The Simon and Speck Block Ciphers on AVR 8-Bit Microcontrollers

Ray Beaulieu, Douglas Shors, Jason Smith(✉), Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers

National Security Agency, 9800 Savage Road, Fort Meade, MD 20755, USA
{rabeaul,djshors,jksmit3,sgtreat,beweeks,lrwinge}@tycho.ncsc.mil

Abstract. The last several years have witnessed a surge of activity in lightweight cryptographic design. Many lightweight block ciphers have been proposed, targeted mostly at hardware applications. Typically software performance has not been a priority, and consequently software performance for many of these algorithms is unexceptional. Simon and Speck are lightweight block cipher families developed by the U.S. National Security Agency for high performance in constrained hardware *and* software environments. In this paper, we discuss software performance and demonstrate how to achieve high performance implementations of Simon and Speck on the AVR family of 8-bit microcontrollers. Both ciphers compare favorably to other lightweight block ciphers on this platform. Indeed, Speck seems to have better overall performance than any existing block cipher — lightweight or not.

Keywords: Simon · Speck · Block cipher · Lightweight · Cryptography · Microcontroller · Wireless sensor · AVR

1 Introduction

The field of lightweight cryptography is evolving rapidly in order to meet future needs, where interconnected, highly constrained hardware and software devices are expected to proliferate.

Over the last several years the international cryptographic community has made significant strides in the development of lightweight block ciphers. There are now more than a dozen to choose from, including Present [6], Clefia [20], Twine [21], Hight [14], Klein [12], LBlock [25], Piccolo [19], Prince [7], LED [13], Katan [9], TEA [23], and ITUbee [16]. Typically, a design will offer improved performance on some platform (e.g., ASIC, FPGA, microcontroller, microprocessor) relative to its predecessors. Unfortunately, most have good performance in hardware or software, but not both. This is likely to cause problems when communication is required across a network consisting of many disparate (hardware- and software-based) devices, and the highly-connected nature of

T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 3–20, 2015.
DOI: 10.1007/978-3-319-16363-5_1

developing technologies will likely lead to many applications of this type. Ideally, lightweight cryptography should be *light* on a variety of hardware and software platforms. Cryptography of this sort is relatively difficult to create and such general-purpose lightweight designs are rare.

Recently, the U.S. National Security Agency (NSA) introduced two general-purpose lightweight block cipher families, Simon and Speck [2], each offering high performance in *both* hardware and software. In hardware, Simon and Speck have among the smallest reported implementations of existing block ciphers with a flexible key.[1] Unlike most hardware-oriented lightweight block ciphers, Simon and Speck also have excellent software performance.

In this paper, we focus on the software performance of Simon and Speck. Specifically, we show how to create high performance implementations of Simon and Speck on 8-bit Atmel AVR microcontrollers. This paper should be useful for developers trying to implement Simon or Speck for their own applications and interesting for cryptographic designers wishing to see how certain design decisions affect performance.

2 The Simon and Speck Block Ciphers

Simon and Speck are block cipher families, each comprising ten block ciphers with differing block and key sizes to closely fit application requirements. Table 1 shows the available block and key sizes for Simon and Speck.

Table 1. Simon and Speck parameters.

Block size	Key sizes
32	64
48	72, 96
64	96, 128
96	96, 144
128	128, 192, 256

The Simon (resp. Speck) block cipher with a $2n$-bit block and wn-bit key is denoted by Simon $2n/wn$ (resp. Speck $2n/wn$). Together, the round functions of the two algorithms make use of the following operations on n-bit words:

- bitwise `XOR`, $\oplus$,
- bitwise `AND`, $\&$,
- left circular shift, S^j, by j bits,
- right circular shift, S^{-j}, by j bits, and
- modular addition, $+$.

[1] Simon 64/96 and Speck 64/96, for example, have implementations requiring just 809 and 860 gate equivalents, respectively. Some block ciphers, like KTANTAN [9], have a fixed key and so do not require flip-flops to store it. Such algorithms can have smaller hardware implementations than Simon or Speck, but not allowing keys to change contracts the application space, and can lead to security issues [22].

For $k \in \mathrm{GF}(2)^n$, the key-dependent Simon $2n$ round function is the two-stage Feistel map $R_k \colon \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n$ defined by

$$R_k(x, y) = (y \oplus f(x) \oplus k,\ x),$$

where $f(x) = (Sx \,\&\, S^8 x) \oplus S^2 x$ and k is the round key. The key-dependent Speck $2n$ round function is the map $R_k \colon \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n$ defined by

$$R_k(x, y) = ((S^{-\alpha} x + y) \oplus k,\ S^{\beta} y \oplus (S^{-\alpha} x + y) \oplus k),$$

with rotation amounts $\alpha = 7$ and $\beta = 2$ if $n = 16$ (block size = 32) and $\alpha = 8$ and $\beta = 3$, otherwise. A description of the key schedules, not necessary for this paper, can be found in the Simon and Speck specification paper [2].

3 AVR Implementations of Simon and Speck

We have implemented Simon and Speck on the Atmel ATmega128, a low-power 8-bit microcontroller with 128K bytes of programmable flash memory, 4K bytes of SRAM, and thirty-two 8-bit general purpose registers. To achieve optimal performance, we have coded Simon and Speck in assembly. The simplicity of the algorithms makes this easy to do, with the help of the Atmel AVR 8-bit instruction set manual [1].

Our assembly code was written and compiled using Atmel Studio 6.0. Cycle counts, code sizes and RAM usage were also determined using this tool. Our complete set of performance results is provided in Appendix A.

For Simon and Speck, three implementations were developed, none of which included the decryption algorithm.[2]

(1) **RAM-minimizing implementations.** These implementations avoid the use of RAM to store round keys by including the pre-expanded round keys in the flash program memory. No key schedule is included for updating this expanded key, making these implementations suitable for applications where the key is static.

(2) **High-throughput/low-energy implementations.** These implementations include the key schedule and unroll enough copies of the round function in the encryption routine to achieve a throughput within about 3% of a fully-unrolled implementation. The key, stored in flash, is used to generate the round keys which are subsequently stored in RAM.

[2] This is because one is likely to use encrypt-only modes in lightweight cryptography. But the techniques discussed here should serve as a starting point for other kinds of implementations, useful for a broad range of applications. Regarding decryption functionality, we note that the Simon and Speck encryption and decryption algorithms consume similar resources and are easy to implement. Simon, in particular, has a decryption algorithm that is closely related to the encryption algorithm, and so little additional code is necessary to enable decryption.

(3) **Flash-minimizing implementations.** The key schedule is included here. Space limitations mean we can only provide an incomplete description of these implementations. However, it should be noted that the previous two types of implementations already have very modest code sizes.

In almost all cases, the registers hold the intermediate ciphertext, the currently used round key, and any data needed to carry out the computation of a round. For the algorithms with the 128-bit block and 192-bit or 256-bit key, there may not be enough registers to hold all of this information, and so it may be necessary to store one or two 64-bit words of round key in RAM. Additional modifications were necessary for the 128-bit SIMON block ciphers.

We describe our various assembly implementations of SIMON 64/128 and SPECK 64/128 on an AVR 8-bit microcontroller using pseudo assembly code. When it is not obvious, we show how to translate the pseudocode into actual assembly instructions. Although our implementations were aimed at the ATmega128 microcontroller, the same techniques are applicable to other AVR microcontrollers, e.g., the ATtiny line (except when noted).

4 SIMON AVR Implementations

We describe various implementations of SIMON 64/128 on the ATmega128 microcontroller. Implementations of the other algorithms in the family are similar.

The ATmega128 has thirty-two 8-bit registers. For the purpose of exposition, we regard a sequence of four such registers as a 32-bit register, denoted by a capital letter such as X, Y, K, etc. For instance, $X = (X_3, X_2, X_1, X_0)$ identifies the 32-bit register X as a sequence of four 8-bit registers. We denote the contents of these 32-bit registers by lower case letters x, y, k, etc.

At the start of the encryption process, we assume a 64-bit plaintext pair (x, y) resides in RAM, and is immediately loaded into an (X, Y) register pair for processing. After 44 encryption rounds, the resulting ciphertext is stored in RAM. The encryption cost is the number of cycles needed to transform the plaintext into the ciphertext, including any loading of plaintext/ciphertext into or out of RAM.[3] Our performance numbers do not count the cost of RAM used to hold the plaintext, ciphertext, or key but do include RAM used for temporary storage (e.g., stack memory) and RAM to hold the round keys.

Except for the small code size implementations of SIMON, developers should avoid the use of any loops or branching within a round, since this can significantly degrade overall performance and does not greatly reduce code size.

4.1 A Minimal RAM Implementation of SIMON

Here we assume the round keys have been pre-expanded (by an external device) and stored in flash. When a round key is required, it is loaded from flash directly into a register. Loading a byte from flash consumes three cycles.

[3] The SIMON and SPECK specification paper [2] did not count these cycles required for loading, although it seems proper to do so. The current performance numbers include these costs.

We begin by describing the `Rotate` operation, which performs a left circular shift by one. Let $X = (X_3, X_2, X_1, X_0)$ and let $Z_0 = 0$. The 8-bit register Z_0 should be cleared and reserved at the start of encryption. The 1-bit rotation of the 32-bit register X is carried out using AVR's logical shift left (`LSL`), rotate left through carry (`ROL`), and add with carry (`ADC`) instructions, as follows.

$$
\begin{array}{lll}
(1) & X_0 \leftarrow \texttt{LSL}(X_0) & \text{(logical shift left)} \\
(2) & X_1 \leftarrow \texttt{ROL}(X_1) & \text{(rotate left through carry)} \\
(3) & X_2 \leftarrow \texttt{ROL}(X_2) & \text{(rotate left through carry)} \\
(4) & X_3 \leftarrow \texttt{ROL}(X_3) & \text{(rotate left through carry)} \\
(5) & X_0 \leftarrow \texttt{ADC}(X_0, Z_0) & \text{(add with carry)}
\end{array}
$$

In general, this *standard* approach to performing a rotation requires $n + 1$ cycles on an n byte word. Rotations by more than one bit can be achieved by repeated one-bit rotations, though this is not always the most efficient approach.

Note that the rotation by 8 is free, because it's just a byte permutation.[4] The rotations by 1 and 2 are also inexpensive. Our pseudo instruction `Move` is shorthand for two AVR `MOVW` instructions, each of which copies two 8-bit words from one register to another in a single cycle. In order to use the `MOVW` instruction, the 8-bit registers must be properly aligned [1]. Our other mnemonics are also shorthand for readily apparent AVR instructions. Table 2 describes how to implement a round of Simon.

Table 2. Low-RAM software implementation of the Simon round function.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`XOR`	$K \leftarrow K \oplus Y$	$K = y \oplus k$	4
`Move`	$Y \leftarrow X$	$Y = x$	2
`Rotate`	$X \leftarrow S^1(X)$	$X = S^1(x)$	5
`Move`	$T \leftarrow X$	$T = S^1(x)$	2
`And`	$T \leftarrow T \,\&\, S^8(Y)$	$T = S^1(x) \,\&\, S^8(x)$	4
`Rotate`	$X \leftarrow S^1(X)$	$X = S^2(x)$	5
`XOR`	$X \leftarrow X \oplus T$	$X = S^2(x) \oplus S^1(x) \,\&\, S^8(x)$	4
`XOR`	$X \leftarrow X \oplus K$	$X = y \oplus S^2(x) \oplus S^1(x) \,\&\, S^8(x) \oplus k$	4

The basic code for a round executes in 42 cycles. For the minimal RAM implementation, this code is put in a loop which has a 3 cycle per round overhead. Since Simon 64/128 requires 44 rounds, an encryption will take about $44 \cdot (42 + 3) = 1980$ cycles. There are ten cycles needed for setup, a subroutine call, and a return plus an additional 32 cycles to load the plaintext into registers

[4] This rotation is also easily implemented (but not for free) on some common 16-bit microcontrollers, like the MSP430, and using x86 SSE instructions (where no rotate is available but a byte permutation operation is).

(2 cycles/byte) and load the resulting ciphertext back into RAM (2 cycles/byte). Altogether, SIMON 64/128 has an encryption cost of 253 cycles/byte.

In terms of code size, each instruction, with the exception of the `Load` instruction, takes twice as many bytes to store in flash as the number of cycles it takes to execute. The four byte `Load` instruction, in Table 2, consumes 8 bytes of flash. The expanded key is stored in $44 \cdot 4 = 176$ bytes of flash. So the total amount of flash used is around $68 + 44 \cdot 4 = 244$ bytes. More bytes are required for the counter, loading plaintext, storing ciphertext and other miscellaneous overhead so the actual value is 290 bytes. No RAM was used for this implementation.

4.2 A High-Throughput/Low-Energy Implementation of SIMON

High-throughput implementations are useful when encrypting multiple blocks of data. The round keys for such an implementation can initially be stored in flash, as with our low-RAM implementations, or they can be generated using the key schedule. In either case, the encryption process begins by placing all of the round keys into RAM, where they are held until all of the blocks in a stream of data have been encrypted. For our implementations, we used the key schedule to generate the round keys, but for our timings we have not included the time to generate and store these since this *setup* time is assumed to be small compared to the time required to encrypt the data stream.

For SIMON 64/128, we unrolled four rounds of the code and iterated this 11 times to carry out the 44 round encryption. Because the key is now loaded from RAM, the `Load` requires only 8 cycles instead of the 12 which were needed when loading from flash. Due to the larger code size, a four cycle overhead for each update of the loop counter (as opposed to three cycles) is required, together with an additional ten cycles as described earlier and 32 more cycles for loading plaintext in registers and storing the ciphertext in RAM. The amount of unrolling was calibrated to achieve throughput to within 3 % of the fastest (fully unrolled) implementation, avoiding large increases in code size with minor improvements in throughput. The cost of this implementation is 221 cycles/byte.

The code size for the encryption algorithm is about $68 \cdot 4 = 272$ bytes. Together with the key schedule and other overhead, this implementation of SIMON 64/128 uses 436 bytes of flash. Because all of the round keys are placed in RAM, $44 \cdot 4 = 176$ bytes of RAM are also required.

4.3 A Minimal Flash Implementation of SIMON

To reduce the already small size of SIMON, we implemented several frequently used operations as subroutines. These included the `XOR`, the 1-bit `Rotate` and the `Move` instructions. Doing this, we saved a dozen bytes for SIMON 64, and more than 50 bytes for SIMON 128. For SIMON 32, these techniques ended up not saving any space and degraded throughput, so we did not use them.

SIMON 64/128, in particular, can be implemented using 240 bytes of flash and with $4 \cdot 44 = 176$ bytes of RAM to store the round keys. We note that for an additional 28 bytes of flash, decrytpion capability can be added: To decrypt,

one uses the round keys in reverse order, loads the swapped ciphertext words into the encryption round function, and reads out the plaintext words with a similar swap. The swapped loading and output (together with the regular loading and output) can be done with 8 additional bytes of code each, and the round keys can be generated and stored in normal or reverse order with 12 additional bytes of code, for a total of $8 + 8 + 12 = 28$ bytes.

5 Speck AVR Implementations

In this section, we describe the same types of implementations that we previously developed for the Simon 64/128 block cipher. We stress that most of the implementation techniques described here apply immediately to the other members of the Speck family.

Using the same notation as in Simon, we assume a 64-bit plaintext pair (x, y) resides in RAM. This is immediately loaded into the 64-bit register pair (X, Y). After 27 encryption rounds, the resulting ciphertext is loaded into RAM. The encryption cost is the number of cycles required to load the plaintext into registers, transform the plaintext into the ciphertext, and store the ciphertext in RAM.

For all of our implementations, we avoid the use of any loops or branching within a round. This has the effect of making our code slightly larger but significantly improves the overall performance.

For Simon, our main tool for trading off code size for throughput was to partially unroll loops. It turns out that for Speck, there is more opportunity to tune the implementation (for example using multiply instructions to accomplish the rotation by 3 or by allowing a round to end with plaintext and key words in the *wrong* places) to make it smaller or faster. Consequently, we will spend a little more time describing Speck implementations.

5.1 A Low-RAM Speck Implementation

Here we assume the round keys have been pre-expanded and stored in flash. When a round key k is required, it is loaded from flash directly into a register with a cost of 12 cycles. Table 3 describes how to implement the Speck round.

Table 3. Low-RAM software implementation of the Speck round function.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow K \oplus S^{-8}(X)$	$K = (S^{-8}(x) + y) \oplus k$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y)$	15
`XOR`	$Y \leftarrow Y \oplus K$	$Y = S^3(y) \oplus (S^{-8}(x) + y) \oplus k$	4
`Move`	$X \leftarrow K$	$X = (S^{-8}(x) + y) \oplus k$	2

Note that the rotation by 8 is free, as we noted in the SIMON discussion. The `Rotate` and `Move` instructions were described earlier for SIMON. To be clear, we describe the `Add` operation in greater detail. If $X = (X_3, X_2, X_1, X_0)$ and $Y = (Y_3, Y_2, Y_1, Y_0)$, then addition, `Add`, is carried out using the following AVR instructions in the given order.

$$\begin{array}{lll} (1) & X_1 \leftarrow X_1 + Y_0 & \text{(add without carry, ADD)} \\ (2) & X_2 \leftarrow X_2 + Y_1 & \text{(add with carry, ADC)} \\ (3) & X_3 \leftarrow X_3 + Y_2 & \text{(add with carry, ADC)} \\ (4) & X_0 \leftarrow X_0 + Y_3 & \text{(add with carry, ADC)} \end{array}$$

Our basic code for a round executes in 41 cycles, and this code is iterated 27 times in a loop to produce the ciphertext. The loop has an overhead of 3 cycles per round, and since SPECK 64/128 requires 27 rounds, an encryption takes about $27 \cdot (41 + 3) = 1188$ cycles. An additional 10 more cycles for overhead (3 cycles for setup, 3 for a subroutine call, and 4 for a return) plus 32 cycles for loading plaintext from RAM into registers and storing ciphertext back into RAM, brings the total number of cycles for an encryption to 1230, which translates to $1230/8 \approx 154$ cycles/byte.

As we noted in the SIMON discussion, each instruction, with the exception of the `Load` instruction, takes twice as many bytes to store in flash as the number of cycles it takes to execute. The `Load` instruction requires 8 bytes of flash.

The SPECK round keys consume $27 \cdot 4 = 108$ bytes of flash. The total amount of flash required for the round and expanded key is $66 + 108 = 174$ bytes. Additional bytes, required for the counter, loading and storing plaintext and ciphertext and other overhead, brings the total to 218 bytes. No RAM was required for this implementation.

5.2 A Faster Low-RAM SPECK Implementation

If a higher-throughput/lower-energy implementation is required, we can easily modify the implementation that we just described to obtain one which encrypts at a rate of 142 cycles/byte using around 342 bytes of flash and no RAM.

This is done by iterating two rounds multiple times. Two rounds can be implemented without the use of the `Move` that appeared in Table 3, thereby saving two cycles per round. The two rounds are iterated in a loop 13 times (for a total of 26 rounds) and a final single round of code as described in Table 3 is used for the 27th round. For members of the family requiring an even number of rounds (like SPECK 64/96) this extra code for the final round is not required. For SPECK 64/128, the number of cycles for a complete encryption is around $2 \cdot 39 \cdot 13 + 41 + 13 \cdot 3 + 10 + 32 = 1136$, or about 142 cycles/byte. About 342 bytes of flash are required.

The pseudocode for the two rounds is shown in Table 4. There, x_1 and y_1 are the values of the input to the second round, stored in the K and Y registers and l is a new round key which is loaded into the X register. The output to the second round is again stored in the *proper* registers, i.e., the X and Y registers.

Table 4. A faster, low-RAM software implementation of two rounds of Speck.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow K \oplus S^{-8}(X)$	$K = (S^{-8}(x) + y) \oplus k = x_1$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y)$	15
`XOR`	$Y \leftarrow Y \oplus K$	$Y = S^3(y) \oplus (S^{-8}(x) + y) \oplus k = y_1$	4
`Load`	$X \leftarrow l$	$X = l$	12
`Add`	$K \leftarrow S^8(S^{-8}(K) + Y)$	$K = S^8(S^{-8}(x_1) + y_1)$	4
`XOR`	$X \leftarrow X \oplus S^{-8}(K)$	$X = (S^{-8}(x_1) + y_1) \oplus l$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y_1)$	15
`XOR`	$Y \leftarrow Y \oplus X$	$Y = S^3(y_1) \oplus (S^{-8}(x_1) + y_1) \oplus l$	4

5.3 A High-Throughput/Low-Energy Speck Implementation

As in the corresponding Simon implementations, the Speck encryption process begins by placing all of the round keys into RAM and holding them there until the data stream has been encrypted. Although we used the key schedule to generate the round keys, the round keys could also be pre-computed and stored in flash before loading them into RAM. For our timings, we have not included the time to generate the round keys and store them in RAM.

The easiest way to obtain a fast encryption algorithm is just to modify the fast, low-RAM implementation described previously using the code found in Table 4, but now loading the round keys from RAM instead of from flash. The cost of loading four bytes from RAM is 8 cycles, as opposed to 12 if we load from flash. So the total cost to encrypt a block of data is $2{\cdot}35{\cdot}13{+}37{+}13{\cdot}4{+}10{+}32 = 1041$ cycles, or about 130 cycles/byte. Not including the key schedule, the code will occupy about 232 bytes. Taking into account the key schedule, the code size is about 352 bytes and the implementation uses 108 bytes of RAM.

We could speed this up even more, at the expense of using more flash, by unrolling four rounds of code instead of just two. If we unroll all of the rounds we get a heavy implementation, in terms of flash, that encrypts a block in just 123 cycles/byte. If this sort of implementation is appealing but the flash usage is not, there is another way to get the same throughput using significantly less flash. However, the technique only works on those AVR microcontrollers, such as the ATmega128, which include the AVR unsigned multiplication instruction `MUL`. The ATtiny line does not have the `MUL` instruction.

We now describe how to do the 3-bit rotation operation `Rotate` in 14 cycles using 20 bytes of flash with the AVR `MUL` instruction. This should be compared to the in-place rotation used earlier, which required 15 cycles and 30 bytes.

The `MUL` instruction operates on two 8-bit registers containing unsigned numbers and produces the 16-bit unsigned product in 2 cycles. The result is always placed in the AVR register pair (R_1, R_0), the low bits in R_0 and the high bits

in R_1. Letting $Y = (Y_3, Y_2, Y_1, Y_0)$ and $X = (X_3, X_2, X_1, X_0)$, the `Rotate` operation, $X \leftarrow S^3(Y)$, is implemented as follows:

$$
\begin{array}{lrcl}
(1) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_0, 8) \\
(2) & (X_1, X_0) & \leftarrow & (R_1, R_0) \\
(3) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_2, 8) \\
(4) & (X_3, X_2) & \leftarrow & (R_1, R_0) \\
(5) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_1, 8) \\
 & & & \\
(6) & X_1 & \leftarrow & X_1 \oplus R_0 \\
(7) & X_2 & \leftarrow & X_2 \oplus R_1 \\
(8) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_3, 8) \\
(9) & X_3 & \leftarrow & X_3 \oplus R_0 \\
(10) & X_0 & \leftarrow & X_0 \oplus R_1
\end{array}
$$

Here, 8 represents an 8-bit register with the value 8 stored in it. This register should be initialized and reserved at the start of the encryption process. It should be noted that this rotate algorithm is not in-place. Additionally, for a w-byte word, the performance can be worse than for the standard approach: For $w = 2j$, the running time is $7j$ cycles for this technique versus $3 \cdot (2j+1) = 6j+3$ cycles for the standard approach. Thus, for the highest speed implementations, the technique is advantageous for Speck 48 and Speck 64. For Speck 96 it doesn't increase throughput, but it can still significantly decrease the flash usage, and so we adopt it. For Speck 128, the method will actually decrease throughput. The code for a round of Speck is provided in Table 5.

Table 5. A high-throughput/low-power round implementation of Speck

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	8
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow S^{-8}(X) \oplus K$	$K = (S^{-8}(x) + y) \oplus k$	4
`Rotate`	$X \leftarrow S^3(Y)$	$X = S^3(y)$	14
`XOR`	$X \leftarrow X \oplus K$	$X = (S^{-8}(x) + y) \oplus k \oplus S^3(y)$	4

The output of the first round is stored in the (K, X) register pair. In order to get the output in the proper, X and Y registers, we need to move X into Y and K into X. However, if we don't want to incur additional cycles by doing this, then we can proceed in a manner similar to the fast, low-RAM implementation by iterating three consecutive rounds, all identical up to a relabeling of registers. The final output after the third round will be back in the (X, Y) register pair. The code for doing this is shown in Table 6.

If we iterate three rounds in a loop nine times as just described, Speck 64/128 performs a full encryption in around $34 \cdot 27 + 4 \cdot 9 + 10 + 32 = 996$ cycles, or about 125 cycles/byte and, including the key schedule, uses 316 bytes of flash.

Table 6. A high-throughput/low-energy three round implementation of Speck

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	8
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow S^{-8}(X) \oplus K$	$K = (S^{-8}(x) + y) \oplus k = x_1$	4
`Rotate`	$X \leftarrow S^3(Y)$	$X = S^3(y)$	14
`XOR`	$X \leftarrow X \oplus K$	$X = S^3(y) \oplus (S^{-8}(x) + y) \oplus k = y_1$	4
`Load`	$Y \leftarrow l$	$Y = l$	8
`Add`	$K \leftarrow S^8(S^{-8}(K) + X)$	$K = S^8(S^{-8}(x_1) + y_1)$	4
`XOR`	$Y \leftarrow S^{-8}(K) \oplus Y$	$Y = (S^{-8}(x_1) + y_1) \oplus l = x_2$	4
`Rotate`	$K \leftarrow S^3(X)$	$K = S^3(y_1)$	14
`XOR`	$K \leftarrow K \oplus Y$	$K = S^3(y_1) \oplus (S^{-8}(x_1) + y_1) \oplus l = y_2$	4
`Load`	$X \leftarrow m$	$X = m$	8
`Add`	$Y \leftarrow S^8(S^{-8}(Y) + K)$	$Y = S^8(S^{-8}(x_2) + y_2)$	4
`XOR`	$X \leftarrow S^{-8}(Y) \oplus X$	$X = (S^{-8}(x_2) + y_2) \oplus m = x_3$	4
`Rotate`	$Y \leftarrow S^3(K)$	$Y = S^3(y_2)$	14
`XOR`	$Y \leftarrow Y \oplus X$	$Y = S^3(y_2) \oplus (S^{-8}(x_2) + y_2) \oplus m = y_3$	4

The high-throughput data for Speck 64/128, found in Table 7 of Appendix A, was obtained by unrolling 6 rounds instead of 3 in order to get within 3% of the fastest, fully unrolled, implementation. That gave us a 122 cycles/byte implementation but required about twice the flash.

5.4 A Small Flash Speck Implementation

Speck can be implemented to have small code size. We have not attempted to minimize the code size without regard to the throughput, so smaller implementations are possible. The primary savings in space was achieved by implementing the Speck round function as a subroutine. In this way, both the key schedule and encryption function could use it without having to duplicate code.

6 Cipher Comparisons

Although it is not the primary purpose of this paper, we shall compare the performance of Simon and Speck with several other block ciphers, and with a few stream ciphers. Comparisons along these lines can already be found in the Simon and Speck specification paper [2]. We have not endeavored to implement all of the discussed ciphers from scratch since this was outside the scope of our paper. Instead, when possible, we (or others) modified the best implementations available from existing sources so that they fit within our framework. Our comparison data and a further discussion of our methodology can be found in Appendix B.

For brevity, we only included algorithms with an encryption cost of 1000 cycles/byte or less. For this reason, or because the data was unavailable, well-known lightweight block ciphers such as PRESENT, CLEFIA, KATAN, etc., are not listed. We use a performance efficiency measure, *rank*, that is similar to a commonly used metric (see [16,18,21]), proportional to throughput divided by memory usage.

Among all block ciphers, SPECK ranks in the top spot for every block and key size which it supports. Except for the 128-bit block size, SIMON ranks second for all block and key sizes. Among the 64-bit block ciphers, HIGHT has respectable performance on this platform, ranking higher than AES. Although TWINE ranks lower than AES, its performance is reasonable and, additionally, it has very good hardware performance, making it one of the better lightweight designs. Some software-based designs, like ITUBEE, IDEA and TEA have poorer performance than AES and are not compensated by particularly lightweight hardware implementations. KLEIN, a hardware-based design, is slow and has the least competitive overall performance among the 64-bit block ciphers in our comparison.

Not surprisingly, AES-128 has very good performance on this platform, although for the same block and key size, SPECK has about twice the performance. For the same key size but with a 64-bit block size, SIMON and SPECK achieve two and four times better overall performance, respectively, than AES. A few of the block ciphers in our comparison could not outperform AES, even though they had smaller block and key sizes.

If an application requires high speed, and memory usage is not a priority, AES has the fastest implementation (using 1912 bytes of flash, 432 bytes RAM) among all block ciphers with a 128-bit block and key that we are aware of, with a cost of just 125 cycles/byte [8]. The closest AES competitor is SPECK 128/128, with a cost of 138 cycles/byte for a fully unrolled implementation. Since speed is correlated with energy consumption, AES-128 may be a better choice in energy critical applications than SPECK 128/128 on this platform.[5] However, if a 128-bit block is not required, as we might expect for many applications on an 8-bit microcontroller, then a more energy efficient solution (using 628 bytes of flash, 108 bytes RAM) is SPECK 64/128 with the same key size as AES-128 and an encryption cost of just 122 cycles/byte, or SPECK 64/96 with a cost of 118 cycles/byte.

We have also compared three of the four Profile I (software) eSTREAM competition finalists, Salsa 20/12 [4], Sosemanuk [3], and HC-128 [24] since there is a general perception that well-designed stream ciphers must have better performance than block ciphers.[6] Interestingly, all of the stream ciphers are less efficient than the majority of the block ciphers listed. For high-speed/low-energy applications, if memory is not a concern, only Sosemanuk can beat the fastest implementations of AES and SPECK.

[5] We do not know, for a fact, that the high-speed AES implementations, which require frequent calls to RAM, are more energy efficient than the high-speed SPECK implementations which use mostly register-to-register operations.

[6] No data for the other finalist, Rabbit [5], was available.

A Simon and Speck AVR Performance

In this section we present the results of our three AVR implementations of Simon (Table 8) and Speck (Table 7). The headings of each column indicate the block size/key size in bits. The first three rows of data correspond to the low flash implementation, the next three rows to the low RAM implementation and the last three rows are for the low cost (i.e., high-speed/low-energy) implementation. RAM and flash are measured in bytes and cost is measured in cycles/byte. For none of the algorithms is any sort of functionality for the decryption operator included, although for Simon decryption is essentially the same as encryption, and Speck decryption uses about the same amount of resources as Speck encryption.

Table 7. Speck AVR implementation data.

	32/64	48/72	48/96	64/96	64/128	96/96	96/144	128/128	128/192	128/256
flash	**118**	**150**	**156**	**184**	**192**	**218**	**254**	**278**	**330**	**348**
RAM	44	69	72	108	112	174	180	264	272	280
cost	171	149	155	158	164	154	159	169	174	179
flash	108	154	158	214	218	324	330	460	468	476
RAM	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
cost	144	134	140	148	154	152	157	171	176	181
flash	440	556	586	588	628	502	624	452	632	522
RAM	44	66	69	104	108	168	174	256	272	288
cost	**114**	**104**	**108**	**118**	**122**	**127**	**131**	**143**	**147**	**151**

Table 8. Simon AVR implementation data.

	32/64	48/72	48/96	64/96	64/128	96/96	96/144	128/128	128/192	128/256
flash	**152**	**190**	**202**	**230**	**240**	**304**	**304**	**392**	**392**	**404**
RAM	64	108	108	168	176	312	324	544	552	576
cost	193	206	206	222	232	262	272	346	351	366
flash	130	190	190	282	290	474	486	760	768	792
RAM	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
cost	207	222	222	242	253	287	297	379	384	401
flash	400	454	466	562	436	592	492	510	646	522
RAM	64	108	108	168	176	312	324	544	552	576
cost	**172**	**191**	**191**	**209**	**221**	**253**	**264**	**337**	**339**	**357**

B Comparison Data and Methodology

Fair comparisons adhere to a common framework. The framework provides three important pieces of information. First, it provides a high-level, device-independent description of what the cipher is expected to do. At a lower level, device-dependent implementation details are provided. Finally, a performance metric is chosen to make the comparisons meaningful. Needless to say, any ranking depends on the framework used, especially the performance metric. The following application types are especially relevant to lightweight cryptography.

- **Fixed Key/Small Data.** Fixed key applications assume the key will never (or rarely) be changed. In hardware, area requirements can be reduced (depending on the design and the implementation) because state for a key schedule may not be required. In software, the key, or better yet the expanded key, can be stored in long-term memory and the key schedule can be relinquished. For small data size comparisons, we may assume we are encrypting just a single block and so incur the full cost of any setup. This application type may be appropriate for simple authentication applications.
- **Fixed Key/Large Data.** This is the same as the previous type except that the data stream is assumed to be large. For comparison purposes, we may assume the amount of encrypted data approaches infinity, amortizing away the setup costs. This may be appropriate for various sensor applications.
- **Flexible Key/Small Data.** Here, we assume the key is changed often. In hardware or software, this necessitates the inclusion of the key schedule. For comparison purposes, we may assume that we are encrypting a single block of data with a never-before-seen key. This type of application may be appropriate when the block cipher is contained in a general purpose crypto module and the key enters the device from outside of the module.
- **Flexible Key/Large Data.** Similar to the preceeding except the data stream is large. For comparison purposes, we may again assume the data stream is (effectively) infinite, amortizing away all setup costs.

It is generally recognized in lightweight cryptography that use of the decryption operator should be avoided, if possible, in order to conserve resources. If resources are not a big concern, one should use AES. For software applications on a microcontroller, implementations should be assembly coded in order to reduce compiler vagaries and to provide for maximal performance (i.e., to reduce code size and memory usage and to increase throughput).

For our comparisons, shown in Table 9, we (mostly) used the Fixed Key/ Small Data framework. For our implementations, expanded key is stored in flash and only encryption functionality is provided. Key schedules are also absent. The encryption procedure begins by loading the plaintext from RAM into registers. The plaintext is then transformed into the ciphertext using the encryption operator. The resulting ciphertext is then loaded in RAM. This completes the encryption process. RAM for holding the plaintext and ciphertext is not costed

Table 9. Comparisons of Simon and Speck with some other block and stream ciphers on the ATmega128, 8-bit microcontroller in the Fixed Key/Small Data framework. Values in square brackets, [ ], are our best estimates based on the existing literature. The higher the rank, the better the *overall* performance. Since our goal was to optimize the rank, these implementations are not necessarily the fastest possible.

Size	Name	Flash (bytes)	RAM (bytes)	Cost (bytes)	Rank
Comparisons with Block Ciphers					
48/96	Speck	158	0	140	45.2
	Simon	190	0	222	23.7
64/80	Twine	1208	23	326	2.4
	Klein	766	18	762	1.6
80/80	ITUbee	586	0	294	5.8
64/96	Speck	214	0	148	31.6
	Simon	282	0	242	14.7
	Klein	[766.]	[18.]	[955.]	[1.3]
64/128	Speck	218	0	154	29.8
	Simon	290	0	253	13.6
	Hight	336	0	311	9.6
	IDEA	[397.]	0	338	[7.5]
	Twine	1208	23	326	2.4
	TEA	350	12	638	4.2
96/96	Speck	324	0	152	20.3
	Simon	474	0	287	7.4
	SEA	[1066.]	0	805	[1.2]
128/128	Speck	460	0	171	12.7
	AES	970	18	146	6.8
	Simon	760	0	379	3.5
Comparisons with Stream Ciphers					
128/256	Speck	476	0	181	11.6
	AES	1034	18	204	4.7
	Simon	792	0	401	3.1
512/256	Salsa 20/12	1092	275	177	3.4
384/128	Sosemanuk	11140	712	118	0.7
32768/128	HC-128	23100	4556	169	0.2

but RAM used for temporary storage (e.g., on the stack) is. Our low-RAM implementations of Simon and Speck were appropriate for this comparison. For the other ciphers, implementations fitting the framework were based on the best existing code (or performance data) we could find which maximized the overall performance metric, *rank*, which is defined to be

$$(10^6/\text{cost})/(\text{flash} + 2 \cdot \text{RAM});$$

higher values of rank correspond to better performance.[7] In some cases, this just amounted to stripping out the code for the decryption and key schedule algorithms in existing implementations. In other cases, we wrote the code ourselves.

Note that our performance metric is an *overall* measure of performance and takes into account flash, RAM and throughput. However, depending on priorities, this metric may be irrelevant. If the main concern is energy efficiency, then a more appropriate metric is just throughput and a fair comparison will require implementations optimized for this purpose, resulting in altered rankings. We have already alluded to this in our discussion in Sect. 6.

Referring to Table 9, all implementations, except for HC-128, were assembly coded. *Size* is block size/key size for block ciphers and state size/key size for stream ciphers. The *cost* is the number of cycles per byte to transform a block of plaintext into a block of ciphertext.

The ITUBEE data is taken directly from its specification paper [16]. Data for SEA, IDEA and KLEIN are taken from [10,11,18], respectively. In some cases, code size estimates had to be made to fit our framework. The TEA and HIGHT implementations are our own. For AES, our numbers were kindly provided by Dag Arne Osvik, one of the authors of [8], who made suitable code modifications to fit our framework. The TWINE data, fitting our framework, was provided by two of the TWINE designers, Kazuhiko Minematsu and Tomoyasu Suzaki. Our numbers for Salsa 20/12 were obtained by scaling down the cost of the Salsa 20/20 implementation provided in [15]. Data for Sosemanuk and HC-128 was obtained from [17]. We did not include the considerable setup time for HC-128 and the moderate setup time for Sosemanuk, and of course this setup time *should* be considered in the Fixed Key/Small Data framework.[8]

References

1. Atmel Corporation. 8-bit AVR Instruction Set, Rev. 0856I-AVR-07/10. http://www.atmel.com/images/doc0856.pdf
2. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L. The Simon and Speck Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013). http://eprint.iacr.org/
3. Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., Sibert, H. SOSEMANUK, a fast software-oriented stream cipher. In: CoRR, abs/0810.1858 (2008)
4. Bernstein, D.: The Salsa20 family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008)

[7] The *rank* is similar to the metric found in [21] except we have imposed a penalty for using too much RAM — hence the factor of 2. Without the factor of 2, flash and RAM have the same cost, which seems unjustifiable.

[8] The HC-128 stream cipher implementation does not actually fit on the ATmega128 due to its excessive use of RAM. The C implementation of HC-128 described in [17] has a setup cost of over 2,000,000 cycles.

5. Boesgaard, M., Vesterager, M., Pedersen, T., Christiansen, J., Scavenius, O.: Rabbit: a new high-performance stream cipher. In: Johansson, T. (ed.) Fast Software Encryption, vol. 2887, pp. 307–329. Springer, Heidelberg (2003)
6. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
7. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knežević, M., Knudsen, L.R., Leander, G., Nikov, V., Parr, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE – a low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012)
8. Bos, J., Osvik, D., Stefan, D.: Fast Implementations of AES on Various Platforms. Cryptology ePrint Archive, Report 2009/501 (2009). http://eprint.iacr.org/
9. de Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009)
10. Eisenbarth, T., Gong, Z., Güneysu, T., Heyse, S., Indesteege, S., Kerckhof, S., Koeune, F., Nad, T., Plos, T., Regazzoni, F., Standaert, F., van Oldeneel tot Oldenzeel, L.: Compact implementation and performance evaluation of block ciphers in attiny devices. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172–187. Springer, Heidelberg (2012)
11. Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., Uhsadel, L.: A survey of lightweight cryptography implementations. IEEE Des. Test Comput. **24**(6), 522–533 (2007)
12. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: a new family of lightweight block ciphers. In: Juels, A., Paar, C. (eds.) RFID. Security and Privacy. LNCS, vol. 7055, pp. 1–18. Springer, Heidelberg (2011)
13. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917. Springer, Heidelberg (2011)
14. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 45–59. Springer, Heidelberg (2006)
15. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013)
16. Karakoç, F., Demirci, H., Emre Harmancı, A.: ITUBEE: a software oriented lightweight block cipher. In: Avoine, G., Kara, O. (eds.) Lightweight Cryptography for Security and Privacy. LNCS, vol. 8162, pp. 16–27. Springer, Heidelberg (2013)
17. Meiser, G.: Efficient implementation of stream ciphers on embedded processors. Masters Thesis, Ruhr-University Bochum (2007)
18. Rinne, S., Eisenbarth, T., Paar, C.: Performance analysis of contemporary lightweight block ciphers on 8-bit microcontrollers. In: SPEED - Software Performance Enhancement for Encryption and Decryption (2007)
19. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: an ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 342–357. Springer, Heidelberg (2011)

20. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit blockcipher CLEFIA (extended abstract). In: Biryukov, A. (ed.) Fast Software Encryption. LNCS, vol. 4593, pp. 181–195. Springer, Heidelberg (2007)
21. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: a lightweight, versatile block cipher. www.nec.co.jp/rd/media/code/research/images/twine_LC11.pdf
22. Wei, L., Rechberger, C., Guo, J., Wu, H., Wang, H., Ling, S.: Improved meet-in-the-middle cryptanalysis of KTANTAN. Inf. Secur. Priv. ACISP **2011**, 433–438 (2011)
23. Wheeler, D., Needham, R.: TEA, a tiny encryption algorithm. In: Preneel, B. (ed.) Fast Software Encryption. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (1995)
24. Wu, H.: The stream cipher HC-128. www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf
25. Wu, W., Zhang, L.: LBlock: a lightweight block cipher. In: Lopez, J., Tsudik, G. (eds.) Applied Cryptography and Network Security. LNCS, vol. 6715, pp. 327–327. Springer, Heidelberg (2011)