Published in Workshop on Information Security Theory and Practices 2007 LNCS 4462

Survey and Benchmark of Stream Ciphers for Wireless Sensor Networks

Nicolas Fournel[1], Marine Minier[2], and Stéphane Ubéda[2]

[1] LIP, ENS Lyon - Compsys (Bureau 347)
46, Alle d'Italie - 69364 LYON Cedex 07 - France
Nicolas.Fournel@ens-lyon.fr
[2] CITI - INSA de Lyon - ARES INRIA Project
Bâtiment Léonard de Vinci
21 Avenue Jean Capelle, 69621 Villeurbanne Cedex - France
FirstName.Name@insa-lyon.fr

Abstract. For security applications in wireless sensor networks (WSNs), choosing best algorithms in terms of energy-efficiency and of small-storage requirements is a real challenge because the sensor networks must be autonomous. In [22], the authors have benchmarked on a dedicated platform some block-ciphers using several modes of operations and have deduced the best block cipher to use in the context of WSNs.
This article proposes to study on a dedicated platform of sensors some stream ciphers. First, we sum-up the security provided by the chosen stream ciphers (especially the ones dedicated to software uses recently proposed in the European Project Ecrypt, workpackage eStream [27]) and presents some implementation tests performed on the platform [16].
Keywords: stream ciphers, sensors, benchmarks.

Introduction

Sensor networks are made by the tremendous advances and convergence of micro-electro-mechanical systems (MEMS), wireless communication technologies and digital electronics. Sensor networks are composed of a large number of tiny devices or sensors which monitor their surrounding area to measure environmental information, to detect movements, vibrations, etc. Wireless sensor networks can be really useful in many civil and military areas for collecting, processing and monitoring environmental data. A sensor node contains an integrated sensor, a microprocessor, some memories, a transmitter and an energy battery. Sensor nodes communicate through a radio device in order to manage the network and to gather the produced data to a specific node called the sink node. Despite the relative simplicity of its basic components, sensor networking offers a great diversity: various hardwares (MicaZ, Telos, SkyMote, AVR or TI micro-controllers), various radio and physical layers (868MHz and 2,4GHz) using different types of modulations, various OS (TinyOS, Contiki, FreeRTOS, JITS), various constraints (real-time, energy, memory or processing), various applications (military or civil uses).

In such a context, a specific care must be invested in the design of the applications, communication protocols, operating systems and of course security protocols that will be used. Lots of protocols have been proposed to enforce the security offered by sensor networks. In despite of the increasing request in this new area of research, few articles presented real results of implementations or benchmarks concerning the security primitives which can be used in sensor networks. In [22], the authors present such results concerning theoretical aspects and benchmarks for the most famous block ciphers (including AES, MISTY1, Skipjack,...). Even if block ciphers have lots advantages compared with stream ciphers (they could be used for the both secure modes required for the sensor networks: the pairwise secure links and the secure group communications), the stream ciphers are usually used when wireless communications are required (as done in the WEP for example) because they could reach important flows for limited costs and the use of the "one time pad" encryption do not propagate errors induced by the communication channel. This cipher method combines using a modular addition (for example the XOR) the plaintext with a random key or "pad" used only once and having the same length than the plaintext. The pad also called pseudo-random sequence is produced using a pseudo-random generator or a synchronous stream cipher and is generated from the secret shared key K and an initial value IV, that must be different for each encryption. So, in the context of sensor networks, stream ciphers could be useful for pairwise secure associations.

This article then proposes to theoretically sum-up the security provided by some stream ciphers dedicated to software uses (especially the ones recently proposed in the European Project Ecrypt, workpackage eStream [27]) and presents some implementation tests performed on a dedicated platform of sensors [16].

This paper is organized as follows: Section 1 presents several stream ciphers and evaluates their current security based on the most recent results. Section 2 presents the dedicated platform and describes the methodology used to perform our benchmarks. Section 3 provides our results and our analysis concerning the benchmarking whereas Section 4 concludes this paper.

1 The studied stream ciphers

We decided to study and to benchmark the stream ciphers dedicated to software uses (Profile 1 of the eStream call for primitives) submitted to the eStream call and belonging to the Focus Phase 2 (see the website of the eStream project `http://www.ecrypt.eu.org/stream/phase2list.html` for more details of their choice). This project is ongoing so the security evaluation of the proposed stream ciphers is always in hand. Even if the security study concerning these primitives is not finished, it seems interesting for us to study their performances on a dedicated architecture with strong constraints due to their high efficiency and their high reliability in wireless context. Moreover, in most of cases, the code size required for stream ciphers is smaller than the one required for block ciphers.

We have added to those ciphers three other pseudo-random generators due to their fame and their great use: RC4 (always used in WPA and in `https`), SNOW v2 [12] (the updated version of the NESSIE call for primitives [26]) and AES-CTR (the block cipher AES used in a particular mode, the CTR one) used in WPA2. Moreover, the AES-CTR is in fact a modified block cipher and then its performances correspond to those of a block cipher. We could then compare the results obtained for it as a block cipher to those of stream ciphers.

All the stream ciphers presented in this section uses at least the following parameters as suggested in the initial call of eStream: a secret shared key K of at least 128 bits and an IV value of at least 64 bits, that must be absolutely different at each new encryption.

In our security analysis, we claim that a stream cipher is secure until now if no attack (with a complexity less than 2^{128} or respecting the recommendations of the authors) has been exhibited against it until now.

RC4 RC4 was introduced in 1987 by R. Rivest [30] for the RSA laboratories. RC4 is most commonly used to protect Internet traffic using the SSL (Secure Sockets Layer) protocol. It is composed of an initialization phase that transforms the secret shared key K of length between 40 and 1024 bits into an initial S permutation from $N = 2^n$ into itself (typically $n = 8$). The stream sequence $z(t)$ is then produced by outputting particular values of the S permutation updated at each clock.

Security In despite of many efforts provided by the cryptographers to try to break RC4, very few attacks are known against it. The strangest remains the "Finney property" (see [23] for more details). However, some statistical bias could be exhibited (see [23]) that allow to construct distinguishing attacks against RC4. An other attack proposed by S. Fulher, I. Mantin et A. Shamir [15] exploits the bad re-synchronization of RC4 and could be applied in the WEP case (see [25] and [24]).

RC4 stays a secure cipher for good initial choices: if the key scheduling algorithm is strengthened by pre-processing the base key (of at least 128 bits) and any counter or initialization vector by passing them through a hash function such as MD5 or by discarding the first 256 output bytes of the pseudo-random generator before beginning encryption (as described in [21]). Using those recommendations, we say that RC4 is secure.

SNOWv2 SNOW is a stream cipher submitted by P. Ekdahl and T. Johansson to the NESSIE call for primitives [11]. Several attacks have been exhibited against the first SNOW version ([17] and [8]) and thus obliged the authors to modify their initial submission. That has been done and a second version of SNOW, SNOW v2, have been proposed [12]. In this version, the secret shared key has a length of 128 or 256 bits whereas the use of an IV value of 128 bits is optional. This stream cipher uses an LFSR of length 16 on $GF(2^{32})$ and a non linear finite state machine called FSM. The first 32 bits output is generated after 32 clocks.

Security The only attack describes against SNOW v2 has been proposed in [32] and requires 2^{225} output words (2^{230} bits) and 2^{225} steps of analysis to distinguish the output of SNOW 2.0 from a truly random bit sequence. This attack does not really endanger the security of SNOW because it just allows to distinguish the output sequence from a perfect random one and the required complexity is not currently reachable. So, we say that SNOW v2 is secure.

AES-CTR The AES-CTR is not exactly a stream cipher (see for example [18] for more details). In fact, it uses the AES block cipher (see [14] for further details) in a particular mode of operation. The block cipher AES uses a key of length 128, 192 or 256 bits and encrypts using a parallel structure blocks of size 128 bits. The CTR mode of operations consists in ciphering a counter value - that must be used only one time for a given key as mentioned in the chapter 2 of [18] - with a particular key K and x-oring the ciphertext obtained with the corresponding block of plaintext. The counter corresponding with a IV value is then updated to cipher in "one-time pad" mode the next plaintext block.

Security The AES block cipher has been chosen in 2001 as the new block cipher standard by the NIST after 4 years of study. So, we say that this block cipher (used in all the known modes of operation) is secure. Moreover, the study of this block cipher allows us to compare the performances of it as a block cipher with the other stream ciphers.

DRAGON DRAGON [10] was submitted to the eStream call for primitives and is one of the FOCUS Phase 2 stream ciphers [27]. It is left unchanged compared to Phase 1, the initial phase of evaluation of the eStream project. Two versions have been proposed: Dragon-256 that uses a secret master key of 256 bits, and a publicly known initialization vector (IV), also of 256 bits; and Dragon-128 that uses 128-bit key and IV. The two versions uses a non-linear feedback shift register (NLFSR) of length 1024 bits and a nonlinear filter function from $\{0,1\}^{192}$ into itself with a 64-bit memory component.

Security In [13], Englund and Maximov describe a distinguishing attack against Dragon-256, under the assumption that the cryptanalyst can obtain an enormous amount of keystream from a single key-IV pair. Both variants of the distinguishing attack require 2^{155} words of keystream with an operational complexity of 2^{187} and uses 2^{32} words of memory for the first variant and with a complexity of 2^{155} in time and of 2^{96} in memory for the second variant. However, those attacks do not take into account the authors recommendations: " To protect against unknown future attacks, and against attacks that require large amounts of keystream, [Dragon] should be rekeyed at least once for every 2^{64} bits of keystream generated".

In [7], an other statistical bias has been exhibited (with a probability equal to $2^{-92.8}$). But to detect this bias the amount of keystream required for the attack is by far larger than the limit of keystream available from a single key. So, until now, we say that Dragon is secure.

HC-256 and HC-128 Two versions of HC have been proposed in [34] and in [35]. The first one HC-256 generates keystream from a 256-bit secret key and a 256-bit initialization vector whereas the second one HC-128 supports 128-bit key and 128-bit initialization vector but only 2^{64} keystream bits can be generated from each key/IV pair. The general principle of the keystream generation for HC-256 is as follows: at each clock, a 32-bit word of one of the two secret tables (initialized with the key K and the IV value) is updated using a non-linear feedback function. Each table contains 1024 32-bit words. Every 2048 steps all the elements of the two tables are updated. At each step, HC-256 generates one 32-bit output using a 32-bit-to-32-bit mapping. HC-128 is the simplified version of HC-256: it uses two secret tables, each one having 512 32-bit elements. At each clock, one element of a table is updated using a non-linear feedback function. All the elements of the two tables are updated every 1024 clocks. At each clock, one 32-bit output is generated from the non-linear output filtering function.
Security Until now, no attack have been found against HC-256 and HC-128. So we say that at this moment, the two HC versions are secure.

LEX The stream cipher LEX has been proposed by A. Biryukov [5] and has been tweaked to enter Phase 2 of estream. This new version extracts parts of the internal state at certain rounds of the block cipher AES. The AES usual key lengths could be used: 128, 192 or 256 bits. The size of the IV is 128 bits. The output sequence is generated by outputting at each AES round certain four bytes from the intermediate variables. The difference with AES is that the attacker never sees the full 128-bit ciphertext but only portions of the intermediate states.
Security The first version of LEX was successfully attacked by Hongjun Wu and Bart Preneel in [36] leading to a modified IV injection as done in the second version of LEX. Until now, no attack have been found against this new version. So we say that at this moment, LEX is secure.

Phelix Phelix is a stream cipher proposed by D. Whiting, B. Schneier, S. Lucks and F. Muller [33]. It uses a 256-bit key and a 128-bit IV value. It has an internal state that consists of nine words of 32 bits each. The state is broken up into two groups: 5 "active" state words, which participate in the block update function, and 4 "old" state words that are only used in the keystream output function. Twenty elementary rounds are applied to produce one 32-bits output block.
Security A very recent attack has been proposed by Hongjun Wu and Bart Preneel against Phelix in [37]. This attack is a differential-linear one assuming nonce reuse (corresponding with a chosen nonce attack). In this context, with 2^{34} chosen nonces and 2^{37} chosen plaintext words, the key of Phelix can be recovered with about $2^{41.5}$ operations. Even if this kind of attacks is not clearly authorized by the cryptographic community, it directly asks the question of the security of Phelix.

Py and Pypy The stream cipher Py has been proposed by E. Biham et J. Seberry in [3]. It uses the same principles of construction than RC4 on two larger tables with a rolling update under keys of length up to 256 bits and IV

of length 128 bits. At each clock, 64 bits of the output sequence are produced. The allowed stream size is 2^{64} bytes for each stream sequence. For the eStream phase 2, an other stream cipher called Pypy (see [4]) has been proposed that outputs every second word of Py (only 32 bits are outputted at each clock).
Security Many attacks have been proposed against Py and Pypy: the first one [28] (improved in [9]) do not really endanger the security of Py and Pypy because they use more than 2^{64} bytes for each stream sequence. More recently, an other series of attacks using chosen IVs to recover the secret key has been proposed in [38] and in [19]. Those attacks seem to be more devastator than the previous ones. Then the security of Py and Pypy must be more carefully studied during the second eStream Phase.

Salsa20 The stream cipher Salsa20 has been proposed by D.J. Bernstein in [2]. It uses a key with a length from 16-byte to 32-byte and an IV of length 16-byte. The core of Salsa20 is a hash function with 64-byte input and 64-byte output. The hash function is used in counter mode as a stream cipher: Salsa20 encrypts a 64-byte block of plaintext by hashing the key, nonce, and block number and xor'ing the result with the plaintext.
Security Until now, no attack has been found against Salsa20. So we say that at this moment, Salsa20 is secure.

SOSEMANUK The stream cipher SOSEMANUK has been proposed by C. Berbain *et al.* in [1]. Its key length is variable between 128 and 256 bits. It accommodates a 128-bit initial value. Any key length is claimed to achieve 128-bit security. The SOSEMANUK cipher uses both some basic design principles from the stream cipher SNOW 2.0 and some transformations derived from the block cipher SERPENT. Sosemanuk aims at improving SNOW 2.0 both from the security and from the efficiency points of view.
Security Until now, no attack with a complexity less than 2^{128} has been found against SOSEMANUK. So we say that at this moment, SOSEMANUK is secure.

2 Methodology

In this section, we present the platform used to perform the benchmarks and we also describe the testing framework.

2.1 The dedicated platform

All the benchmarks performed here are produced using a sensor platform built upon an ARM9 processor. Its processing power and the current evolution in processor size and energy consumption make it a rather good representative for next generation sensor network nodes. Nowadays the ARM7, which used to be a full featured processor, is considered as a 32 bit micro-controller, for example embedded in nearly all Bluetooth devices and in some wireless devices. Today, current sensor network data, like temperatures, require only few processing on

the nodes, but we can state that next generation sensors will capture sounds or even images which will need more powerful nodes. We then decide to use an ARM9 core based CPU architecture for its computing power.

The platform is an ARM based development board. It uses an ARM922T, more precisely an Altera Excalibur EPXA10, which is a FPGA integrating an ARM922T core and usual embedded systems peripherals (*e.g.* UART, Timers) on the same chip. The processor accesses all peripherals and memory levels through two levels of AMBA bus. Memories are organized in a three level hierarchy. First level, the nearest from the processor are caches, two 8 kB of separated cache. At the second level we can find two 256kB and two 128kB scratch pad memories not used in the benchmarks. Finally, main memory, the furthest from processor, is a 128 MB SDRAM.

As far as benchmark construction is concerned, they was compiled with the standard GCC C compiler targeted to ARM processors. For the `libc` and operating systems functionalities, we used a lightweight operating system called `Mutek` [29]. It is Posix threads capable, but for the sack of predictability, all benchmarks are mono-threaded.

The energy and time performance informations are collected thanks to a two step simulation. The first step is the full architecture simulation. The simulator we use for this step is derived from the open source `skyeye` [31] simulator. Skyeye is a functionnal simulator targeted to ARM based embedded systems. Several full platforms are available for simulation like full featured PDA. This simulator is augmented in our case for our CM922T-XA10 platform support and we also added instruction cycle accuracy timing and peripheral activity reporting. This simulator is responsible for generating the linear execution tracelater used by the second simulator `esimu`. `esimu` generates a full profile in terms of time and energy of each benchmark. The results can then be visualized with profiles visualization tools freely available like `KCacheGrind` [20].

2.2 Methodology

We have adopted the methodology provided with the eStream testing framework (see [6] for more details) because it seems to be the most relevant one to evaluate stream ciphers. Indeed, a stream cipher is composed of an initial step, called the warm up phase, that produces from the key and the IV value an internal state that will produce the first output bits or bytes. We then need to test the time required to perform the "key setup" and the "IV setup". Moreover, one of the main advantages of stream ciphers is that they are able to produce very quickly long sequences required for the ciphering operation. We then to measure this particular property.

The set of tests are performed in order to study the specific requirements on the efficiency of the primitives in various situations. The testing framework described in [6] then proposes four performance measures to test the most relevant implementation properties:

– **Encryption rate for long streams**: this aspect reflects the biggest potential advantage over block ciphers and appears as an important criterion

in many applications. We have decided to measure here the encryption rate by ciphering a long stream in chunks of about 4Kb. The encryption speed is computed in cycles/byte by measuring the cycles required to encrypt 10 such blocks under 10 different keys. The time to setup the key and the IV is not considered in this test.
- **Packet encryption rate**: while a block cipher is likely to be a better choice when encrypting very short packets, it is interesting to determine at which length a stream cipher starts to take the lead. The packet encryption rate is measured in cycles/byte for three packet lengths (40, 576 and 1500 bytes) including an IV setup and a MAC finalization if an authenticated encryption is supported (only Phelix has this property). This test is repeated under 10 different keys on several packets.
- **Key and IV setup**: The last test separately measures the efficiency of the key setup and of the IV setup. "This is probably the least critical of the four tests, considering that the efficiency of the IV setup is already reflected in the packet encryption rate, and that the time for the key setup will typically be negligible compared to the work needed to generate and exchange the key." ([6]). The tests are performed for several key and IV values and the results are provided in cycles/key or cycles/IV.
- **Agility**: When an application needs to encrypt many streams in parallel on a single processor, its performance will not only depend on the encryption speed of the cipher, but also on the time spent switching from one session to another. The testing framework performs the following test: it first initiates a large number of sessions (filling 16MB of RAM), and then encrypts streams of plaintext in short blocks of around 256 bytes, each time jumping from one session to another. The results of this test are provided in cycles/byte repeating the test on 270 blocks of 256 bytes under one key.

We also perform some tests concerning the code size required to embed such ciphers on the platform. We refer to two types of memory: the code memory in the form of flash memory and the data memory in the form of RAM. We have performed those tests on the same kind of codes for each stream cipher including a key-setup, an IV-setup and a call to the function that encrypts long streams.

3 Results

To perform our benchmarks using the previous methodology, we have used without modifying them, the C codes provided in the testing framework [6]. All the C codes are available via the webpage `http://www.ecrypt.eu.org/stream/perf/`. To obtain a point of comparison, results are also given for a simple Copy operation, this code is also provided by the testing framework.

3.1 CPU cycles and energy consumption

The results (computed using the `skyeye` + `eSimu` tools) concerning the number of cycles required to perform all the tests are summed up in the table 1. The

results concerning the energy consumption are given in the table 2. The key and the IV sizes used to perform the tests are also specified.

			cycles/byte				cycles/key	cycles/IV	cycles/byte
Algo.	Key	IV	Stream	40 bytes	576 bytes	1500 bytes	Key setup	IV setup	agility
Copy	80	80	2.19	3.72	1.00	7.58	4.40	4.19	7.78
RC4	128	0	26.97	610.95	58.53	33.29	76.41	23581.61	21.24
SNOW v2.0	128	128	25.08	66.38	16.82	23.71	163.41	2273.35	20.87
AES CTR	128	128	206.19	131.52	198.73	195.76	636.49	157.52	202.23
DRAGON	128	128	30.89	177.05	69.76	64.91	421.42	4497.61	33.60
HC-256	128	128	27.00	6044.76	446.11	183.17	141.75	198126.10	49.30
HC-128	128	128	19.35	1484.72	112.12	53.70	141.76	58194.93	31.67
LEX	128	128	47.07	71.41	40.32	41.92	501.41	1415.57	50.71
Phelix	128	128	25.61	90.15	28.36	26.77	1271.42	2154.61	26.99
Py	128	64	214.25	349.23	47.58	60.88	7713.83	9327.43	64.40
Pypy	128	56	44.78	360.95	103.91	74.72	7713.82	9660.11	73.46
Salsa20	128	64	57.54	84.57	55.05	73.07	367.70	118.07	72.60
SOSEMANUK	128	64	14.81	385.63	37.95	30.48	16374.01	1264.09	20.78

Table 1. Number of CPU cycles for the stream ciphers using the testing framework

			nJ/byte				nJ/key	nJ/IV	nJ/byte
Algo.	Key	IV	Stream	40 bytes	576 bytes	1500 bytes	Key setup	IV setup	agility
Copy	80	80	38.32	60.85	16.84	142.07	70.54	67.29	145.35
RC4	128	0	465.17	9843.25	948.49	542.06	1243.66	379636.24	354.43
SNOW v2.0	128	128	438.34	1093.46	280.59	414.20	2656.66	41749.08	365.26
AES CTR	128	128	3587.00	2197.89	3437.36	3384.26	11378.81	2861.89	3499.45
DRAGON	128	128	514.26	2912.69	1144.53	1064.58	6846.80	74109.24	575.67
HC-256	128	128	471.39	102473.69	7577.28	3112.48	2540.02	2705307.85	864.11
HC-128	128	128	342.29	24264.78	1838.04	897.21	2540.20	950661.16	559.97
LEX	128	128	804.03	1186.80	670.42	714.16	8250.66	23850.60	868.13
Phelix	128	128	421.15	1470.51	461.14	454.71	20622.78	35111.32	461.26
Py	128	64	3894.22	5822.63	827.52	1101.62	145194.31	154181.03	1141.65
Pypy	128	56	817.35	6008.43	1859.92	1361.36	145194.15	161834.16	1300.67
Salsa20	128	64	952.19	1394.11	907.17	1275.82	6884.19	2215.93	1268.12
SOSEMANUK	128	64	247.93	6727.04	648.50	528.97	286119.29	20860.01	365.30

Table 2. Number of nJ for the stream ciphers using the testing framework

3.2 Memory requirements

We have performed some tests concerning the memory requirements of all the ciphers (using the same key and IV lengths) using only a speed option of optimization under `gcc` (the `-O2` one). The results concerning the code and the data memory sizes of all ciphers are given in table 3, together with the results for an empty code and always for the Copy in order to evaluate the minimum memory size induced by the benchmark environment.

Algo.	Empty	Copy	RC4	SNOW v2.0	AES-CTR	DRAGON	HC-256
Code size	4992	5040	6064	11152	17456	8512	14432
Data size	480	752	692	6836	13020	2740	692

Algo.	HC-128	LEX	Phelix	Py	Pypy	Salsa20	SOSEMANUK
Code size	12496	13072	9968	8736	8512	6560	21968
Data size	692	5852	724	35248	35248	724	3164

Table 3. Code Memory Requirements and Data Memory Requirements in bytes and in decimal notations

3.3 Analysis

First, we could see that most of stream ciphers (SNOW v2.0, SOSEMANUK, Dragon,...) stay more efficients on the dedicated architecture than the AES block cipher used in the CTR mode if we do not take into account the time required for the key setup and for the IV setup. Some of them such as Salsa20 have also a more efficient key and IV setup. Moreover, the code memory size and the data memory size of the AES-CTR is among biggest (except for Py and Pypy for the data memory size and for SOSEMANUK concerning the code memory size). So using stream ciphers in sensor network applications could be a good solution to achieve high encryption speed in high constraint environments.

We have then compare our benchmarks obtained on the dedicated platform and the results provided on the page of the eStream testing framework (see [6] for more details) that are given for traditional architectures such as Intel Pentium 4, Power PC,... (using several compilers). A really interesting point is that the most reliable stream ciphers on traditional platforms, Py and Pypy, do no longer work rapidly on our platform whereas SNOW v2.0, SOSEMANUK or HC-128 stay relatively fast.

This unusual property comes from the intrinsic structure of the ciphers Py and Pypy: they both uses two rolling tables of 256 bytes. The use of many `memcpy` to build at each iteration those tables explains the bad results obtained: the number of DC-misses is huge compared with the other studied stream ciphers. On our platform, and this is the case on other current sensor nodes, when the CPU accesses memory, it is stalled since it is not superscalar as many high performance

architectures are (the Pentium architecture for example). Our platform embed caches, but their size are about the quarters of level 1 caches of current high performance processors and it has no level 2 cache. Thus this cache architecture is not compatible with the memory access hunger of Py and Pypy algorithms, even if the results obtained for Pypy are rather better. The data are not preserved in data cache at each access and need to be refetched from main memory. This characteristic was underlined by table 3 with the data segment size of the implementation of these two algorithms. In summary, whereas Py and Pypy are really efficient on traditional high performance architectures, they could not be used without modifications in a such constrained environment. Moreover, the structure of Pypy is the same than the one of Py and the results obtained for Pypy stay reasonable so we think that because the Pypy performances are more reasonable, Py is the only algorithm that is totally incompatible with this cache geometry.

An other surprising result concerns the number of cycles required by the Keysetup of SOSEMANUK that is very huge compared with the results obtained on the other traditional platforms. This very bad result could be explained by the excessive code size (shown by table 3) that involves in our highly constrained architecture a performances reduction. When we are looking at the SOSEMANUK C source, we could notice that loops were unrolled for performance purposes. This code is in fact optimized for more powerful architecture. But this improvement induce the same behavior in the instruction cache than Py and PyPy in the data cache. Cache often misses and instruction are fetched from memory while the CPU is stalled. Then, if we want to improve the performances of SOSEMANUK, a good solution seems to be to reduce the code size to make it fit in the cache.

The other observed results here go in the same direction than the one presented in [6]: the most rapid algorithms stay approximatively the same (except for the particular case of Py and Pypy): HC-128, SOSEMANUK, SNOW v2.0, Phelix, RC4 and HC-256. We do not have modified the C code provided on the eStream web page but we think that it could be a solution to improve the results of some algorithms if we use a lower level programming language.

4 Conclusion

We have presented here some benchmarks performed on stream ciphers, the traditional ones (RC4, SNOW v2.0,...) and the candidates of the ECRYPT project. Some results could appear very strange but are in fact conditioned by the physical constraints of our platform.

Due to the ongoing state of the stream ciphers studied here, we do not have to give any recommendation about their use in such constraint environment but in the case of well-known and well-studied stream ciphers, we could notice that SNOW v2.0 is swift as well on traditional platforms as on the highly constrained environment.

As part of future work, we will benchmark the same ciphers on a MS430 16 bit micro-controller. Then, the comparison between the results obtained in [22] concerning the performances of block ciphers using several modes of operation and the stream ciphers presented here will be more pertinent. We also want to estimate the general loose of performance produced by the addition of a stream cipher in a real sensor communication environment.

References

1. 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 oriented software-oriented stream cipher. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.
2. Daniel J. Bernstein. Salsa20 specification. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.
3. Eli Biham and Jennifer Seberry. Py: A fast and secure stream cipher using rolling arrays. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.
4. Eli Biham and Jennifer Seberry. Pypy: Another version of py. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2006. `http://www.ecrypt.eu.org/stream/`.
5. Alex Biryukov. A new 128-bit key stream cipher lex. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2006. `http://www.ecrypt.eu.org/stream/`.
6. C. De Cannière. estream optimized code HOWTO. eSTREAM, ECRYPT Stream Cipher Project, 2005. `http://www.ecrypt.eu.org/stream/perf/`.
7. Joo Yeon Cho. An observation on dragon. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/002, 2007. `http://www.ecrypt.eu.org/stream`.
8. Don Coppersmith, Shai Halevi, and Charanjit S. Jutla. Cryptanalysis of stream ciphers with linear masking. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 515–532. Springer, 2002.
9. Paul Crowley. Improved cryptanalysis of py. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/010, 2006. `http://www.ecrypt.eu.org/stream`.
10. Ed Dawson, Kevin Chen, Matt Henricksen, William Millan, Leonie Simpson, Hoon-Jae Lee, and SangJae Moon. Dragon: A fast word based stream cipher. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.
11. P. Ekdahl and T. Johansson. SNOW - a new stream cipher. In *Proceedings of First NESSIE Workshop*, Heverlee, Belgique, 2000.
12. P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In *Selected Areas in Cryptography – SAC 2002*, volume 2295 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2002.
13. Hakan Englund and Alexander Maximov. Attack the dragon. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/062, 2005. `http://www.ecrypt.eu.org/stream`.
14. FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001. U.S. Department of Commerce/N.I.S.T.

15. S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas in Cryptography - SAC 2001*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2001.
16. N. Fournel, A. Fraboulet, and P. Feautrier. Booting and Porting Linux and uClinux on a new platform. Research Report RR2006-08, LIP - ENS Lyon, Feb 2006.
17. P. Hawkes and G. Rose. Guess-and-determine attacks on SNOW. In *Selected Areas in Cryptography - SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 37–46. Springer-Verlag, 2002.
18. R. Housley. Using advanced encryption standard (aes) counter mode with ipsec encapsulating security payload (esp). IETF, RFC 3686, 2004. `http://www.rfc-archive.org/getrfc.php?rfc=3686`.
19. Takanori Isobe, Toshihiro Ohigashi, Hidenori Kuwakado, and Masakatu Morii. How to break py and pypy by a chosen-iv attack. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/060, 2006. `http://www.ecrypt.eu.org/stream`.
20. Available online, `http://kcachegrind.sourceforge.net/`, nov 2006.
21. RSA laboratories. Rsa security response to weaknesses in key scheduling algorithm of rc4. available at `http://www.rsasecurity.com/rsalabs/node.asp?id=2009`, 2007.
22. Yee Wei Law, Jeroen Doumen, and Pieter Hartel. Survey and benchmark of block ciphers for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(1):65–93, 2006.
23. I. Manti and A. Shamir. A practical attack on broadcast rc4. In *Fast Software Encryption - FSE 2001*, volume 2335 of *Lecture Notes in Computer Science*, pages 152–164. Springer-Verlag, 2001.
24. Itsik Mantin. A practical attack on the fixed rc4 in the wep mode. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 395–411. Springer, 2005.
25. Itsik Mantin. Predicting and distinguishing attacks on rc4 keystream generator. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2005.
26. NESSIE. Nessie phase 1 : selection of primitives. `https://www.cryptonessie.org/`, 2001.
27. Network of Excellence in Cryptology ECRYPT. Call for stream cipher primitives. `http://www.ecrypt.eu.org/stream/`.
28. Souradyuti Paul, Bart Preneel, and Gautham Sekar. Distinguishing attacks on the stream cipher py. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2006.
29. Frédéric Pétrot and Pascal Gomez. Lightweight Implementation of the POSIX Threads API for an On-Chip MIPS Multiprocessor with VCI Interconnect. In *DATE 03 Embedded Software Forum*, pages 51–56, 2003.
30. R. Rivest. The RC4 encryption algorithm. RSA Data Security, 1992.
31. Available online, `http://www.skyeye.org/`http://www.skyeye.org/, nov 2006.
32. D. Watanabe, A. Biryukov, and C. De Cannire. A distinguishing attack of SNOW 2.0 with linear masking method. In *Selected Areas in Cryptography 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 222–233. Springer-Verlag, 2003.
33. Doug Whiting, Bruce Schneier, Stephan Lucks, and Frédéric Muller. Phelix - fast encryption and authentication in a single cryptographic primitive. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.
34. Hongjun Wu. Stream cipher hc-256. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2005. `http://www.ecrypt.eu.org/stream/`.

35. Hongjun Wu. Stream cipher hc-128. ECRYPT - Network of Excellence in Cryptology, Call for stream Cipher Primitives - Phase 2 2006. http://www.ecrypt.eu.org/stream/.
36. Hongjun Wu and Bart Preneel. Attacking the iv setup of stream cipher lex. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/059, 2005. http://www.ecrypt.eu.org/stream.
37. Hongjun Wu and Bart Preneel. Differential-linear attacks against the stream cipher phelix. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/056, 2006. http://www.ecrypt.eu.org/stream.
38. Hongjun Wu and Bart Preneel. Key recovery attack on py and pypy with chosen ivs. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/052, 2006. http://www.ecrypt.eu.org/stream.