Efficient Implementation of eSTREAM Ciphers on 8-bit AVR Microcontrollers

Gordon Meiser*†, Thomas Eisenbarth*, Kerstin Lemke-Rust*, Christof Paar*

*Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
Email: {gordon.meiser,eisenbarth,lemke,cpaar}@crypto.rub.de

†Siemens AG
CT IC CERT
80200 Munich, Germany
Email: gordon.meiser@siemens.com

***Abstract*— This work is motivated by the question of how efficient modern stream ciphers in the eSTREAM project (Profile I) can be implemented on small embedded microcontrollers that are also constrained in memory resources. In response to this question, we present the first implementation results for Dragon, HC-128, LEX, Salsa20, Salsa20/12, and Sosemanuk on 8-bit microcontrollers. These ciphers are definitively free for any use, i.e., their use is not covered by intellectual property rights. For the evaluation process, we follow a two-stage approach and compare with efficient implementations of the AES block cipher. First, the C code implementation provided by the ciphers' designers was ported to an 8-bit AVR microcontroller and the suitability of these stream ciphers for the use in embedded systems was assessed. In the second stage we implemented Dragon, LEX, Salsa20, Salsa20/12, and Sosemanuk in assembler to tap the full potential of an embedded implementation. Our efficiency metrics are memory usage in flash and SRAM and performance of keystream generation, key setup, and IV setup. Regarding encryption speed, all stream ciphers except for Salsa20 turned out to outperform AES. In terms of memory needs, Salsa20, Salsa20/12, and LEX are almost as compact as AES.**

In view of the final eSTREAM portfolio (Profile I), Salsa20/12 is the only promising alternative for the AES cipher on memory constrained 8-bit embedded microcontrollers. For embedded applications with high throughput requirements, Sosemanuk is the most suitable cipher if its considerable higher memory needs can be tolerated.

Keywords: stream cipher, software performance, AVR microcontroller, embedded security, eSTREAM, Dragon, HC-128, LEX, Salsa20, Salsa20/12, Sosemanuk, AES.

I. Introduction

Stream ciphers are an important class of encryption algorithms. A stream cipher consists of a keystream generator and an output function [1]. The keystream generator creates a pseudorandom sequence from a small secret key and an Initial Value (IV) with the intent that the keystream appears random to a computationally bounded adversary. Almost all stream ciphers use the standard 'exclusive-or' output function, i.e., the sender obtains one bit ciphertext by the 'exclusive-or' (XOR) of one bit keystream and one bit plaintext, see Fig. 1. Accordingly, the receiver computes one bit of plaintext by the 'exclusive-or' of one bit keystream and one bit ciphertext. Stream ciphers have often been designed for specific applications such as GSM, UMTS, and Wireless Networks. There has been still a lack of alternative stream ciphers in the public domain.

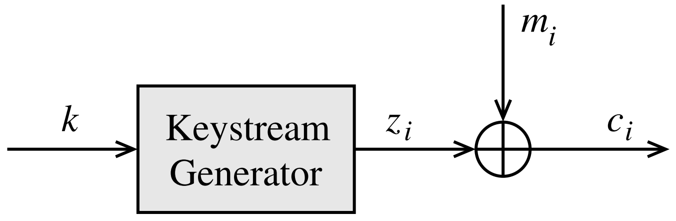

Fig. 1. Encryption of a stream cipher with an 'exclusive-or' output function denoting plaintext m_i, secret key k, keystream z_i, and ciphertext c_i.

In contrast to stream ciphers, block ciphers are said to be memoryless [1], i.e., they apply the same encryption function to successive blocks of plaintexts. However, a block cipher can be employed as a keystream generator by adding small amounts of memory using special modes of operation, e.g., the counter mode [1].

In 2005, the European Network of Excellence in Cryptology (ECRYPT) launched a call for stream cipher primitives [2] to identify new stream ciphers suitable for widespread adoption that may also serve as an alternative for the block cipher Advanced Encryption Standard (AES) [3]. Profile I of the eSTREAM project asked for stream ciphers for software applications with high throughput requirements, while Profile II aims at identifying stream ciphers suitable for hardware applications with restricted resources such as limited storage, gate count, or power consumption. Long-term security is the most important criterion for eSTREAM since it is essential to achieve confidence about the resistance of the ciphers against cryptanalytic attacks. However, the performance of the stream cipher primitive is also a decisive factor for the eSTREAM project.

The current testing framework of the eSTREAM project [4]–[6] exclusively targets general-purpose 32-bit and 64-bit CPUs for Profile I candidates. Given the great importance of small embedded controllers in the real world (the market share of embedded processors is more than 99%), we feel that such an evaluation is of value.

This paper is driven by the question of how efficient candidates in the focus of Profile I can be implemented on constrained 8-bit embedded microprocessors. Small 8-bit microprocessors are constrained in resources such as flash memory and RAM. Besides throughput, efficiency has an additional meaning in this context: resources needed by an implementation of a stream cipher should be kept small, since embedded applications are very often cost constrained. In fact, in many situations cost (given by memory consumption) is more crucial than throughput, in particular since many embedded applications only crypt small payloads. 8-bit embedded microprocessors are widely used in various embedded applications, including smart cards, household appliances, industrial controls, cars and many more systems. Modern cars, for instance, are equipped with more than fifty microcontrollers. In embedded systems, cryptography is needed for authentication, secure messaging, and software download. In this work, we evaluate performance aspects of stream ciphers that are definitively not restricted by any intellectual property conditions. In detail, this work covers Dragon [7], HC-128 [8], LEX [9], Salsa20 [10], Salsa20/12 [11], and Sosemanuk [12].

On April 15, 2008, the eSTREAM project announced a final portfolio [13] of more successful stream ciphers at the end of the eSTREAM project. This final portfolio of the software candidates comprises (in alphabethical order) HC-128, Rabbit, Salsa20/12, and Sosemanuk. Salsa20/12 is a variant of the stream cipher Salsa20 that is reduced from twenty to twelve iterations of an internal round function. In [13], Salsa20/12 is said to still provide a comfortable margin for security. Dragon is assessed to be a highly successful design that has resisted all attacks so far, but is not included in the portfolio because performance does not compare too well with the other candidates of the final portfolio [13]. LEX is not considered for the final portfolio as very recently new cryptanalytic results on the cipher were found that lead to a break of the eSTREAM security claims [13], [14]. Rabbit [15] is a patented stream cipher so that it is not free for commercial use and not in the scope of our work.

Section II provides a brief description of each cipher. An overview of the parameters of each cipher is given in Table I. In this work we concentrate on the minimum key length of 128 bit which is seen as an adequate security level for contemporary applications [13].

For the first evaluation on the suitability of the Profile I candidates, the C code implementation provided by the designers was ported to an 8-bit AVR microcontroller. For comparison, we also evaluated the byte-oriented AES taken from Gladman [16]. Our comparison metric includes (i) throughput of keystream, (ii) time needed for key setup, (iii) time needed for IV setup, (iv) memory allocation in flash (program code), and (v) memory allocation in SRAM (variables). In our second stage we implemented Dragon, LEX, Salsa20, Salsa20/12, and Sosemanuk in assembler to reach the full potential of an embedded implementation for each cipher. The metrics of the stream cipher are now compared to an efficient AES implementation in assembler that is based upon [17].

TABLE I

CHARACTERISTIC SIZES OF THE FOCUSED CIPHERS WITH THE SECURITY LEVEL OF 128 BIT. THE LAST COLUMN INDICATES WHETHER THE CIPHER IS INCLUDED IN THE FINAL eSTREAM PORTFOLIO.

Cipher	**Key Size** [bits]	**IV Size** [bits]	**State Size** [bits]	Final eSTREAM Portfolio
Dragon	128	128	1088	no
HC-128	128	128	32768	yes
LEX	128	128	256	no
Salsa20	128	128	512	no
Salsa20/12	128	128	512	yes
Sosemanuk	128	128	384	yes

II. A Brief Description of the Stream Ciphers

A. *Dragon*

Dragon is a word-based stream cipher using 32-bit words [7]. This cipher is based on a word-oriented non-linear feedback shift register (NLFSR) utilizing an update function and a 64 bit memory, acting as a counter during keystream generation. The update function is used during key initialization phase as well as during keystream generation phase and bijectively maps 192 bit values. It uses XOR addition, addition modulo 2^{32} for mixing and six 32×32 substitution boxes (s-boxes) for adding non-linearity that are deduced from two 8×32 bit s-boxes. The update function serves as a feedback generator for the NLFSR. During keystream generation phase it is also used as a non-linear output function.

Storage requirements include 2048 bytes to store Dragon's two 8×32 s-boxes. The size of the NLFSR is chosen quite big (128 byte) for an implementation on 8-bit microcontrollers because many low cost microcontrollers have RAM in the same order of magnitude.

B. *HC-128*

HC-128 is also a 32-bit word-based stream cipher [8]. The main part of HC-128 are two huge secret tables, each consisting of 512 32-bit elements. The algorithm makes use of different non-linear functions during initialization phase and during key stream generation phase. During initialization phase the secret key and IV are expanded into the two tables. Then the cipher is executed and its output is used to replace all table entries once. During each step of the keystream generation phase, one table element is updated using a non-linear function. Hence the two tables are updated after 1024 steps. In each step one 32-bit element is produced as an output.

The two secret tables of the HC-128 have a total size of 4096 bytes. This exceeds the RAM of many 8-bit microcontrollers by far. The 32-bit design as well as the possibility of parallelization do not yield an advantage on a microcontroller.

C. LEX

The LEX (Leak EXtraction) stream cipher is a modification of the AES block cipher [9]. It uses internal states of each round to create the keystream. Amongst the candidates of this framework, it is the only non-32-bit word oriented cipher. For further information regarding AES, refer to [3]. The setup phase is a simple AES encryption of the IV with the secret key. From now on the encrypted IV is repeatedly re-encrypted. During each of these encryptions 320 bits of keystream are extracted, 32 bits from each round of the AES. The 32 bits are extracted right after the ShiftRows [3] operation at different locations for even and odd rounds.

Efficient implementations of the AES are available in hardware as well as in software [16], [18]. Because of the higher output of the LEX (320 bits instead of 128 bits for each full AES encryption), a speed up of a factor of roughly 2.5 can be expected.

Very recently, serious cryptanalytic results on the cipher LEX have been reported [13], [14]. The new attack requires (at the moment) about $2^{30.5}$ chunks of 320 bits encrypted under the same key, and has a running time of 2^{112} [14]. As far as eSTREAM is concerned this attack is a break of the security claims, i.e., LEX is proved to offer less than 2^{128} security level.

D. Salsa20 and Salsa20/12

Salsa20 [10] uses a hash function with a 64-byte input and a 64-byte output to generate the keystream. Its input consists of an 8-byte random value and an 8-byte counter, together composing an 128-bit IV. Key and IV are padded with a standard sequence to reach the required input length of 64 bytes. Since Salsa20 does not need to prepare an internal state, there exists no setup phase. Due to the counter's length a re-keying is required every 2^{70} bytes. The hash function is built up of word mixing, addition modulo 2^{32}, XOR, and word-wise rotation.

The original Salsa20 cipher uses 20 iterations of the round function, whereas Salsa20/12 [11] is reduced to 12 rounds with the sole purpose of saving execution time. In [13], Salsa20/12 is assessed to already provide a comfortable margin for security. As Salsa20/12 is included in the final portfolio of eSTREAM [13], we also added performance results of Salsa20/12 in our work.

The lack of a huge state and the simplicity of the operations favors the cipher's implementation on an 8-bit microcontroller.

E. Sosemanuk

The Sosemanuk stream cipher is based on the SNOW 2.0 stream cipher and uses parts of the SERPENT block cipher as well [12]. Like most of the other ciphers it is based on 32-bit words. The cipher consists of a 10-word LFSR and a 2-word finite state machine (FSM) to produce the output. For the setup phase a modified SERPENT, reduced to 24 rounds, is used to fill the LFSR as well as the FSM. During key stream generation four consecutive output words of the FSM are fed through the SERPENT's third s-box and then XOR-ed to the output words of the LFSR to produce the keystream.

Like LEX, Sosemanuk may benefit from efficient existing implementations, SNOW and SERPENT in this case.

TABLE II
SPECIFICATION OF THE MOST POPULAR AVR DEVICES (ATMEGA FAMILY).

Device	Flash [kbyte]	SRAM [byte]
ATmega8	8	1024
ATmega16	16	1024
ATmega32	32	2048
ATmega64	64	4096
ATmega128	128	4096
ATmega128l	128	8192

III. AVR MICROCONTROLLER FAMILY

AVR microprocessors are a family of 8-bit RISC microcontrollers. The individual device classes differ in SRAM and flash memory size, as listed in Table II. Its memory is organized as a Harvard architecture with a 16-bit word program memory and an 8-bit word data memory. The devices of the ATmega series have 32 general purpose registers of 8-bit word size. Most of the microcontroller's instructions are one-cycle. All of the microcontrollers listed in Table II can be clocked at up to 16 MHz. Further information belonging to the devices of the ATmega series from Atmel can be found at [19].

Due to its easy usage, its low power consumption and the freely available development tools, the AVR microcontrollers are widely used in many areas of embedded systems. Typical application areas involving security are wireless sensor networks (WSN) and smart cards [20].

IV. FRAMEWORK SET-UP AND TOOLS

In this section we describe how the API-conform eSTREAM implementations are ported to AVR microcontrollers. We also give an introduction to the software development tools and how we measure clock cycles, flash size, and SRAM size.

A. Porting to AVR Microcontrollers

The ciphers come with a set of associated files according to the eSTREAM API [21]. In order to reduce the size of the code and to solve the dependencies we move only the parts of each file that are required for execution into one AVR file. One problem in porting code to an AVR microcontroller is the limited amount of SRAM. A solution for saving valuable SRAM lies in moving s-boxes or comparable big static data arrays into flash memory. Another issue are the different sizes of integer variables in a 32-bit-oriented environment and an 8-bit-oriented environment of an AVR. Thus all variables used have to be adapted to the standard integer sizes of the AVR microcontroller.

B. Development Tools

For the software development we used the tool 'WinAVR' [22]. This is a suite of executable, open source software development tools for the Atmel [23] AVR series of RISC microprocessors hosted on the Windows platform. WinAVR contains avr-gcc (compiler), avrdude (programmer), avr-gdb (debugger) and a tool for automatic makefile generation. Once the code compiles without errors we used the output files from WinAVR to execute the code in 'AVR Studio 4' [24] and simulate it on a chosen AVR device. AVR Studio 4 is an Integrated Development Environment (IDE) for writing and debugging AVR applications on the Windows platform. We are able to use all the functions as known by common debugging tools such as watching registers and variables. At every state we can obtain the number of CPU cycle counts, which enables us to measure clock cycles for benchmarking throughput.

Code size and SRAM size are measured in WinAVR. After compilation, WinAVR shows a summary of the used flash and SRAM size. We use the tool avr-sizex [25] to display these values in percentage. It has to be pointed out that SRAM size as provided by WinAVR includes only static variables that are initialized at the beginning. Because of that, WinAVR returns a smaller value than the actual required SRAM size. Especially, the cipher specific structure `ECRYPT_ctx` is not included in this value because it is initialized during runtime and is non-static. To provide a better statement on actual SRAM size the byte size of the cipher specific structure `ECRYPT_ctx` is calculated manually and added to the size of the static variables.

C. C language configuration

The test sequence in C language is the following (all data in pseudo code):

```
ECRYPT_init()
ECRYPT_keysetup(key)
ECRYPT_ivsetup(iv)
ECRYPT_process_bytes(encryption,
blocksize)
ECRYPT_ivsetup(iv)
ECRYPT_process_bytes(decryption,
blocksize)
```

The parameter `blocksize` is adapted for each cipher. After each call of a function the CPU cycles needed are recorded by using AVR Studio 4. This configuration is very close to the one in the eSTREAM API. We encrypt one block of size `blocksize` under the key `key`, the IV `iv` and a plaintext. After encryption we do a new keysetup and decrypt the created ciphertext. If we get the original plaintext, the test succeeds.

D. Assembly language configuration

The common configuration in assembly language looks like this:

```
call PREINIT
call INIT
call KEYSETUP
call IVSETUP
call ENCRYPT
call END
```

In the `PREINIT` phase we initialize the stack pointer and define the start address of the SRAM. After that, in the `INIT` phase we write they *key*, *IV* and *plaintext* in the flash memory. Key setup, IV setup and encryption are the same as in the C configuration. Because a microcontroller is a finite state machine (FSM), the algorithm ends in an endless loop, jumping all the time to the marker `END`.

V. Results for C Implementations

This section provides the results on efficiency of the implementation in C language. Details on the framework used are provided in Section IV-C.

A. Memory Usage

A microcontroller holds restrictions in the size of available flash memory and SRAM. Flash memory is used to store static information like program code or huge look-up tables. The usually even smaller SRAM is used for dynamic access during program execution. The memory requirements of each cipher's implementation determine the smallest possible AVR device.

Table III shows the memory allocation in flash memory. In more detail, Table III provides (i) the size of the flash memory which is used to store the program code, (ii) the required size of flash memory for the storage of static arrays, like s-boxes for instance, (iii) the total size of program code and static arrays, and (iv) the associated AVR device, i.e., the smallest device on which the implementation of the cipher can be executed without errors.

In the tables there exists an implementation named 'Salsa20 V2'. This is a slightly modified version of the original Salsa20 implementation. We merely substitute the original rotation macro with four improved rotation functions for left rotation of 7, 9, 13 and 18 bits. As shown in Table V with our improved version of Salsa20, there is a great saving of cycles when replacing the rotation macro with a rotation function that uses permutation of bytes prior to rotations and adapts better to an 8-bit microcontroller. The original macro does 32 shifts, no matter how many bits should be rotated. This improvement saves nearly 75% of cycles needed by the original macro. Accordingly, we built the implementation named 'Salsa20/12 V2' for the cipher Salsa20/12.

AES, Dragon, LEX, Salsa20, Salsa20 V2, Salsa20/12 V2,

TABLE III

MEMORY ALLOCATION IN FLASH OF C IMPLEMENTATIONS.

Cipher Implementation	Program Code [byte]	Static Arrays [byte]	Memory (Total) [byte]	[percentage]	Device
AES	4616	2048	6664	40,67%	ATmega16
Dragon	55386	2048	57434	43,82%	ATmega128
HC-128	23100	0	23100	17,62%	ATmega128l
LEX[3]	16278	5120	21398	65,30%	ATmega32
Salsa20	4478	0	4478	54,66%	ATmega8
Salsa20 V2	3842	0	3842	46,90%	ATmega8
Salsa20/12 V2	3842	0	3842	46,90%	ATmega8
Sosemanuk (M)	42656	2048	44704	68,21%	ATmega64
Sosemanuk (F)	22600	2048	24648	75,22%	ATmega32

TABLE IV

MEMORY ALLOCATION IN SRAM OF C IMPLEMENTATIONS.

Cipher Implementation	ECRYPT_ctx [byte]	Static Variables [byte]	Total SRAM [byte]	[percentage]	Device
AES	241	88	329	32.13%	ATmega16
Dragon	405	424	829	20.24%	ATmega128
HC-128	4324	232	4556	55.62%	ATmega128l
LEX[3]	232	200	432	21.09%	ATmega32
Salsa20	64	258	322	31.45%	ATmega8
Salsa20 V2	64	258	322	31.45%	ATmega8
Salsa20/12 V2	64	258	322	31.45%	ATmega8
Sosemanuk (M)	448	192	640	15.63%	ATmega64
Sosemanuk (F)	448	192	640	31.25%	ATmega32

Sosemanuk (M), and Sosemanuk (F)[1] can not be reduced to smaller AVR devices because of their consumption of flash memory[2]. HC-128 instead has to use the ATmega128l device because of its immense usage of the SRAM shown in Table IV.

Table IV has nearly the same structure as Table III, but focuses on the amount of SRAM needed by the ciphers. Column 2 of Table IV shows the requirement of the cipher specific structure `ECRYPT_ctx`, which represents the internal state of the cipher. This value is important because WinAVR disregards dynamic variables in the value given in Column 3 of Table IV as discussed in Section IV-B.

B. Performance

In the following performance benchmarks we use input and output arrays that are of the block size of each cipher. This means that one block of data is encrypted with each cipher.

Table V shows the number of cycles for the initialization, key setup, IV setup, and encryption for each cipher. As seen in Table V HC-128 has an immense cycle count in IV setup function. However, HC-128 achieves the lowest number of cycles for encryption of one block of data.

[1]If it is necessary to get the Sosemanuk cipher running on a smaller device than an ATmega64, this goal can be achieved by replacing all macros used by Sosemanuk (M) with functions, i.e., Sosemanuk (F). In this case the needed size of flash memory shrinks to 24,648 bytes. The big drawback of this modification is the reduced encryption speed as visible in the Tables V and VI.

[2]Though the flash size entries of Table III seem to indicate that AES, Dragon, Salsa20 V2, and Salsa20/12 V2 can be implemented on a smaller AVR device, actually, this is not possible.

[3]For the discussion on the security of LEX please refer to Section II-C.

TABLE V

PERFORMANCE OF INITIALIZATION, KEY SETUP, IV SETUP, ENCRYPTION OF C IMPLEMENTATIONS. ALL NUMBERS GIVEN ARE MEASURED CPU CYCLES.

Cipher Implementation	Init.	Key Setup	IV Setup	Encryption
AES	586	6953	196	12574
Dragon	2700	2136	24052	24227
HC-128	1452	460	2082876	10804
LEX[3]	1426	2619	7367	8061
Salsa20	1700	249	71	90802
Salsa20 V2	1700	248	70	48942
Salsa20/12 V2	1700	248	70	31414
Sosemanuk (M)	1282	32851	33972	14134
Sosemanuk (F)	1282	56327	61149	19938

Table VI focuses on the throughput of the encryption function for each cipher while Table V gives the number of cycles for one block size. In more detail, Table VI provides (i) the corresponding block size, (ii) the quotient of the count of cycles from Table V and the block size, and (iii) the throughput of the encryption function. The throughput is computed by dividing the CPU clock (assuming 8 MHz) by the quotient of the count of cycles and block size.

As shown in Table VI as well as in Fig. 2 and Fig. 3, the ciphers can be classified into three groups regarding throughput. HC-128, Sosemanuk (M), Dragon, LEX, and Sosemanuk (F) are in the fast group. Salsa20 V2, AES and Salsa20 reside in the slow group. Salsa20/12 V2 achieves medium throughput. Notable is the fact that the Salsa20 V2 implementation nearly doubles and Salsa20/12 V2 triples the throughput of the original Salsa20 implementation. Note further that for

TABLE VI

THROUGHPUT OF ENCRYPTION OF C IMPLEMENTATIONS.

Cipher Implementation	**Block Size [byte]**	**Ratio [cycles/byte]**	**Throughput [bytes/sec] @8MHz**
AES	16	785.88	10180
Dragon	128	189.27	42267
HC-128	64	168.81	47390
LEX[3]	40	201.53	39697
Salsa20	64	1418.78	5639
Salsa20 V2	64	764.72	10461
Salsa20/12 V2	64	490,84	16298
Sosemanuk (M)	80	176.68	45281
Sosemanuk (F)	80	249.23	32100

long keystreams the encryption is the dominant factor (but remember the huge time for IV setup of HC-128).

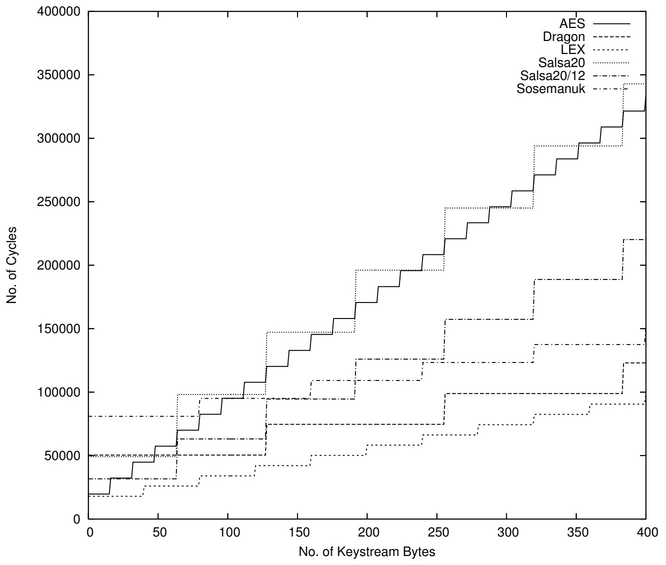

Fig. 2. Cycles versus number of keystream bytes for up to 400 bytes of keystream. Results are shown for AES, Dragon, LEX, Salsa20 V2, Salsa20/12 V2, and Sosemanuk (M). Note that HC-128 is not printed here as it is by far the least efficient cipher in this domain. LEX turns out to be most efficient.

When only a small amount of keystream has to be generated, it can be seen from Table VI and Fig. 2 that LEX is the most efficient. Until 48 byte keystream, AES is the second most efficient that is then passed by Dragon.

VI. RESULTS FOR ASSEMBLER IMPLEMENTATIONS

This section provides the results on efficiency of the implementation in assembler. Details on the framework used are provided in Section IV-D. We used an efficient assembler implementation of the AES cipher [17] to be able to compare our assembler implementations of Dragon, LEX, Salsa20, Salsa20/12, and Sosemanuk to that version. We did not implement HC-128 in assembler because of its big SRAM memory needs that prevent from using any small AVR device.

We modified the AES implementation from [17] to make it even smaller and faster. The decryption functionality has been removed as it is not needed for keystream generation. Furthermore, we contributed our own implementation of the MixColumns function that is better in respect to both, performance and code size.

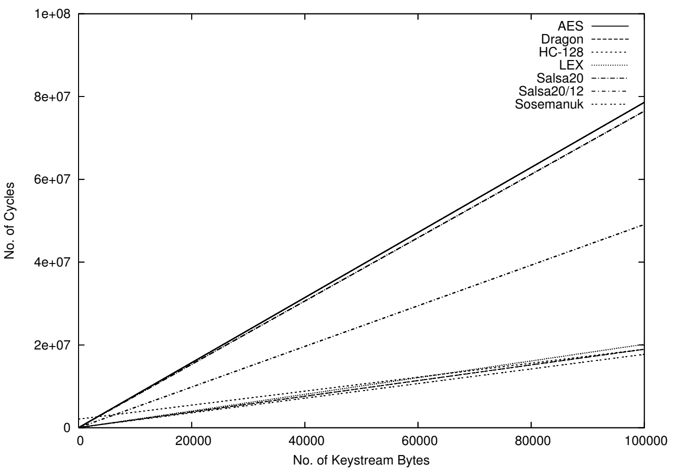

Fig. 3. Cycles versus number of keystream bytes for up to 100,000 bytes of keystream. Results are shown for AES, Dragon, HC-128, LEX, Salsa20 V2, Salsa20/12 V2, and Sosemanuk (M). In this domain three groups are clearly separated. The group of the most efficient ciphers includes Sosemanuk (M), HC-128, Dragon, and LEX (from the most efficient to the least efficient at 100,000 bytes of keystream).

For each cipher, two different implementations have been created. Dragon, Salsa20, and Sosemanuk are implemented in a function based version and a macro based version, respectively. The function based versions are realized with the goal of minimizing the use of flash memory. In contrast, the macro based versions are optimized to reach high throughput rates. Salsa20/12 is only implemented in a function based version. LEX is treated as a special case. The version called 'LEX' is the transformation of the C version of LEX into assembler that includes five substitution tables optimized for 32-bit versions. By contrast, the version named 'LEX V2' is an assembler implementation derived from the AES implementation [17].

A. Optimization of the Implementation

In this section we describe the techniques and ideas which lead to the improvements of the implementations. Because an 8-bit microcontroller operates on blocks (or words) of 8-bit size, some computations, like a word-level (32×32)-bit multiplication, are much harder to realize than on general purpose CPUs. On the other hand it is very easy to access the separate bytes of a 32-bit value directly, i.e. no shifts, rotations, logical ANDs, XORs or ORs are necessary. We now give a short overview about the functions and macros of which we generated improved versions because they possess a great potential of optimization. It is important to note that the savings in cycles mentioned below are only possible if the C code is translated into assembly code in a native and unoptimized way. However, the C compiler partly optimizes the code during translation. Because of this we do not save as many cycles in practice as the numbers below may indicate.

We start with the *ROTL32()* macro that is included in the eSTREAM API. This is very often used in the various ciphers to rotate 32-bit values by n bits to the left.

```
#define ROTL32(v, n) (U32V((v) << (n)) | \
((v) >> (32 - (n))))
```

This macro needs 32 shifts and one OR operation for execution, independent of the number n. This is not very efficient, so we developed improved versions for left rotations by a static number of bits.

Another frequently used macro is the *U8TO32_BIG()* macro which looks like this:

```
#define U32TO32_BIG(v) SWAP32(v)
#define SWAP32(v) \
  ((ROTL32(v,  8) & U32C(0x00FF00FF)) | \
   (ROTL32(v, 24) & U32C(0xFF00FF00)))
```

That means, the *U8TO32_BIG()* macro calls the *SWAP32()* macro which calls the *ROTL32()* macro. It does a swapping of the four bytes of a 32-bit value. On an 8-bit microcontroller we have the ability to access the separate bytes directly, hence we simply read the bytes in reverse order. This completely avoids the use of the *U8TO32_BIG()* macro which saves 276 CPU cycles ($(2 \cdot 32 \cdot 4)$ cycles for shift operations, $(2 \cdot 4) + 4$ cycles for binary ORs and $(2 \cdot 4)$ binary ANDs).

Dragon

The greatest potential to optimize the Dragon cipher lies in the *U8TO32_BIG()* macro, the G and H macros and the *BASIC_ROUND()* macro. As we already know we can save up to 276 cycles with every call of the *U8TO32_BIG()* macro. This macro is used 8 times during the key setup and so we save a total of 2,208 cycles. The DRAGON_UPDATE macro uses XORs, addition modulo 2^{32} and the virtual s-boxes G_1 to G_3 and H_1 to H_3. The s-box G_1 is defined as follows (in C notation):

```
#define G1(x) \
  (sbox2[x & 0xFF]) ^ \
  (sbox1[(x >> 8) & 0xFF]) ^ \
  (sbox1[(x >> 16) & 0xFF]) ^ \
  (sbox1[(x >> 24) & 0xFF])
```

To access the separate bytes of the 32-bit value x, 48 right-shifts, 4 ANDs and 3 XORs are needed, which results in a total saving of $(8 \cdot 4)+(16 \cdot 4)+(24 \cdot 4)+(4 \cdot 4)+(3 \cdot 4) = 220$ cycles for the assembler implementation. That are $1,320$ cycles for every call of the DRAGON_UPDATE macro (6 s-box calls are used in the macro) and altogether $21,120$ cycles within the 16 rounds during the IV setup. Since the *U8TO32_BIG()* macro is used 8 times during key setup, we save another $2,208$ cycles. The remaining instructions do not offer great opportunities in enhancing the speed or scaling down the code size.

LEX

As we already mentioned in Section VI we implemented two versions of LEX in assembly code. If we take a short look on the LEX based version, we see that during encryption the values of the last round s_0 to s_3, respectively t_0 to t_3, must be saved in order to compute the values of the current round. These are 16 bytes from the last round and 16 from the current round. The ATmega8 possesses only 32 registers whereby we need some of them for pointer handling and temporary values. Hence the C version swaps the values of the last round out to the SRAM. We analyzed the algorithm and found out that it is only necessary to swap six 8-bit values. This optimization saves a lot of cycles, because we are not forced to write the values back to the SRAM. The second version is derived from the modified AES implementation of [17] which we used as reference. This version is in fact faster and much smaller.

Salsa20 and Salsa20/12

The parts with the greatest potential for optimization are the left rotations by 7, 9, 13 and 18 bits included in the quarterround function. These are implemented in the optimized way. The original *ROTATE* macro, used in C language, needs 132 cycles for execution. That means that only the rotations in the quarterround function need 528 CPU cycles. The application of the 4 improved rotation versions for rotation by 7, 9, 13 and 18 bits needs only 73 cycles instead of 528 of the original rotation version. This is a total saving of $36,400$ cycles for the 80 calls of the quarterround function. The rest of the quarterround function provides no big optimization potential.

Sosemanuk

The Sosemanuk implementation features two inline functions and 20 macros to realize a successful encryption. Consequently, this circumstance increases the size of the code. Sosemanuk uses left rotation by 1, 3, 5, 7, 13, and 22 bits and shift operations by 3 and 7 bit. We can save further cycles by accessing separate bytes of 32-bit values directly and because of a reduced (32×32)-bit multiplication function. The C version multiplication requires 16 (8×8)-bit multiplications and 15 8-bit additions. If we disregard some necessary additional operations such as moving of values between registers and copy operations this way of multiplication needs 62 cycles. The improved implementation in assembler only requires ten (8×8)-bit multiplications and nine 8-bit additions, which can be summarized to 38 cycles.

B. *Memory Usage*

Table VII shows the memory allocation in flash memory, similar to Table III for the C implementations. In contrast to Table IV, for the assembler implementations there is no need for separate views on the SRAM. Hence Column 2 of Table VIII is the total size of required SRAM. Considering the function based versions and LEX V2, we notice that flash memory requirements are low for AES, Salsa20, Salsa20/12, and LEX V2, moderate for Dragon and high for Sosemanuk. High amounts of program code also typically indicate a high degree of implementation complexity. This is especially true for Sosemanuk. In terms of SRAM usage, AES, Salsa20, Salsa20/12, and LEX are again most efficient, followed by Dragon and Sosemanuk.

TABLE VII

MEMORY ALLOCATION IN FLASH OF ASSEMBLER IMPLEMENTATIONS.

Cipher Implementation	Program Code [byte]	Static Arrays [byte]	Total Memory [byte]	[percentage]	Device
AES	958	256	1214	14.81%	ATmega8
Dragon (M)	25102	2048	27150	82.86%	ATmega32
Dragon (F)	4850	2048	6898	84.20%	ATmega8
LEX[3]	1486	5120	6606	80.64%	ATmega8
LEX V2	1136	256	1392	17.00%	ATmega8
Salsa20 (M)	2984	0	2984	36.43%	ATmega8
Salsa20 (F)	1452	0	1452	17.72%	ATmega8
Salsa20/12 (F)	1452	0	1452	17.72%	ATmega8
Sosemanuk (M)	44648	2048	46696	71.25%	ATmega64
Sosemanuk (F)	9092	2048	11140	67.99%	ATmega16

TABLE VIII

MEMORY ALLOCATION IN SRAM OF ASSEMBLER IMPLEMENTATIONS.

Cipher Implementation	Total SRAM [byte]	[percentage]	Device
AES	224	21.88%	ATmega8
Dragon (M)	560	27.34%	ATmega32
Dragon (F)	560	54.69%	ATmega8
LEX[3]	304	29.69%	ATmega8
LEX V2[3]	304	29.69%	ATmega8
Salsa20 (M)	280	27.34%	ATmega8
Salsa20 (F)	280	27.34%	ATmega8
Salsa20/12 (F)	280	27.34%	ATmega8
Sosemanuk (M)	712	17.38%	ATmega64
Sosemanuk (F)	712	69.53%	ATmega16

TABLE IX

PERFORMANCE OF INITIALIZATION, KEY SETUP, IV SETUP, ENCRYPTION OF ASSEMBLER IMPLEMENTATIONS. ALL NUMBERS GIVEN ARE MEASURED CPU CYCLES.

Cipher Implementation	Init.	Key Setup	IV Setup	Encryption
AES	192	793	57	2931
Dragon (M)	756	538	21232	16648
Dragon (F)	756	537	23680	17527
LEX[3]	316	1484	5216	5502
LEX V2[3]	313	779	3413	3754
Salsa20 (M)	464	199	60	17812
Salsa20 (F)	460	199	60	18400
Salsa20/12 (F)	460	199	60	11716
Sosemanuk (M)	514	14627	8559	8739
Sosemanuk (F)	519	15252	9143	9459

TABLE X

THROUGHPUT OF ENCRYPTION OF ASSEMBLER IMPLEMENTATIONS.

Cipher Implementation	Block Size [byte]	Ratio [cycles/byte]	Throughput [bytes/sec] @8MHz
AES	16	183.19	43671
Dragon (M)	128	130.06	61510
Dragon (F)	128	136.93	58424
LEX[3]	40	137.55	58161
LEX V2[3]	40	93.85	85242
Salsa20 (M)	64	278.31	28745
Salsa20 (F)	64	287,50	27826
Salsa20/12 (F)	64	183,07	43700
Sosemanuk (M)	80	109,24	73235
Sosemanuk (F)	80	118,24	67660

TABLE XI

TIME-MEMORY TRADEOFF FOR THE ASSEMBLER IMPLEMENTATIONS.

Cipher Implementation	Time Memory Tradeoff Metric [cycles/byte][byte]
AES	222393
Dragon (M)	3531197
Dragon (F)	944541
LEX[3]	908655
LEX V2[3]	130639
Salsa20 (M)	830485
Salsa20 (F)	417450
Salsa20/12 (F)	265818
Sosemanuk (M)	5100954
Sosemanuk (F)	1317166

C. Performance

Performance benchmarks are provided in Table IX, Table X, and Table XI. Table IX shows the number of cycles for the initialization, key setup, IV setup and encryption for each cipher. Table X focuses on the throughput of the encryption function for each cipher while Table IX gives the number of cycles for one block size. Table XI introduces a time-memory tradeoff metric, i.e., the product of the ratio of cycles per keystream byte (shown in Column 3 of Table X) and the total amount of flash memory (shown in Column 4 of Table VII). Low values of this metric indicate high efficiency in the time-memory tradeoff. Considering the function based versions and LEX V2, high speeds are achieved by Sosemanuk, Dragon, LEX V2 while Salsa20 and AES are significantly slower and Salsa20/12 resides in a medium range (see also Fig. 4 and Fig. 5). In the time-memory tradeoff, LEX V2 is the most efficient followed by AES and Salsa20/12.

VII. CONCLUSION

This contribution provides the first implementation results for Dragon, HC-128, LEX, Salsa20, Salsa20/12, and Sosemanuk on 8-bit microcontrollers and therefore answers the question of how efficient freely available candidates in the focus of eSTREAM Profile I can be implemented on small embedded microcontrollers that are also constrained in memory resources. Except for Salsa20 all studied stream ciphers reach higher speeds at keystream generation than AES. In terms of memory, Salsa20, Salsa20/12, and LEX can be implemented

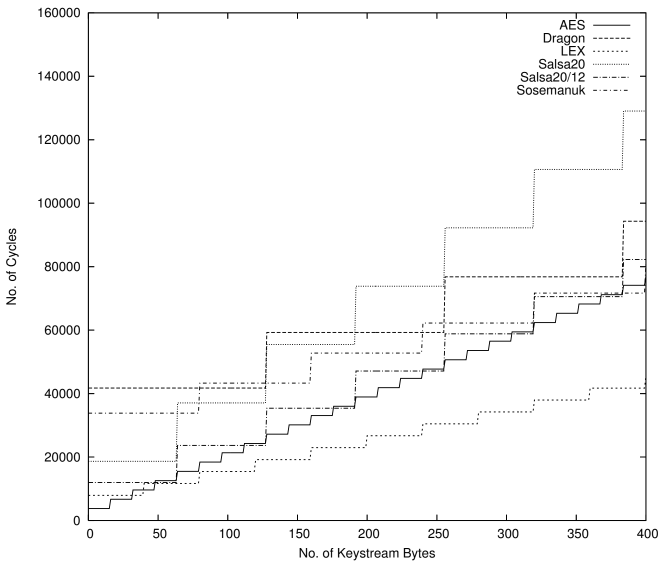

Fig. 4. Cycles versus number of keystream bytes for up to 400 bytes of keystream. Results are shown for AES, Dragon (F), LEX V2, Salsa20 (F), Salsa20/12 (F), and Sosemanuk (F). LEX V2 turns out to be most efficient.

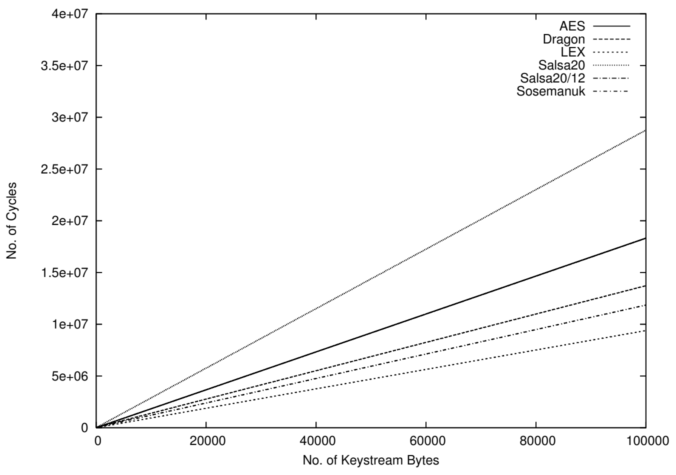

Fig. 5. Cycles versus number of keystream bytes for up to 100,000 bytes of keystream. Results are shown for AES, Dragon (F), LEX V2, Salsa20 (F), Salsa20/12 (F), and Sosemanuk (F). The sequence starting from the most efficient is LEX V2, Sosemanuk (F), Dragon (F), Salsa20/12 (F), AES, and Salsa20 (F). Note that Salsa20/12 (F) and AES are very close to each other.

almost as compact as AES, while Dragon and Sosemanuk require noticeable more memory resources and may be sub-optimum for embedded applications with very low memory constraints. Because of high SRAM memory requirements, HC-128 is assessed to be not suitable for small embedded microcontrollers. In the time-memory tradeoff metric, only LEX yields significantly better results than AES. In respect of the recently announced serious cryptanalytic attack against LEX [13], [14], it is strongly recommended to observe further developments before considering whether the remaining security level of LEX can be still sufficient for certain applications.

In view of the final eSTREAM portfolio for Profile I [13], Salsa20/12 is the only promising alternative for the AES cipher on memory constrained 8-bit embedded microcontrollers. For embedded applications with high throughput requirements, Sosemanuk is the most suitable cipher if its considerable higher memory needs can be tolerated.

References

[1] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.
[2] "Call for Stream Cipher Primitives, Version 1.3, 12th April 2005," eSTREAM, ECRYPT Stream Cipher Project, 2005. [Online]. Available: http://www.ecrypt.eu.org/stream/call/
[3] "FIPS 197: Announcing the Advanced Encryption Standard (AES)," November 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[4] "eSTREAM – Update 1, September 2, 2005," eSTREAM, ECRYPT Stream Cipher Project, Report 2005/057, 2005. [Online]. Available: http://www.ecrypt.eu.org/stream/papersdir/057.pdf
[5] C. D. Cannire, "eSTREAM testing framework," eSTREAM, ECRYPT Stream Cipher Project. [Online]. Available: http://www.ecrypt.eu.org/stream/perf/
[6] D. J. Bernstein, "Notes on the ECRYPT Stream Cipher project (eSTREAM)." [Online]. Available: http://cr.yp.to/streamciphers.html
[7] K. Chen, M. Henricksen, W. Millan, J. Fuller, L. Simpson, E. Dawson, H. Lee, and S. Moon, "Dragon: A fast word based stream cipher." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/dragon/dragon_p3.pdf
[8] H. Wu, "The Stream Cipher HC-128." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf
[9] A. Biryukov, "A New 128-bit Key Stream Cipher LEX." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/lex/lex_p3.zip
[10] D. J. Bernstein, "Salsa20." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/salsa20/salsa20_p3.zip
[11] ——, "Salsa20/8 and Salsa20/12." [Online]. Available: http://www.ecrypt.eu.org/stream/papersdir/2006/007.pdf
[12] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert, "SOSEMANUK, a fast software-oriented stream cipher." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_p3.pdf
[13] S. Babbage, C. Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw, "The eSTREAM portfolio," eSTREAM, ECRYPT Stream Cipher Project, 2008. [Online]. Available: http://www.ecrypt.eu.org/stream/portfolio.pdf
[14] O. Dunkelman, "Personal communication, dated April 23, 2008."
[15] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, "The Stream Cipher Rabbit." [Online]. Available: http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf
[16] B. Gladman, "Brian Gladman's Home Page." [Online]. Available: http://fp.gladman.plus.com/AES/index.htm
[17] "RijndaelFurious Implementation," 01 2008, http://point-at-infinity.org/avraes/.
[18] "AES Lounge." [Online]. Available: http://www.iaik.tugraz.at/research/krypto/AES/
[19] "Specification papers of the devices of the ATmega series." [Online]. Available: http://www.atmel.com/dyn/products/devices.asp?family_id=607#760
[20] Atmel, "Overview of Secure AVR Microcontrollers 8-/16-bit RISC CPU," 2007, http://www.atmel.com/products/SecureAVR/.
[21] "eSTREAM Optimized Code HOWTO," eSTREAM, ECRYPT Stream Cipher Project, 2005. [Online]. Available: http://www.ecrypt.eu.org/stream/perf/
[22] "WinAVR : avr-gcc for Windows, Release 20060421, 21th April 2006." [Online]. Available: http://winavr.sourceforge.net/
[23] "ATMEL Corporation." [Online]. Available: http://www.atmel.com
[24] "AVR Studio 4.12, Atmel Corporation, Build 460, November 2005." [Online]. Available: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
[25] "AVR-sizex." [Online]. Available: http://alt.kreatives-chaos.com/download/avr-sizex.exe