, it is essential that cryptographic algorithms achieve consistently good performance on a wide variety of 8, 16, and 32-bit architectures. This is far from trivial since, for example, a 32-bit ARM Cortex-M3 microcontroller has much different architectural and micro-architectural characteristics than an 8-bit AVR ATmega microcontroller. The former 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 take 8 bits of data, which means the usable register space is 256 bits. ARM and AVR also differ significantly in their ability to execute multi-bit shifts and rotations, which are performance-critical operations of many symmetric cryptosystems. The arithmetic/logic unit of ARM microcontrollers features a fast barrel shifter capable to shift or rotate a 32-bit word by an arbitrary amount of bits in a single clock cycle. Furthermore, a shift or rotation can be combined with many other arithmetic/logical instructions, which means multi-bit shifts and rotations are essentially free on ARM. The situation is completely different for 8 and 16-bit architectures since most of them have only single-bit shift and rotation instructions, which means that shifting or rotating a register by n bits takes (at least) $n + 1$ clock cycles. Hence, performing a shift or rotation of a 32-bit word on AVR can, in the worst case, require dozens of clock cycles.

In this paper we analyze and compare the multi-platform efficiency of four families of cryptographic permutations that form part of second-round candidates in the current lightweight cryptography standardization project of the US National Institute of Standards and Technology (NIST). These second-round candidates are Ascon [4], Gimli [2], Schwaemm [1], and Xoodyak [3], all of which do not only provide Authenticated Encryption with Associated Data (AEAD) but also hashing. Furthermore, they have in common that the size of their state is similar (i.e. between 320 and 384 bits) and they all consist of only simple arithmetic/logical operations (Schwaemm is a classical ARX construction, while Ascon, Gimli, and Xoodyak can be characterized as "AndRX" designs, which means they use the logical AND operation as a source of non-linearity). We evaluate the execution time and code size of the four permutations with highly-optimized Assembler implementations for ARM Cortex-M3 and AVR ATmega microcontrollers, whereby we applied the same general optimization strategies and invested a similar amount of optimization effort for each implementation to ensure a fair and consistent evaluation. By focusing solely on the permutations, we aim to make their relative performance more transparent and produce new insights into their multi-platform efficiency, which are not immediately apparent when comparing the execution times collected by other benchmarking initiatives.

2 Overview of the Permutations

In this section, we briefly summarize the main properties of the four permutations we consider in this paper.

Ascon. The main components of Ascon, Ascon-Hash, and Ascon-XOF are two 320-bit wide permutations (called in their specification p^a and p^b) that apply an SPN-based round transformation to Ascon's internal state. The permutation operates over the state as five 64-bit words and is composed of three parts: A 8-bit round constant addition; a substitution layer that applies a bit-wise 5-bit S-box; and a linear diffusion layer over the 64-bit words. The S-Box is normally implemented via bit-wise logical operations and is composed of AND, XOR, and NOT operations, while the linear layer is composed of XOR and two different rotations on the 64-bit wide words. Furthermore, the S-box is based on the χ mapping of Keccak, and the linear layer is based on the Σ functions used in SHA-2.

This permutation is inherently bitsliced and resistant to cache-timing attacks, as it can be implemented without lookup tables using only Boolean operations and rotations [4]

Gimli. Gimli uses a 384-bit bit permutation, designed for achieving security and performance on a wide range of platforms, from 64-bit server-class CPUs to small 8-bit microcontrollers, as well as FPGAs and ASICs [2].

The permutation applies a sequence of rounds to its state, which is viewed as a 3×4 matrix of 32-bit words (Or a $3 \times 4 \times 32$ cuboid), with each round composed of:

1. A non-linear layer, composed of a 96-bit SP-Box applied to each column, composed of three sub-operations: A 32-bit wide rotation of two words; a nonlinear 3-input T-function composed of XOR, AND, OR and shift operations; and lastly a word swap.
2. A linear mixing layer that is applied every second round, which is made of two different swaps: One between adjacent columns, and one every fourth round between non-adjacent columns.
3. A constant addition every fourth round, where a counter and a round constant are added to the first state word.

Sparkle. The AEAD algorithm Schwaemm and the hash function Esch use the variants of the Sparkle permutation.

Sparkle defines a family of permutations over blocks with 256, 384, and 512 bits, parameterized by a number of steps, analogous to a round number. This permutation is analogous to a classical Substitution-Permutation Network (SPN), except that the functions playing the role of the S-Boxes are different in each branch. The permutation consists of two main components:

1. An ARX-Box, called Alzette, that operates as a 64-bit block cipher with a 32-bit key. This block cipher is a four-round iterated block cipher, where each round has a different rotation amounts. Beyond that, Alzette uses 8-bit friendly rotations over the 32-bit words, XOR operations, and 32-bit modular addition.
2. A linear diffusion layer, which draws on the one used by Sparx-128. It is composed of a Feistel round, and by a branch-wide permutation, both of which can be implemented with XOR and shift operations.

The design follows the classical SPN constructions and lends itself well to SIMD implementations [1].

Xoodoo. Xoodoo is a family of 384-bit permutations, operating over twelve 32-bit words, parameterized by the number of rounds. It defines the internal state as a $3 \times 4 \times 32$ bit matrix, and is composed of a round function with five main components: A mixing layer θ composed of XOR and a cyclic shift of planes, two plane shifts ρ_{west} and ρ_{east}, a round constant addition, and a non-linear layer χ, this last one composed of AND and XOR operations. This construction is claimed to be efficient in low-end processors and is used as the core for Xoodyak and Xoofff [3].

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 strategies for ARM and AVR, and describes how we benchmarked the permutations.

The assembly implementations for the ARM Cortex-M3 platform are purely speed-optimized, which means whenever there was a trade-off to be made between execution time and code size, we opted for the optimization that led to the best performance. This means, for example, that the main loop of each permutation is fully unrolled to eliminate the loop overhead. Round constants are not kept in tables in flash or RAM, but put into registers on the fly using `movw` and `movt` instructions or, if they are less than 12 bits long, directly encoded into the instruction as an intermediate value. Such speed-optimized implementations have been developed by the designers of the permutations; we used these implementations as starting point and made some small modifications to increase the readability of the source code (e.g. by using macros) and ensured that they all adhere to the specification of the ARM Application Binary Interface (ABI). For example, the ABI specification requires that the stack pointer is double-word (i.e. 8 bytes) aligned at a public interface; when necessary we modified the source code to ensure full ABI compliance. We also translated the assembler source code of Gimli from the GNU assembler syntax to the syntax used by Keil MicroVision so 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 ARM implementation of Ascon provided by its designers was written in the form of an "inlined" assembly code for the permutation. We converted this implementation into a pure assembly function to ensure consistency across all four permutations.

Our assembly implementations of the permutations for the 8-bit AVR architecture aim for small (binary) code size instead of high speed. Therefore, we refrained from code-size increasing optimization techniques like (full) loop unrolling as otherwise, the code size would become unreasonably large. This can be exemplified using the AVR assembler implementations of Gimli (provided by its designers) as a case study. One of these implementations is size-optimized and, therefore, relatively small (less than 800 bytes), whereas the other is speed-optimized (with fully unrolled loops) and has a code size of more than 19 kB. 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 flash capacity for storing program code is, in general, more scarce on small and cheap devices with an 8-bit microcontroller than on devices equipped with a more powerful 32-bit ARM microcontroller. We developed the assembly implementations of Ascon, Sparkle, and Xoodoo from scratch since at the time we started with our evaluation of the permutations, no optimized AVR assembler code existed for them. However, we took over the size-optimized AVR implementation of the Gimli permutation developed by its designers since it complies with our optimization strategy. We put a similar effort into optimizing the AVR implementation of the permutations to ensure a fair and consistent evaluation.

We evaluated the execution time of both the AVR and the ARM implementations through simulation with a cycle-accurate instruction set simulator, namely the simulator contained in Atmel Studio 7 and Keil MicroVision 5.24, respectively. Execution times obtained by simulation with Atmel Studio are, in general, very close to the timings on "real" hardware. Unfortunately, this is usually not the case for simulation results for ARM since, as mentioned on the Keil website[1], the simulator assumes ideal conditions for memory accesses and "does not simulate wait states for data or code fetches." Therefore, the timings obtained with the simulator should be seen as lower bounds of the actual execution times one will get on a real Cortex-M3 device. In order to get realistic 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 the STM32 VL Discovery, which is equipped with an STM32F100RBT6B Cortex-M3 microcontroller clocked with a nominal frequency of 24 MHz. Due to this relatively low clock frequency, the microcontroller can access flash memory without wait states. Our second board is again an STM32 board, but a more sophisticated one, namely the STM32 Nucleo-64. It comes with an STM32F103RBT6 Cortex-M3 microcontroller clocked with a frequency of 72 MHz. At this frequency, flash accesses require two wait states. Finally, the third board is an Arduino Due, which houses an Atmel SAM3X8E Cortex-M3 microcontroller clocked with a

[1] See `https://www2.keil.com/mdk5/simulation` (accessed 2020-09-14)

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 (cycles)	Time/rate (cycles/byte)
ASCON128a (8 rounds)	1928	466	29.13
Gimli (24 rounds)	3950	1041	65.06
SPARKLE384 (7 steps)	2820	781	24.40
Xoodoo (12 rounds)	2376	657	27.38

frequency of 84 MHz. When operated with its standard configuration, flash accesses require 5 wait states. However, the performance impact of this high number of wait states is, to some extent, mitigated by a "flash accelerator."

4 Results

Table 1 compares the execution time and code size of speed-optimized (i.e. fully unrolled) assembly implementations of the four permutations ASCON128A, GIMLI, SPARKLE384, and XOODOO. These execution times have been determined through simulation with the cycle-accurate instruction set simulator of Keil MicroVision 5.24 using a generic Cortex-M3 model as the target device. The times range from 466 clock cycles (for ASCON128A) to 1041 clock cycles (GIMLI). However, more meaningful than the raw execution time is the execution time divided by the rate since this throughput determines the actual performance of the corresponding authenticated encryption algorithm. The last column of Table 1 specifies the throughput (in cycles per byte) of the permutations, which we calculated by dividing the execution time of the permutation by the rate[2] of the main instance of the corresponding AEAD algorithm (16 bytes for ASCON128A and GIMLI, 32 bytes for SCHWAEMM256-128, and 24 bytes for XOODYAK). SPARKLE384 achieves the highest throughput, closely followed by XOODOO and ASCON128A. GIMLI reaches less than half of the throughput of the other three permutations, but it has to be taken into account that the GIMLI AEAD algorithm aims for 256 bits of security. In terms of code size, ASCON128A is the clear winner, while GIMLI has the largest code size and is more than twice as big as ASCON128A.

Table 2 contains the code size, execution time, and throughput (in terms of permutation time divided by the rate) of code-size optimized AVR assembly implementation of the four permutations on an 8-bit ATmega128 microcontroller. The execution times were simulated using the cycle-accurate instruction set simulator that is part of Atmel Studio 7. Apparently, the obtained AVR timings are significantly worse (by at least an order of magnitude) than the execution times of the permutations on ARM. This massive performance penalty can be explained

[2] We used the rate of the encryption operation to calculate the throughput. Note that XOODYAK processes associated data at a different rate than plaintext and ciphertext.

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 (cycles)	Time/rate (cycles/byte)
ASCON128a (8 rounds)	898	6442	402.63
Gimli (24 rounds)	778	23699	1481.19
SPARKLE384 (7 steps)	702	8318	259.94
Xoodoo (12 rounds)	954	13091	545.46

by the different optimization goals (i.e. size vs. speed) and, more importantly, by the completely different characteristics of the architectures as mentioned in Section 1 (e.g. register space, latency of multi-bit shifts and rotations). In terms of throughput, Sparkle384 is again the winner, but Ascon128a and Xoodoo swapped their position compared to the ARM results, i.e. Ascon128a achieves the second-best throughput and Xoodoo is on the third place. However, while on ARM these three permutations were throughput-wise relatively close to each other, we see a significant difference in AVR since the throughput of Ascon128a is 1.55 times worse than the throughput of Sparkle384, and the throughput of Xoodoo is even more than two times worse. We emphasize again that these results are based on size-optimized implementations, which means all four permutations can reach better throughput rates when optimized for speed. Such speed-optimized implementations were developed in the course of Rhys Weatherley's benchmarking project for AVR and are available online[3]. Interestingly, these benchmarking results indicate that the *relative* performance of the corresponding AEAD algorithms is very similar to our throughput results for the permutations, namely Schwaemm256-128 is significantly faster than Ascon128a and Xoodoo.

As mentioned in the previous section, the simulation results obtained with Keil MicroVision may differ from the execution time on "real" Cortex-M3 hardware since the Keil simulator does not take flash wait states into account. The purpose of flash wait states is to compensate for the difference of the maximum clock frequency with which the microcontroller core and the flash memory can be clocked. Modern Cortex-M3 microcontrollers can reach clock frequencies of more than 200 MHz, which is far above the maximum frequency of flash memory (usually between 16 to 32 MHz). Therefore, we decided to assess the impact of flash wait states on the performance of the four permutations by measuring their execution time on the three Cortex-M3 development boards mentioned in the previous section, namely an STM32 VL Discovery (no flash wait states), an STM32 Nucleo-64 (2 flash wait states), and an Arduino Due (5 flash wait states). However, the SAM3X8E microcontroller on the Arduino board contains a "flash accelerator," which is essentially a small buffer located between the microcontroller core and the flash memory, to mitigate the impact of the wait

[3] See `https://rweather.github.io/lightweight-crypto/performance_avr.html` (accessed 2020-09-14).

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 5 flash wait states (values in parentheses are the performance penalties over the execution time on the VL Discovery board, which has 0 flash wait states).

Permutation	Keil μVision (simulation)	VL Discovery 0 wait states	Nucleo-64 2 wait states	Arduino Due 5 wait states
ASCON128a (8 rounds)	466	467	748 (1.60)	571 (1.22)
Gimli (24 rounds)	1041	1043	1656 (1.59)	1287 (1.23)
SPARKLE384 (7 steps)	781	782	1196 (1.53)	936 (1.20)
Xoodoo (12 rounds)	657	659	1014 (1.54)	795 (1.21)

states. Table 3 shows the measured execution times of the four permutations on these boards. The timings on the VL Discovery are almost identical to those obtained through simulation with Keil MicroVision, which 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 1.60) than the results on the Discovery board and the timings reported by the simulator. These results also show that flash wait states do not impact each permutation to the same extent since the penalty factor for ASCON128a is higher than the penalty factor for Sparkle384. The timings on the Arduino Due are better than the timings on the Nucleo-64, despite the larger number of wait states, which is due to the afore-mentioned flash accelerator.

5 Conclusions

Since there is no single dominating platform in the IoT, designers of lightweight cryptographic algorithms have to aim for multi-platform efficiency, i.e. efficiency on a wide range of microcontroller architectures with highly diverse and divergent characteristics. In this paper, we analyzed to what extent the permutations of the NIST candidates Ascon, Gimli, Schwaemm, and Xoodyak achieve this goal, whereby we used 32-bit ARM Cortex-M3 and 8-bit AVR as evaluation platforms. We benchmarked speed-optimized assembler implementations for ARM, using source code provided by the designers, and size-optimized assembler implementations for AVR, which we mainly developed from scratch. Our results show that the throughput (i.e. permutation time divided by rate) of Ascon, Sparkle, and Xoodoo is very similar on ARM and differs only by a few cycles per byte. On the other hand, on 8-bit AVR, Sparkle significantly outperforms Ascon and Xoodoo by a factor of 1.55 and 2.10, respectively. One reason for this discrepancy between the ARM and AVR results is the cost of multi-bit shifts and rotations, which has a significant impact on the overall execution time. ARM processors are equipped with a fast barrel shifter capable to execute multi-bit shift or rotation operations in one clock cycle. Sifts and rotations on ARM can even be combined with other arithmetic or logical operations and executed in a single cycle, which makes them essentially free. On the other hand, most small 8 and 16-bit microcontrollers, including the AVR Atmega128, do not have a fast barrel shifter and can, therefore, only shift or rotate the content of a register by one bit at a time, which makes multi-bit shifts and rotations of 32 or 64-bit words extremely slow. Our benchmarking results indicate that the designers of Ascon and Xoodoo either "over-optimized" their permutations for ARM, or they neglected efficiency on small 8 and 16-bit microcontrollers. On a more positive note, the results for Sparkle demonstrate that it is possible to design a permutation for multi-platform efficiency.

References

1. C. Beierle, A. Biryukov, L. C. 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*, pages 208–261, 2020.
2. D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, et al. Gimli: a cross-platform permutation. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 299–320. Springer, 2017.
3. J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. Xoodoo cookbook. *IACR Cryptol. ePrint Arch.*, 2018:767, 2018.
4. C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.
5. Ericsson. Ericsson Mobility Report November 2017. Available for download at `http://www.ericsson.com/assets/local/mobility-report/documents/2017/ericsson-mobility-report-november-2017.pdf`, 2017.
6. Mordor Intelligence, Inc. 8-bit Microcontroller Market – Growth, Trends, and Forecast (2020–2025). Available for download at `http://www.mordorintelligence.com/industry-reports/8-bit-microcontroller-market-industry`, 2020.
7. Radiant Insights, Inc. Microcontroller Market Size, Share, Analysis Report 2020. Available for download at `http://www.radiantinsights.com/research/microcontroller-market/`, 2015.