Throughput/code size trade-off for stream ciphers

Cédric Lauradoux

INRIA, Domaine de Voluceau, Rocquencourt,
BP 105, 78153 Le Chesnay Cedex, France
Email : cedric.lauradoux@inria.fr

Abstract. The profile 1 submissions to the eSTREAM call for stream ciphers aim at achieving a high throughput in software. But, for the embedded systems, the trade-off between the throughput and the code size is more critical. We here study the ROM footprints of several eSTREAM stream ciphers on an ARM920T processor. Most notably we propose some modifications in the implementations of several ciphers which lead to a best throughput/code size trade-off.

1 Introduction

The eSTREAM Profile I stream ciphers are dedicated to software applications with high throughput requirements. Then, for this profile, most submitted cipher implementations use lookup tables, inlining or loop unrolling to achieve high performance. The main drawback of those code transformations [1] is an important code size. But, for embedded systems, the code size is a critical issue. In this context, an important task consists in determining the best trade-off between the code size and the throughput for the different stream cipher proposals.

In this paper, we study the descriptions of several stream ciphers and we improve their throughput/size ratios. Most notably, we focus on AES-CTR, Salsa20 and SNOW 2.0. Indeed, we point out that the reference description is not always suitable for size constraints. Restructuring the algorithm may allow some modifications of the implementations based either on code factorization, or on invariant detection, or on code specialization. For instance, Salsa20 allows code factorization, which does not affect the performance. The counter mode of operation may be at the origin of some redundant computations in the implementation of the underlying block cipher. Code specialization depending on the size of the plaintext can also improve the performance like for SNOW 2.0.

The next section describes the benchmark process that we use. Based on this measure, we give the speed and the code size of most eSTREAM

Profile I candidates and of the different reference ciphers, for their reference implementations on an ARM920T processor. Then, the following sections describe some modified implementations leading to a better throughput/code size trade-off for three different stream ciphers. Section 3 is devoted to a faster implementation of AES-CTR. Section 4 studies the unrolling factor for LFSR-based stream ciphers, especially for SNOW 2.0. Finally, Section 5 shows how the code size of Salsa20 can be significantly reduced.

2 Benchmark process

We have benchmarked the eSTREAM profile I candidates and the reference stream ciphers on an ARM920T processor (200MHz) using gcc 3.4.3. ARM processors are 32-bit RISC cores dedicated to the embedded systems [20].

However, measuring and comparing the throughputs of stream ciphers is a difficult task: actually the throughput highly depends on the size of the involved plaintext because of the cost of the IV setup. Therefore, we propose to use as a realistic measure of the performance of a stream cipher its average throughput over all plaintext sizes, where the plaintext sizes follow a given distribution. Namely, the cipher speed s is given by:

$$s = \sum s_i \pi_i$$

where s_i is the cipher speed for a plaintext of i bytes and π_i is the proportion of plaintexts of i bytes to encipher. An example of the distribution of the sizes of TCP packets is given in Figure 1. This estimation of packet size distribution has been done over 4,334,991 packets.

The distribution may also be trimodal [19, 9], and 40-byte, 576-byte and 1500-byte packets are then critical. The TCP packet size distribution emphasizes the importance of both the latency of the IV setup (one IV setup per packet as recommended by the RFC 3686 [14]) and the latency of the keystream generation. The same technique based on the distribution of the plaintext sizes has also been used by Whiting *et al.* [21] to measure the agility of AES candidates.

With this distribution, we have computed the speed (in cycles/byte) of most focus Profile I candidates and of the reference stream ciphers (Table 1). We have used the reference implementations of the eSTREAM testing framework [8]. Table 1 also gives the sizes of the corresponding implementations.

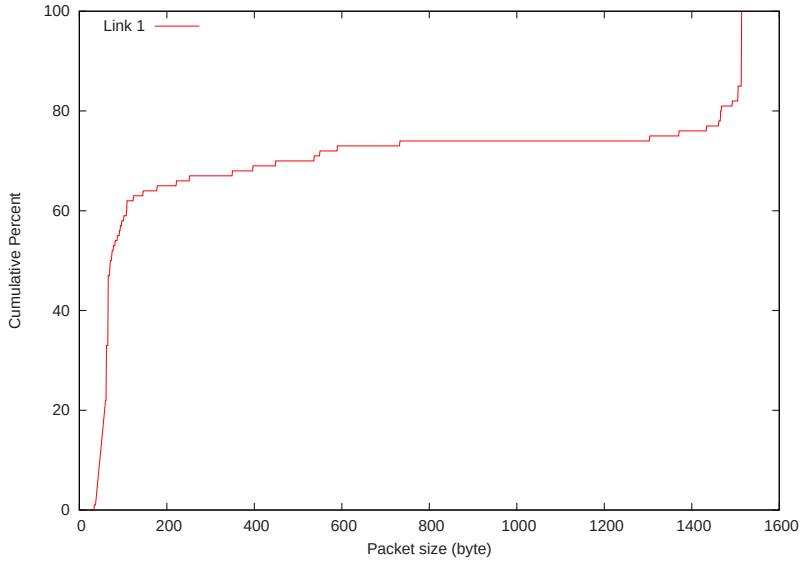

Fig. 1. Distribution of TCP packet size

		Code size (bytes)				
Ciphers	Speed (cycles/byte)	Encryption	IV setup	Key setup	Lookup tables	Total size
RC4	180	980	124	68	0	1352
Salsa20	65	1228	80	424	32	1764
Py	155	1232	304	536	256	2328
Dragon	104	1124	1720	368	2048	5260
HC-256	1316	5864	3252	132	0	9248
Snow 2.0	38	2600	3660	56	6144	12456
LEX	50	6696	156	924	5160	12936
AES	114	8088	12	1264	5160	14524
Sosemanuk	38	3580	4156	9028	2048	18812

Table 1. Comparison of the footprints and of the speeds of the focus eSTREAM Profile I candidates for the reference implementation

Table 1 then clearly points out that the characteristics of the reference implementations are very different for the considered ciphers. The first remark is that some ciphers have a large code size because they involve some relatively large lookup tables: the implementations of SOSEMANUK, SNOW 2.0, LEX and AES-CTR take at least 12 Kbytes, while SNOW 2.0 and SOSEMANUK have the highest throughput among all

these ciphers. On the other side, we have some ciphers with a much smaller code size (between 1 and 5 Kbytes) but with a lower throughput, like RC4, Salsa20, DRAGON and Py (note the results in Table 1 consider Py and not Pypy). It is worth mentioning that, among those ciphers, Salsa20 has clearly the highest throughput.

The following sections now show how we can achieve different throughput/code size trade-offs for some of these ciphers by modifying their reference implementations.

3 AES-CTR

We first focus on AES-CTR since it corresponds to the reference stream cipher which must be compared to all eSTREAM candidates.

The keystream generated by the counter mode (CTR) is produced by enciphering a set of input blocks called counters. The counters are generated from an initial value which is successively incremented by a function f. The only security requirement [12] for the incrementing function f and for the initial value generation is the counter uniqueness. The NIST [12] defines the standard family of incrementing functions for n-bit block ciphers as:

$$f_i(x) = x \oplus trunc_i(x) \ \oplus \ \left(trunc_i(x) + 1 \mod 2^i\right), \ i \leq n \qquad (1)$$

where $trunc_i(x)$ is the function which outputs the i least significant bits of x. In other words, f_i consists in incrementing by 1 the i least significant bits of x. Then, the ciphertext is the result of the exclusive-or between a plaintext block and a keystream block (Figure 2).

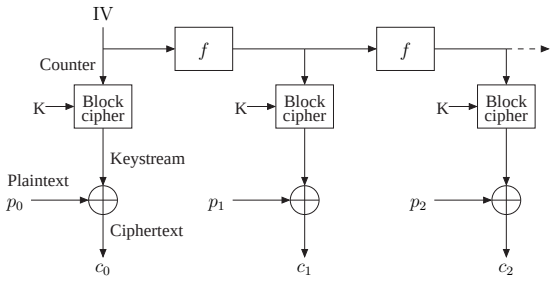

Fig. 2. CTR mode of encryption

Thus, the successive inputs of the block cipher are usually very similar, implying that some computations are redundant and can be eliminated.

To determine the amount of redundant computation, we need to study the propagation of some input differences through the block cipher, exactly as for evaluating the resistance of the block cipher to differential cryptanalysis [6] and to integral attacks [15]. Here, this study aims at detecting invariant computations when a given block cipher enciphers very similar plaintext blocks.

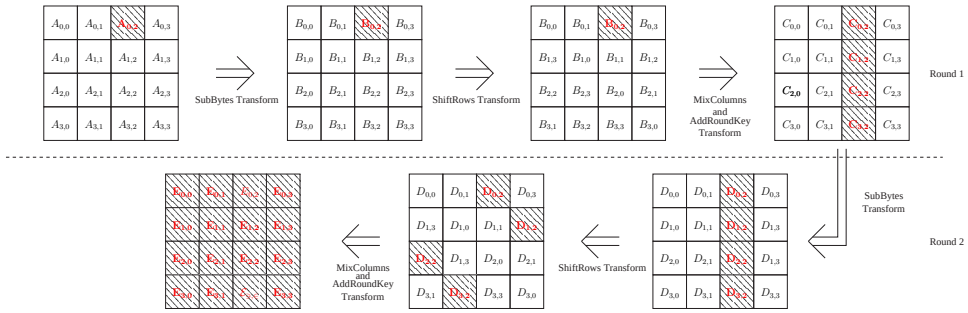

Fig. 3. Difference propagation in the AES

The round function of the AES is composed of four transformations which operate on the internal state as on a matrix of elements in $\mathbf{F}_{2^8}$. Figure 3 shows the propagation of a single byte difference through the first two rounds of the AES. Then, we can use the fact that some computations in the first two rounds are similar for two plaintexts which differ on a single byte.

If we want to exploit all invariant computations when a single byte remains invariant, we need three different implementations of the AES round function: one using the invariant computation in the first round, one using the invariant computation in the second round and one for the full round (full computation). Then, if we want to exploit the invariance of any input byte, it will requires 16 versions for the first round (one per input byte of the matrix) and 4 versions for the second round (one per column). The full round computation of the AES costs 728 bytes (with one 1024-byte lookup table). Thus the cost of implementing all variants exploiting the invariance can be very important: 1344 bytes of overhead when a single byte remains invariant. In order to keep a reasonable code size, we then only exploit the invariant computations when a single input byte remains unchanged. We obtain with this implementation, a 12% speedup (101 cycles per byte) on the standard implementation (114 cycles per

byte). The invariant implementation is more effective for large packets: we obtain a 20% speedup for packets longer than 250 bytes.

4 SNOW 2.0

SNOW 2.0 is another important reference stream cipher whose throughput can be increased. The internal state of SNOW 2.0 [13] is composed of a linear feedback shift register (LFSR) on $\mathbf{F}_{2^{32}}$ and of a finite state machine (Figure 4). First, in order to reduce the code size, we have used only one 1024-byte table for representing the involved AES transformation as observed in [5, 10]. This simple transformation reduces the code size from 12456 bytes in the reference implementation [13] to 9400 bytes. A second major property is that the LFSR can be clocked several times simultaneously in order to increase the throughput: this technique, named leap-forwarding, is well-known in hardware synthesis [18] but can also be applied in software [17, 16]. Then, if the LFSR is clocked k times, it appears that the critical code blocks are identified in two loops:

1. the first loop computes k feedbacks and k outputs.
2. the second loop shifts the LFSR by k elements.

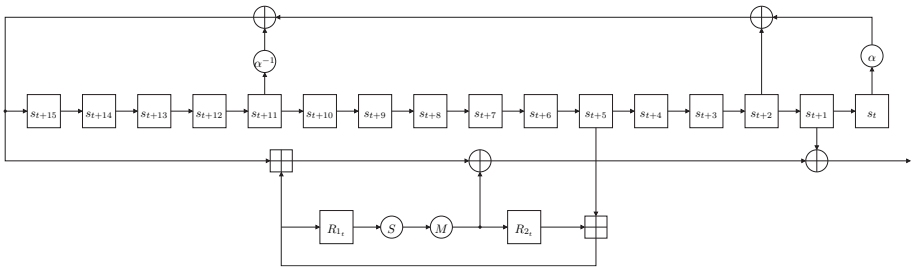

Fig. 4. SNOW 2.0: 16-word LFSR and FSM based on the AES

The parameter k is critical for the speed of the cipher. The reference C implementation of SNOW 2.0 uses a high value of k. High values of k can remove redundant shift operations. Especially, when k is a multiple of the LFSR length (16 for SNOW 2.0), the second loop can be completely eliminated and replaced by variable renaming. This transformation has two drawbacks. First, it increases the latency for small plaintexts. It also requires to fully unroll the first loop since the memory access pattern is

difficult (induction variables modulo 16). Thus, the code size significantly increases. The value $k = 16$ is the one used in the reference optimized implementation of SNOW 2.0. But other values of k may lead to different throughput/code size ratios. Table 2 shows the influence of k on SNOW 2.0 throughput and on the size of the keystream generation function. Especially, we notice that $2 \leq k < 16$ allows a very compact implementation (408 bytes compared to 2680 bytes for the reference implementation). It also achieves a significant speed enhancement for $k = 5$ and $k = 10$ (respectively 11% and 19% compared to the reference implementation).

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Code size (bytes)	360	408	408	408	408	408	408	408	408	408	408	408	408	408	408	2680
Speed (cycles/byte)	66	46	43	42	34	41	37.5	37.4	43	31	36.2	36.8	36.8	42	37	38

Table 2. Effect of k on the throughput and on the code size of the keystream function for SNOW 2.0

The best implementation of SNOW 2.0 for speed achievement keeps several versions of the code corresponding to different values of k and determines the best version to use at the runtime. Such software architectures were first introduced by Byler and al. in [7]. This implementation is more flexible than the implementation with $k = 10$ since it includes the fastest version for some other distribution. However, this implementation costs a huge amount of memory. But, for $k < 16$, all the versions can be gathered and we get a runtime-dependent implementation for only 276 bytes of code (Table 3). To include the $k = 16$ version, we still need to reduce its size (2680 bytes). It can be reduced to 596 bytes with a partial re-roll of the code. It is worth mentioning that the same transformation, *i.e.* re-rolling, can be applied to the IV setup in order to reduce the code size. Our runtime-dependent implementation is slightly slower (Table 3) than the implementation corresponding to $k = 10$. This overhead is due to some additional control operations in the runtime-dependent implementation.

Therefore, as shown in Table 3, our implementation of SNOW 2.0 can lead to a slightly better throughput while its size has been divided roughly by a factor 2. Another critical parameter for the code size of LFSR-based stream ciphers is the number of elements in the shift register which are accessed at the same time to generate the keystream bytes. For SNOW 2.0, we access 5 elements of the shift register to output 4

Implementation	k	Code size (byte)	Speed (cycles/byte)
Reference C code [13]	fixed $k = 16$	12456	38
One table for AES	fixed $k = 16$	9400	38
Partial re-roll (IV) and one version	fixed $k = 10$	5804	31
Partial re-roll and two versions	runtime	6528	34.4

Table 3. Trade-off between the code size and the throughput for SNOW 2.0

keystream bytes. If we increase the number of accessed elements, the cost of induction variables will be critical. This parameter considerably affects the code size of SOSEMANUK [2] and of DRAGON [11], implying that the previously described modifications do not lead to such improvements for these eSTREAM candidates.

5 Salsa20

Salsa20 is an eSTREAM candidate designed by Bernstein [3]. It is a 32-bit oriented cipher based on bitwise xor, addition modulo 2^{32} and left rotation. The Salsa20 basic transformation is called quarterround function. It operates on 4 words as shown in Figure 5.

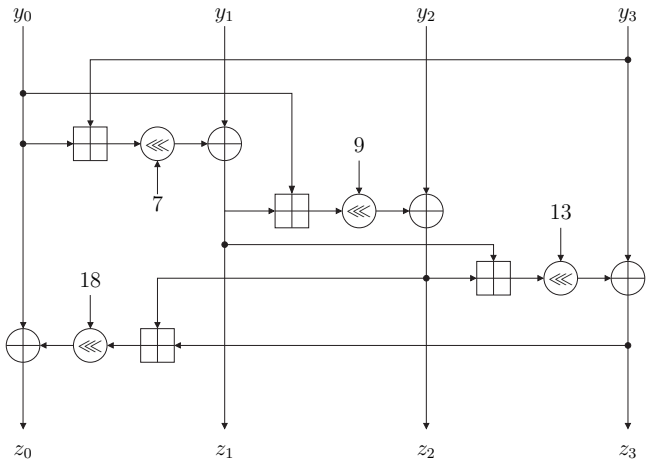

Fig. 5. Quarterround function of Salsa20

The internal state of Salsa20 is represented by 4×4 matrix of 32-bit words. Salsa20 has an iterative structure. The round function of Salsa20

is composed of two transformations: the columnround transformation applies the quarterround function on a permutation of each column of the state matrix. Then, the rowround transformation applies the quarteround function on a permutation of each row. In the full version of Salsa20, both operations are applied 10 times. A final transformation is applied to the internal state after the 10 rounds: the original internal state matrix is added the current state matrix.

The reference code of Salsa20 [3] implements the columnround and the rowround separately. This section of the code is the most memory-consuming (1902 bytes). But both transformations can be gathered in a single operation using that:

$$\text{rowround}(u) = \big(\text{columnround}(u^t)\big)^t$$

for any row vector u, where u^t denotes the transposed vector. The transposition only introduces a few additional load/store operations to the reference C code since we use a temporary state matrix to store the result of columnround.

Implementation	Code size (byte)	Speed (cycles/byte)
Reference C code [3]	1764	65
Our implementation	868	69

Table 4. Trade-off between the code size and the speed for Salsa20

This use of a single transformation is very effective: our implementation significantly reduces the code size (51%) with only 6% of overhead on the speed as shown in Table 4. With this implementation, Salsa20 appears to be one of the stream ciphers which provide a very interesting throughput/code size trade-off among the Profile I candidates.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Côme Berbain, Ollivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. SOSEMANUK, a fast software-oriented stream cipher, 2005. Submitted to eSTREAM - `http://www.ecrypt.eu.org/stream/sosemanuk.html`.
3. Daniel J. Bernstein. Salsa20, 2005. Submitted to eSTREAM - `http://www.ecrypt.eu.org/stream/salsa20p2.html`.

4. Daniel J. Bernstein. Comparison of 256-bit stream ciphers. In *The State of the Art of Stream Ciphers - SASC 2006*, 2006.
5. Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient Software Implementation of AES on 32-bit Platforms. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, Lecture Notes in Computer Science 2523, pages 159–171. Springer Verlag, 2003.
6. Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *Advances in Cryptology - CRYPTO '90*, Lecture Notes in Computer Science 537, pages 2–21. Springer-Verlag, 1990.
7. Mark Byler, Michael Wolfe, James R. B. Davies, Christopher Huson, and Bruce Leasure. Multiple version loops. In *International Conference on Parallel Processing - ICPP '87*, pages 312–318, 1987.
8. Christophe De Cannière. eSTREAM testing framework, 2005. `http://www.ecrypt.eu.org/stream/perf/`.
9. Kimberly Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. In *INET' 98*. Internet Society, 1998.
10. Matthew Darnall and Doug Kuhlman. AES Software Implementations on ARM7TDMI. In *International Conference in Cryptology in India - INDOCRYPT 2006*, Lecture Notes in Computer Science 4329, pages 424–435. Springer Verlag, 2006.
11. Ed Dawson, Kevin Chen, Matt Henricksen, William Millan, Leonie Simpson, Hoon-Jae Lee, and SangJae Moon. DRAGON: A Fast Word Based Stream Cipher, 2005. Submitted to eSTREAM - `http://www.ecrypt.eu.org/stream/dragon.html`.
12. Morris Dworkin. Recommendation for Block Cipher Modes of Operation. Technical report, National Institute of Standards and Technology, 2001.
13. Patrik Ekdahl and Thomas Johansson. A New Version of the Stream Cipher SNOW. In *Selected Areas in Cryptography - SAC 2002*, Lecture Notes in Computer Science 2595, pages 47–61. Springer Verlag, 2002.
14. Russ Housley. Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP) - RFC 3686. Technical report, Internet Engineering Task Force (IETF) - Network Working Group, 2004.
15. Lars R. Knudsen and David Wagner. Integral cryptanalysis. In *Fast Software Encryption - FSE 2002*, Lecture Notes in Computer Science 2365, pages 112–127. Springer Verlag, 2002.
16. Cédric Lauradoux. Shift Register - A Shift Register Code Generator, 2006. `http://www-rocq.inria.fr/codes/LFSR/index.html`.
17. Cédric Lauradoux. From Hardware to Software Synthesis of Linear Feedback Shift Registers. In *Workshop on Performance Optimization for High-Level Languages and Libraries - POHLL 2007*. IEEE, 2007. *To appear*.
18. Menahem Lowy. Parallel Implementation of Linear Feedback Shift Registers for Low Power Applications. In *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, pages 458–466. IEEE, 1996.
19. Sean McCreary and Kimberly Claffy. Trends in Wide Area IP Traffic Patterns: A View from Ames Internet Exchange. In *International Teletraffic Congress Specialist Seminar on teletraffic issues related to mobile systems and mobility - ITC 2000*. CAIDA, 2000.
20. David Seal. *ARM Architecture Reference Manual*. Addisson and Wesley, 1998.
21. Doug Whiting, Bruce Schneier, and Steve Bellovin. AES Key Agility Issues in High-Speed IPsec Implementations. `http://www.cs.columbia.edu/~smb/papers/AES-KeyAgile.pdf`.