Design and efficient implementation of a chaos-based stream cipher

Mohammed Abu Taha* and Safwan El Assad

Institut d'Electronique et de Télécommunications de Rennes (IETR),
Université de Nantes, France
Email: mohammad.abu-taha@etu.univ-nantes.fr
Email: safwan.el-assad@univ-nantes.fr
*Corresponding author

Audrey Queudet

Institut de Recherche en Communications et
Cybernétique de Nantes (IRCCyN),
Universit´e de Nantes, France
Email: audrey.queudet@univ-nantes.fr

Olivier Deforges

Institut d'Electronique et de Télécommunications de Rennes,
INSA de Rennes, France
Email: olivier.deforges@insa-rennes.fr

Abstract: We designed and implemented a stream cipher cryptosystem based on an efficient chaotic generator of finite computing precision (N = 32). The proposed structure of the chaotic generator is formed by a Key-setup, an IV-setup, a non-volatile memory, an output and an internal state function. The cryptographic complexity mainly lies in the internal state containing two recursive filters, with one, two or three delays. Each recursive filter includes a perturbation technique using a linear feedback shift register. The first recursive filter includes a discrete skew tent map, and the second one includes a discrete piecewise linear chaotic map. The chaotic generator is implemented in sequential and parallel versions using Pthread library in a secure manner. The proposed Stream ciphers have very good performance in terms of security and execution time. Experimental results highlight the robustness of the proposed system against known cryptographic and statistical attacks.

Keywords: stream cipher; chaotic generator; chaotic multiplexing; parallel computing.

Reference to this paper should be made as follows: Abu Taha, M., El Assad, S., Queudet, A. and Deforges, O. (2017) 'Design and efficient implementation of a chaos-based stream cipher', *Int. J. Internet Technology and Secured Transactions*, Vol. 7, No. 2, pp.89–114.

Biographical notes: Mohammed Abu Taha received his MS in Informatics from Palestine Polytechnic University, Hebron-Palestine. He is currently pursuing his PhD from Nantes University, France. His research interests include security of image and video, Linux-based real-time applications and embedded systems, and parallel programming.

Safwan El Assad joins the University of Nantes, France in September 1987, where he is currently an Associate Professor. Since 2005, his main researches are in chaos-based crypto and crypto-compression systems for secure transmitted and stocked data.

Audrey Queudet graduated in Computer Engineering at Polytechnic School of the University of Nantes (France). She is an Associate Professor at the University of Nantes. Her research interests include real-time scheduling theory, quality of service guarantees for soft real-time systems, and Linux-based real-time operating systems and applications.

Olivier Deforges received his PhD in Image Processing in 1995. He is a Professor with the National Institute of Applied Sciences (INSA) of Rennes since 2005. His principal research interests are image and video lossy and lossless compression, image understanding, fast prototyping, and parallel architectures.

1 Introduction

Cryptography was used in the past to keep military information and diplomatic correspondence secure and to protect national security. In recent times, the range of cryptography applications has been widely expanded, following the development of new communication means. Cryptography is used to ensure that the contents of a message are confidentially transmitted and cannot be altered. Chaos is one interesting field of research dealing with nonlinear, deterministic, and dynamic systems. It is applied to many different domains such as physics, robotics, biology, finance and encryption. The most important chaos properties are the high dependency on initial conditions and parameter variation, ergodicity and the random-like behaviour. These properties entice researchers to develop chaotic secure communication systems (Kocarev, 2001; El Assad and Farajallah, 2016; Farajallah et al., 2016; Setti et al., 2005; Cimatti et al., 2007; Abu Taha et al. 2015; Arlicot, 2014; Caragata et al., 2010; Chetto et al., 2014). Under certain conditions, chaos can be generated by any nonlinear dynamic system (Smale, 1967). For public channels including network communication and for computer communication, most data transactions (valuable information) need to be protected from malicious attacks and threats (Li and Lee, 2016; Masoumi et al., 2016; Jo and Koh, 2016). A block symmetric cipher is one of the classical encryption technique widely used in the literature. The Advanced Encryption Standard (AES) is one of the most famous symmetric encryption for block ciphers. The stream cipher is used to secure useful information that must be transmitted continuously over the network communication for example. Generally stream ciphers are more efficient than block ciphers in two situations:

1 in software applications requiring a very high encryption or decryption rate

2 in hardware applications where physical resources(e.g., chip area, power, etc) are restricted.

Handling a stream cipher encryption with block ciphers is possible by using counter and output feedback modes (CTR, OFB). Because the AES is very secure and widely adopted, its two modes, namely CTR and OFB are used as stream ciphers. However, to

benefit from both advantages of stream ciphers compared to block ciphers, several stream cipher designs such as RC4 and eSTREAM algorithms have been produced. RC4 is one of the widely known stream ciphers and a hardware implementation was performed in an efficient way by Gupta et al. (2013). However, RC4 is now broken. The eSTREAM project was a multi-year effort, running from 2004 to 2008, to promote the design of efficient and compact stream ciphers suitable for the widespread adoption of Estream (eSTREAM, 2008). Nevertheless, until now most of the eSTREAM ciphers are still not definitely secure (Manifavas et al., 2015). Chaos-based stream ciphers are used to enhance the security issue (Machicao et al., 2012).

In this paper, we propose a new chaos-based stream cipher. The proposed system is based on an efficient chaotic generator using two chaotic recursive filters, a technique of disturbance and chaotic multiplexing. The remainder of the paper is structured as follows. The next section reviews the related work and Section 3 recalls the main technique used in parallel programming. The structure of the proposed stream cipher is described in Section 4. We detail the description of the proposed chaotic generator in Section 4.1 and Section 4.2 provides its parallel implementation. Next Section 4.3 gives the computation performance of the generator. In Section 5, we set out the performance of the stream cipher in terms of encryption speed and security using known cryptographic and statistical attacks. Finally, Section 6 concludes our contribution and outlines some directions for future work.

2 Related work

In the following paper we recall the main related works in standard and chaos-based stream ciphers.

2.1 AES-CTR and eSTREAM software

AES-CTR mode

Counter mode, a standard introduced by Diffie and Hellman in 1979 is one of the best known modes used for stream ciphers. Counter mode switches a block cipher into a stream one. It generates the next keystream block by encrypting successive values of a counter. After each block encryption, the counter must be different and this can be done simply by incrementation of the counter by some constant, typically one. CTR mode has significant efficiency advantages over the standard encryption modes without weakening the security. In particular its tight security has been proven. On the other hand most of the perceived disadvantages of CTR mode are not valid criticisms, but rather caused by a lack of knowledge (Lipmaa et al., 2000).

Rabbit

Rabbit is a stream cipher algorithm. Its rose/developed as a fast software encryption method in 2004. It is one of the most effective algorithm developed in the eSTREAM project. Rabbit is directed to be used in both software and hardware applications. The Rabbit Algorithm takes a 128-bit key and a 64-bit IV vector as input. At each iteration it, generates a 128-bit output. The output is pseudo-random in its nature. The heart of this

cipher consists of 513 internal state bits. Clearly the output generated in each iteration is some combination of these state-bits. The 513 bits are divided into eight 32-bit state variables, eight 32-bit counters and one counter carry bit. The state functions which update these state variables are nonlinear and thus build the basis of the security provided by this cipher (Boesgaard et al., 2005; eSTREAM, 2008). The designers provided the security analysis considering several possible attacks: algebraic, correlation, and statistical attacks. They conclude that no huge weakness of Rabbit has been found. However in 2009, Kircanski and Youssef in their paper provide a differential fault analysis attack on Rabbit algorithm. The fault model in which they analyse the cipher is the one in which the attacker is assumed to be able to fault a random bit of the internal state. The attack requires around 128–256 faults, a precomputed table of size around 241.6 bytes, this technique enables to recover the complete internal state of Rabbit in about 238 steps.

Salsa20/r

Salsa20/r is one of the eSTREAM finalist algorithms for software implementation, where $r = 8, 12, 20$ represents the number of iterations of the round function. The algorithm is constructed on a pseudo-random function based on a 32-bit addition, bitwise XOR and rotation operations, which maps a 256-bit key, a 64-bit nonce (IV initial vector), and a 64-bit stream position to a 512-bit output (Bernstein, 2008; eSTREAM, 2008). The Salsa20/8 version is very fast but not secure enough. Its weakness comes from a differential cryptanalysis performed by Tsunoo et al. (2007). Salsa20/12 and Salsa20/20 algorithms seem to be secure so far, because no better attack than the brute-force attack has been reported.

HC-128 and HC-256

HC-128 is an efficient software stream cipher, which consists of two secret tables, each one with 512 32-bit elements. At each step they update one element from one of the two tables using a nonlinear feedback function. All the elements of the two tables are updated every 1,024 steps. At each step, one 32-bit output is generated from the nonlinear output function. HC-256 is a new version that differs from HC-128 by the size of secret tables which is 1,024 32-bit elements instead of 512 32-bit ones. All the elements of the two tables are updated every 2,048 steps. At each step, HC-256 produces one 32-bit output (Wu, 2008, 2004; eSTREAM, 2008). However, in 2010, Kircanski and Youssef provide in a differential fault analysis attack on HC-128 their paper. The attack is based on the fact that, some of the inner state words of HC-128 may be exploited several times without being updated. Consequently, the complete internal state is recovered using about 7968 faults.

SOSEMANUK

SOSEMANUK is a software stream cipher that has a key length ranging from 128 to 256 bits. It takes an initial value IV vector of 128 bits. SOSEMANUK has two main components: a linear feedback shift register (LFSR) and a finite state machine (FSM). The LFSR operates on 32-bit words and at every clock a new 32-bit word is computed. The FSM has two 32-bit memory registers: at each step the FSM takes an input word

from the LFSR, updates the memory registers and produces a 32-bit output (Berbain et al., 2008; eSTREAM, 2008). In 2011, Salehani et al. (2011) made a differential attack on SOSEMANUK. The attack needed around 6144 faults to recover the secret inner state of the cipher.

2.2 Chaos-based stream cipher

Abderrahim et al. (2014) in their paper propose a chaos-based stream cipher based on symbolic dynamic description and synchronisation. Their main contribution concerns a pseudo-random number generator (PRNG) based on an appropriate mixture of perturbed chaotic maps. The synchronisation of the emitter/receiver is performed by a symbolic dynamic-based method. One of the characteristics of their proposed stream cipher is that the chaotic symbolic dynamic sequences are easy to produce. The obtained bit rate, with an Intel Core i7 processor clocked at 3.5 GHz, and 8G of RAM is 10 Mbps. Lu et al. (2004), proposed a one-way-coupled chaotic map lattice for cryptography of a self-synchronising stream cipher. The system performs an analytical computation into real numbers, and incorporates some algebraic operations on integer numbers. The encryption/decryption operation is done in parallel using multiple chaotic maps. The authors claim that the system has a good security level, and good reliability against strong channel noise. They provide an encryption speed (around 914 Mbps on a 2 GHz CPU). In 2007, Li et al. published a stream cipher also based on a spatiotemporal chaotic system as done previously in Lu et al. (2004). The chaotic system uses coupled logistic maps, and simple algebraic computations. The system produces parallel keystreams for encrypting plaintexts via bitwise XOR. Security analysis is performed to prove the robustness of the system. The encryption speed is 700 Mbits in a computer with a 1.8 GHz CPU and 1.5 GB RAM. The eSTREAM project ciphers have better performance in time than the three chaos-based stream ciphers. In the following sections we will describe our chaos-based stream cipher in sequential and parallel implementation.

3 Parallel programming techniques

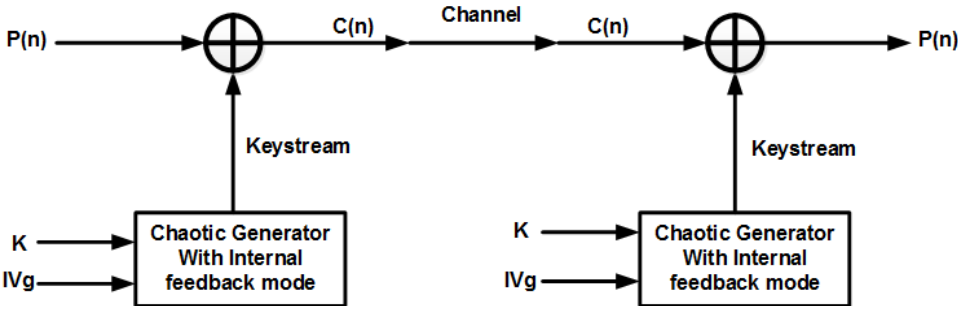
As processors' speeds no longer significantly increase, multicore systems have become more popular. Thus to benefit from these systems, programmers have turned to parallel programming. Therefore, programmers have to deal more and more with parallel programming. Parallelism is achieved thanks to multiple processes running at the same time on multiple processors (Rani, 2011). It explicitly breaks the task down into small units of execution, where each unit can be executed in parallel on a single processor. In this way multiple parts of the same task can run in parallel (Sinnen, 2007; Lozi et al., 2016). Parallel programming can be implemented using several different software interfaces, or parallel programming models. The programming model used in any application depends on the underlying hardware architecture of the system on which the application is expected to run: *shared memory* architecture or *distributed memory* environment. In *shared-memory* multiprocessor architectures, threads can be used to implement parallelism. Threads are lightweight processes, existing within a single operating system process. Threads share the same memory address space and state information of the process that contains them. Parallel programming can be implemented for shared memory systems using *automatic parallelisation* (Banerjee et al., 1993),

POSIX threads (Butenhof, 1997) and *Solaris threads* (Butenhof, 1997), or *OpenMP* (Dagum and Enon, 1998). Among distributed memory programming models, the message passing interface (MPI) model (Gropp et al., 1996) is commonly used to parallelise applications. MPI is a very explicit programming model. The programmer implements the distribution of the tasks, communication between them, and decides how the work is allocated between the various threads. With the emergence of multi-core systems, hybrid programming models have also been developed. Within a single node, fast communication through shared memory can be exploited, and a networking protocol can be used to communicate across the nodes. Programs can then take advantage of both shared memory and distributed memory. In our parallel implementation we used *POSIX threads*.

4 Description of the proposed chaos-based stream cipher

In this section we present a synchronous stream cipher based on a novel chaotic generator with its two implemented versions (sequential and parallel). In sequential implementation, each generator call produces a 32-bit sample that is immediately converted into 4 bytes and stored in a buffer, before being Xored with 4 bytes from the plaintext to obtain 4 ciphered bytes and so on. In the parallel implementation, each generator call produces four 32-bit samples that are immediately converted into 16 bytes stored in a buffer and then Xored with 16 bytes from the plaintext. Here, a question of synchronisation between generated samples arises after each generator call. More details about this question are given in Section 4.2. For a given plaintext data, the generator produces the necessary keystreams to obtain ciphering data. In Figure 1 the general structure of stream cipher encryption and decryption processes are shown.

Figure 1 Stream cipher encryption/decryption structure



As with any encryption system, the secret key K and the initial IV vector must be shared between the sender and the receiver. The key must be kept secret while the IV vector is not necessarily kept secret but must be a nonce. The common method to share the secret K between the two parties is a symmetric key distribution based on either symmetric encryption using a key distribution centre (KDC) or asymmetric encryption using the RSA (Rivest, Adi Shamir and Leonard Adleman) algorithm (Stallings, 2006). The IVg is changed every new session as a key session.

4.1 Description of the proposed chaotic generator

The architecture of the proposed chaotic generator is composed of several black-boxes as presented in Figure 2. The detailed description of the internal state and the output function is given in Figure 3. The secret key `K`, the initial vector `Nonce IVg` and parameters are the inputs of the chaotic generator. From these inputs, the `IV-setup` computes another three IVs values and the `Key-setup`, in case of parallel implementation, creates another three keys. Then, four IVs and four keys will be used by four threads in the system. Since chaos is sensitive to any small changes in the secret key, the creation of each new key in the `Key-setup` entity is achieved by the circular shift rotation of the three bit value of $K1\_s$,$K1\_p$ parameters (see Figure 3). Moreover, the creation of each new IV in the `IV-setup` entity is achieved by the circular shift rotation of the three-bit value of U_s, U_p. Before the execution of the program is completed, a new IV value is generated and stored in the non-volatile memory box. The generation of this new value comes from `/dev/urandom` Linux PRNG (Gutterman et al., 2006). The internal state, which contains the main cryptographic complexity of the system, is formed by two recursive filters of order three. The first recursive cell contains a discrete Skew tent map and the second one contains a discrete piecewise linear chaotic map (PWLCM). These maps are used as nonlinear functions. We give below the outputs of the recursive cell containing the Skew tent map and of the recursive cell containing the PWLC map respectively. Hence the output equation of the recursive cell Skew tent map is:

$$X\_s = STmap\{F1[n-1],\, P1\} \oplus Q1 \tag{1}$$

with

$$F1[n-1] = \mathrm{mod}\left[U\_s + \sum_{i=1}^{3} [K(i)\_s \times X(n-i)\_s],\, 2^N \right] \tag{2}$$

And the output equation of the recursive cell PWLC is:

$$X\_p = PWLCmap\{F2[n-1],\, P2\} \oplus Q2 \tag{3}$$

with

$$F2[n-1] = \mathrm{mod}\left[U\_p + \sum_{i=1}^{3} [K(i)\_p \times X(n-i)\_p],\, 2^N \right] \tag{4}$$

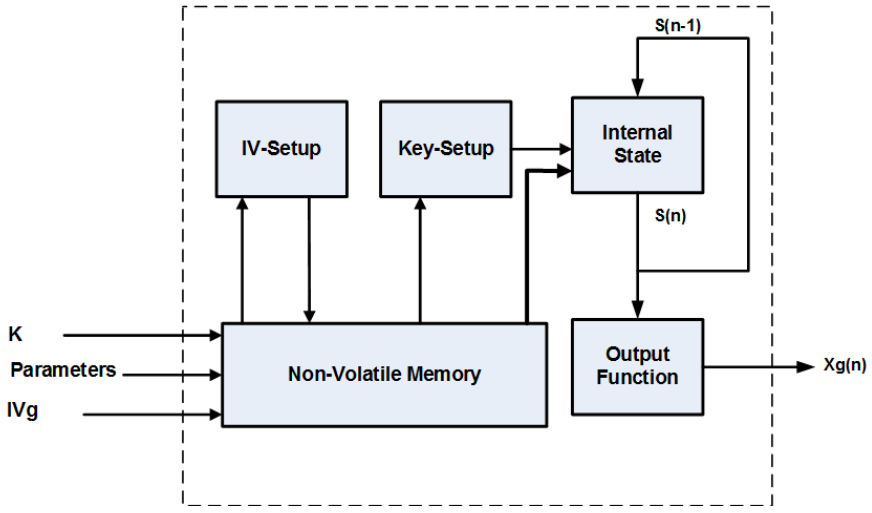
In the equations above, $P1$ and $P2$ are control parameters in the range $[1,\, 2^N - 1]$ and $[1,\, 2^{N-1} - 1]$ respectively. $Q1$ and $Q2$ are perturbing signals produced by the LFSRs. $K1\_s$, $K2\_s$, $K3\_s$, $K1\_p$, $K2\_p$, $K3\_p$ are the coefficients of the recursive cells in the interval $[1;\, 2^N - 1\ ]$. U_s and U_p, each of 32 bits, represent IVg of 64 bits. The equations of the *discrete* skew tent and *discrete* PWLCM maps are respectively given by (Masuda and Aihara, 2002; Lian et al., 2007; El Assad, 2012; Desnos et al., 2014): *discrete* skew tent map:

$$X_s[n] = \begin{cases} \left\lceil 2^N \times \dfrac{X_s[n-1]}{P1} \right\rceil & \text{if } 0 < X_s[n-1] < P1 \\ 2^N - 1 & \text{if } X_s[n-1] = P1 \\ \left\lceil 2^N \times \dfrac{2^N - X_s[n-1]}{2^N - P1} \right\rceil & \text{if } P1 < X_s[n-1] < 2^N \end{cases} \tag{5}$$

Discrete PWLCM map:

$$X_p[n] = \begin{cases} \left\lceil 2^N \times \dfrac{X_p[n-1]}{P2} \right\rceil & \text{if } 0 < X_p[n-1] \le P2 \\ \left\lceil 2^N \times \dfrac{X_p[n-1] - P2}{2^{N-1} - P2} \right\rceil & \text{if } P2 < X_p[n-1] \le 2^{N-1} \\ \left\lceil 2^N \times \dfrac{2^N - P2 - X_p[n-1]}{2^{N-1} - P2} \right\rceil & \text{if } 2^{N-1} < X_p[n-1] \le 2^N - P2 \\ \left\lceil 2^N \times \dfrac{2^N - X_p[n-1]}{P2} \right\rceil & \text{if } 2^N - P2 < X_p[n-1] \le 2^N - 1 \\ 2^N - 1 - P2 & \text{otherwise} \end{cases} \tag{6}$$

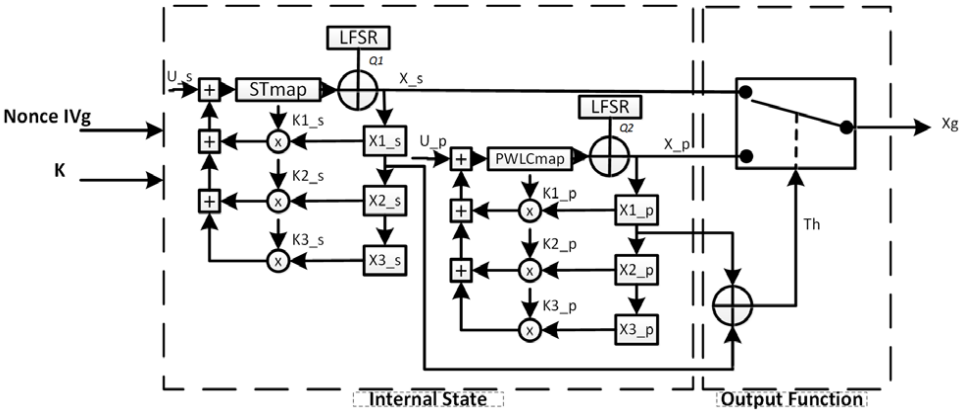
Figure 2 Architecture of the proposed generator with internal feedback mode (see online version for colours)



The values produced $X_s[n]$, $X_p[n]$ by the recursive cells in `the internal state` are entered to `the output function`. Then, the output sequence $Xg(n)$ is obtained using a chaotic multiplexing controlled by the chaotic sequence $X1_s(n-1) \oplus X1_p(n-1)$ and by a threshold $Th = 2^{N-1}$, as shown in Figure 3. The output sequence is defined as follows:

$$Xg(n) = \begin{cases} Xs[n], & \text{if } 0 < X\_s[n-1] \oplus X\_p[n-1] <= Th \\ Xp[n], & \text{otherwise} \end{cases} \tag{7}$$

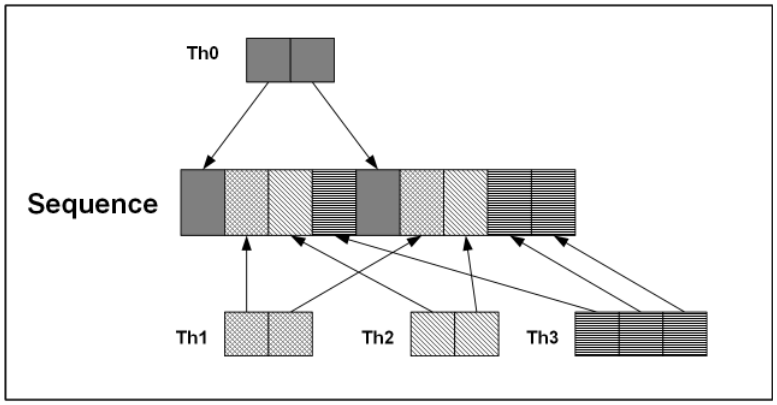
Figure 3 Detailed description of the internal state and the output function



4.2 Parallel implementation of the chaotic generator using Pthread

Usually a multi thread process launches several threads that run concurrently. In our implementation, we parallelised the sequential version of our chaotic generator using the standard API used for implementing multithreaded applications, namely *POSIX Threads* or *pthread* (Pacheco, 2011). *pthread* is a library of functions that programmers can use to implement parallel programs. Unlike MPI, *pthread* is used to implement shared-memory parallelism. It is not a programming language (such as C or Java). It is a library that can be linked with C programs. The source code is compiled with *gcc* and using the *-lpthread* option. In our multithreaded approach, data sequences are partitioned among several threads. Threads execute the same instructions on different data sets. The number of samples to be processed and the starting point of the samples' subset data are different for each thread. The different threads are created and launched via a call to *pthread create()*. In our case, we create a number of threads equals to the number of cores chosen in our system. The function *pthread create()* takes the thread as parameter. Each thread will call the *computation* function. This function ensures the generation of the samples and the conversion to bytes. Then the computed sequences from threads will be stored in a buffer in a systematic manner to gain a maximum performance. Each sequence from each thread is then stored consecutively as illustrated in Figure 4. In the *main()* function, we wait for the termination of all threads by calling the *pthread join()* function. To describe the decomposition of the sequences among the threads, we give the following example: consider that 4 cores are available on the platform and that the sequence length is *seq length* = 3,125,000 samples. Four threads will then be created. The first thread computes samples from index *i*min = 0 * 3,125,000/4 = 0 to index *i*max =(0 + 1) * (3,125,000/4) − 1 = 781249. The second thread computes samples from index *i*min = 1 * 3,125,000/4 = 781,250 to index *i*max = (1 + 1) * (3,125,000/4) − 1 = 1,562,499 and so on until the last thread that will compute the rest of samples. The remainder of samples that resulted from the division of the number of sequences to the number of threads, if it exist, will also be computed by the last thread. Samples from each thread are stored in a shared result array, each thread filling specific index values.

Figure 4 Storing of samples that generated by different threads



4.3 Computing performance of the chaotic generator

To evaluate the computing performance of the proposed chaotic generator we performed some experiments using a two 32-bit multi-core Intel Core (TM) i5 processors running at 2.60 GHz with 16 G of memory. This hardware platform was used on top of an Ubuntu 14.04 Trusty Linux distribution. Here after, for different sizes of data bytes, we give the average generation time in micro second GT (μs), the average bit rate en Mega bit par second BR(Mbit/s), and the average of the required number of cycles to generate one byte, NCpB(Cycles/B). The average is determined by using 100 different secret keys for each data size. For parallel implementation we choose 4 threads in parallel running on a four-cores platform. The results obtained for GT (μs), BR(Mbit/s) and NCpB(Cycles/B) are given in Tables 1, 3 and 4 and are depicted in Figures 5, 6 and 7 for sequential and parallel implementation. The number of cycles required to generate one byte NCpB is given by:

$$\mathrm{NCpB} = \frac{\mathrm{CPU\ Speed}_{(\mathrm{Hertz})}}{\mathrm{Db}_{(\mathrm{Byte/s})}} \tag{8}$$

Table 1 Generation time for sequential and parallel generators

Data (bytes)	*GT/Seq (μs)*	*GT/Parl (μs)*
64	6	705
128	8	726
256	11	743
512	19	753
1,024	32	763
2,048	57	801
4,096	109	810
16,384	332	835
32,768	520	847

Table 1 Generation time for sequential and parallel generators (continued)

Data (bytes)	*GT/Seq (µs)*	*GT/Parl (µs)*
64	6	705
65,536	712	764
125,000	1282	1325
196,608	1830	1869
393,216	2902	2436
786,432	5502	4835
3,145,728	21723	19539
12,582,912	85009	49154

Table 2 NCpB performance of some PRNG

PRNG	*NCpB (cycles/B)*
Wang et al. (2016)	160
Akhshani et al. (2014)	45
Jallouli et al. (2016)	24.68
Proposed algorithm	17.3

Table 3 Bit rate for sequential and parallel generators

Data (bytes)	*BR/Seq (Mbit/s)*	*BR/Parl (Mbit/s)*
64	85.33	0.73
128	128	1.41
256	186.18	2.76
512	215.58	5.44
1,024	256	10.74
2,048	287.44	20.45
4,096	300.62	40.45
16,384	394.8	156.97
32,768	504.12	309.5
65,536	736.36	686.24
125,000	780.03	754.72
196,608	859.49	841.55
393,216	1,083.99	1,291.35
786,432	1,143.49	1,301.23
3,145,728	1,158.49	1,287.98
12,582,912	1,184.15	2,047.92

Table 4 NCpB for sequential and parallel generators

Data (bytes)	*NCpB-S (cycles/B)*	*NCpB-P (cycles/B)*
64	232.5	27,173.2
128	155	14,068.4
256	106.5	7,187.1
512	92	3,646.4
1,024	77.5	1,847
2,048	69	970
4,096	66	490.4
16,384	50.2	126.4
32,768	39.3	64.1
65,536	26.9	28.9
125,000	25.4	26.3
196,608	23.1	23.6
393,216	18.3	15.4
786,432	17.3	15.2
3,145,728	17.1	15.4
12,582,912	16.8	9.7

As we can see from these results, the parallel implementation is only better for data size equal to or bigger than 393,216 bytes. This is due to the overhead time caused by the synchronisation between threads. In Table 2 we compare our obtained results in terms of NCpB with some known chaos-based generators, for data size equal to 786,432 bytes that correspond to a image size of 512 * 512 * 3. As we can see, the obtained performance is good.

Figure 5 Generation time for parallel and sequential generators (see online version for colours)

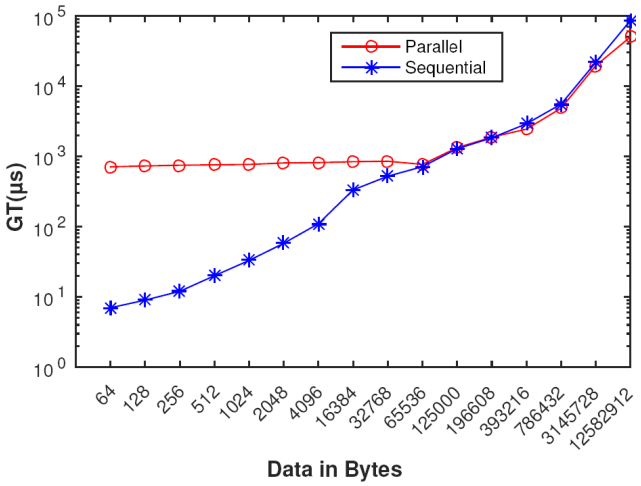


Figure 6 Bit Rate for parallel and sequential generators (see online version for colours)

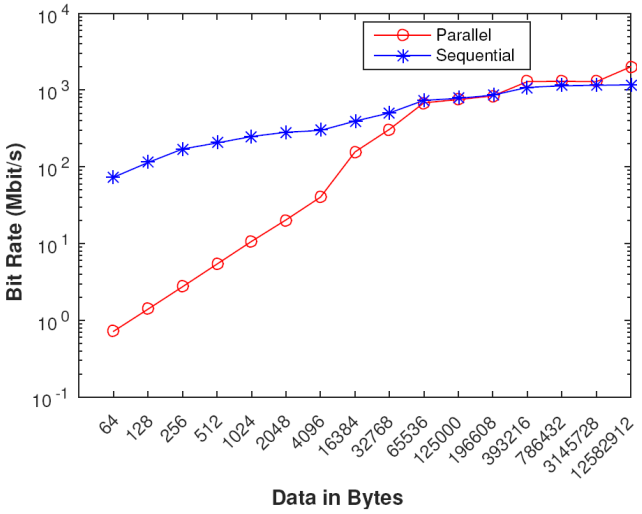
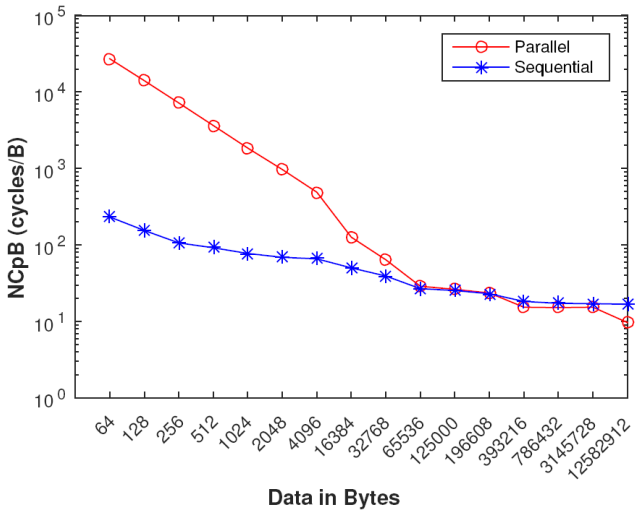


Figure 7 NCpB for parallel and sequential generators (see online version for colours)



5 Encryption speed and security analysis of the proposed stream cipher

5.1 Time performance

The computation performance is determined by: the average encryption time Enc T(μs), the average encryption throughput ET(Mbit/s) defined in equation (9), and the average number of cycles to encrypt one byte NCpB(Cycles/B) defined previously in equation (8).

$$\mathrm{ET} = \frac{\mathrm{Image}_{\mathrm{Size}}(\mathrm{Mbit})}{\mathrm{Encryption}_{\mathrm{Time}}(s)} \tag{9}$$

Table 5 Performance results of proposed sequential stream cipher with different data bytes

Data in bytes	*Enc-T(μs) Seq/Parl*	*ET (Mbit/s) Seq/Parl*	*NCpB (cycles/B) Seq/Parl*
512	21/778	213.01/5.31	92.9/3,650.7
1,024	33/792	259.1/11.1	78.2/1,889
2,048	60/806	286.5/19.9	70.2/973
4,096	116/822	299.3/39.3	67.0/491.3
49,152	659/1,619	569.0/231.6	34.8/85.6
196,608	2,455/2,419	610.9/620.0	31.9/31.2
786,432	9,088/8,099	660.2/740.8	30.0/26.7
3,145,728	35,560/24,190	674.9/978.8	29.3/20.2
12,582,912	121,899/88,597	787.5/1,083.5	25.1/18.3
50,331,648	398,089/319,785	964.6/1,200.8	20.5/16.5

We report in Table 5 and in Figures 8, 9, 10 the obtained results of the computation performance for sequential and parallel implementation of the proposed stream cipher. The decryption time is approximatively equal to the encryption time.

For big data size, from 196,608 bytes upwards, the parallel implementation is better than the sequential one and on average the NCpB of the stream cipher takes approximatively eight cycles more compared to the NCpB of the chaotic generator.

Figure 8 Encryption time for parallel and sequential cryptosystem (see online version for colours)

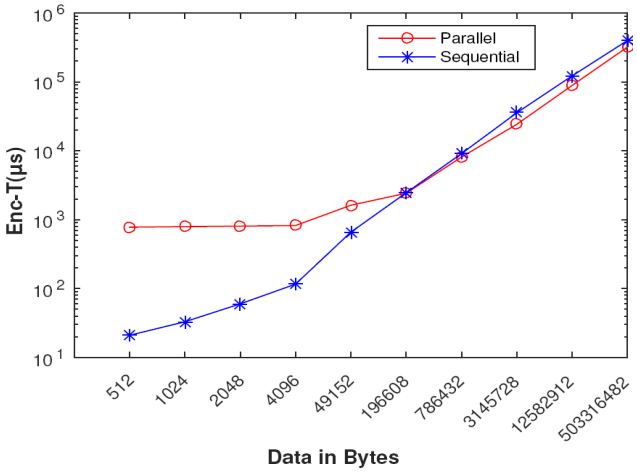


Figure 9 Encryption throughput for parallel and sequential cryptosystem (see online version for colours)

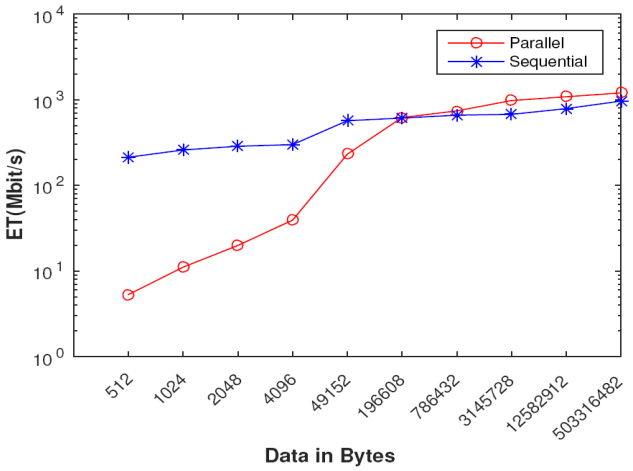
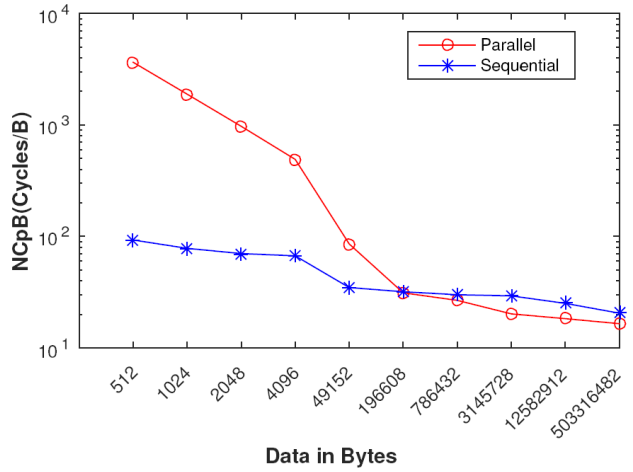


Figure 10 NCpB for parallel and sequential cryptosystem (see online version for colours)



In Table 6, we report a comparison of time computation for the proposed algorithm (for different data size images of Lena) with three chaos-based algorithms and the most Known stream ciphers of eStream project (Maxime, 2016). For big data, the proposed algorithm has better results than Abderrahim et al. (2014) and Lu et al. (2004). We also observed that the time computation of eStream's algorithms is better than the proposed system until we reach the big data size, for which, our system will be faster. For very big data size (201,326,592) such as videos, the obtained NCpB is around 9. In addition, the proposed chaotic system has a strong nonlinearity compared to the other systems thus, its robustness against cryptographic attacks is higher.

Table 6 Performance results comparison of some stream ciphers

Stream cipher-Alg	*Image size (B)*	*Enc-time (μs)*	*ET (Mbit/s)*	*NCpB (cycles/B)*
Abderrahim et al.	-	-	10	2,800
Hauping et al.	-	-	914	17
Ping et al.	-	-	700	20
Rappit	256 × 256 × 3	811.3	1,848.8	9.5
	512 × 512 × 3	3,256	1,842.6	9.5
	1,024 × 1,024 × 3	12,950	1,853.9	9.5
HC-128	256 × 256 × 3	1,221	1,228.1	14.4
	512 × 512 × 3	4,895	1,225.6	14.4
	1,024 × 1,024 × 3	19,647	1,221.5	14.4
Salsa20/12	256 × 256 × 3	836.4	1,793.4	9.8
	512 × 512 × 3	3,389	1,770	9.9
	1,024 × 1,024 × 3	13,483	1,779.9	9.9
SOSEMANUK	256 × 256 × 3	880.3	1,704	10.3
	512 × 512 × 3	3,570	1,680	10.5
	1,024 × 1,024 × 3	14,134	1,698	10.4
AES-CTR	-	-	-	21.2
Proposed chaos stream cipher (Seq)	256 × 256 × 3	2,455	610.9	31.9
	512 × 512 × 3	9,088	660.2	30.0
	1,024 × 1,024 × 3	35,560	674.9	29.3
Proposed chaos stream cipher (Parl)	256 × 256 × 3	2,419	620	31.2
	512 × 512 × 3	8,099	740.8	26.7
	1,024 × 1,024 × 3	24,190	978.8	20.2
	201,326,592	1,200,178	1,881	8.8

5.2 Security analysis

In this section we evaluated the software security implementation and the security of the proposed chaotic system against cryptanalytic and statistic attacks.

5.2.1 Software security implementation

Software security analysis is another crucial factor to ensure the quality of the source code and to restrict all security threats. Because it is still possible to read data out of memory even if the application no longer has pointers to it, it is necessary to incorporate data security within the source code. In cryptographic applications, sensitive information (e.g., secret keys) must be kept in memory for the minimum amount of time possible and should be written over/deleted, not just released, when no longer needed. One first step consists in erasing such sensitive data from memory once it is no longer needed in order to prevent any security attacks. The idea is to zero-fill buffers which contained sensitive information. In practice, we used the following functions to decontaminate (i.e., zero) a buffer and guarantee that the compiler will not optimise it away: the `secure_memzero()`

function depends on a function pointer `memset ptr` that itself points to the `memset() function`. It uses the key and the key size and will put zero value on the allocated memory related to the key by call `memset()`. The function `memset()` is invoked to write a specific value in a buffer that was allocated previously. We used this function to write a zero value in the buffer. While Some compilers optimise away the call to `memset()` function. To overcomes this, we declared `memset ptr` as a *volatile* pointer. Since a *volatile* pointer can be manipulated outside the scope of the application, the code is not optimised by the compiler, thus keeping the program unchanged. Furthermore, the data in main memory may leak to the disk through virtual memory, thus representing another source of the most serious leaks (leaks to physical mediums). One solution, which is sufficient to include, is to deactivate the swap space altogether, thus preventing data from being written to the page file by locking it in memory. In our code, we used the `mlock()` function that locks pages in the address range starting at the address and continuing for byte lengths. All pages that contain a part of the specified address range are secured to be resident in the main memory when the call returns successfully. Then, the pages are guaranteed to stay in the main memory until later unlocked.

In order to guarantee the validity of our solution, we carried out a security code review using several static and dynamic techniques: Clang, Gdb, Valgrind, DRD, Callgrind and leak-analysis tools. Results match up well with the security level requested by our chaos-based stream cipher (Taha et al., 2016).

5.2.2 Cryptanalytic attacks

The proposed system has the ability to resist common attacks such as ciphertext only (Siegenthaler, 1985), chosen plaintext attack, brute force attack, and key sensitivity attack. Indeed, encrypting an image several times using the same secrete key, produces totally different ciphered images. This is due to the IV-setup block.

Key space

The size of the secret key, formed by all the initial conditions and by all the parameters of the system, varies from 299 bits, with delay = 1, to 555 bits, with delay = 3. This means that the brute force attack is impracticable.

Key security and sensitivity attack

From the generated sequences it is impossible to find the secret key and this is because of the structure of the chaotic generator which in addition includes a chaotic switching. The knowledge of part of the secret key is not very useful for an attacker because of the intrinsic property of chaotic signal, which is extremely sensitive to the secret key. Besides, we computed the average hamming distance (HD) (of 100 secret keys) of two Keystreams generated each time with two secret keys that differ only by one bit and the result obtained is equal to 0.499993, therefore very close to 50%. In conclusion, the produced keystreams are highly secure. A cryptosystem must be sensitive to one bit change per key used. This property is important in order to resist many attacks (Lian et al., 2005). To test the key sensitivity of the proposed chaos stream cipher, we encrypted 'Lena' image 100 times using 100 secrete keys that differ only by the LSB bit. Then we computed the following parameters: the number of pixel change rate (NPCR),

the unified average changing intensity (UACI) and the HD. The parameters (NPCR, UACI) are necessary but not sufficient to ensure that the proposed cryptosystem is resistant against the key sensitivity attack. For this reason, we add the HD measurement (Mar and Latt, 2008).

The NPCR and UACI, introduced by Eli Biham and Adi Shamir (Biham and Shamir, 1991) are given by the following equations:

$$NPCR = \frac{1}{L \times C \times P} \times \sum_{p=1}^{P} \sum_{i=1}^{L} \sum_{j=1}^{C} D(i, j, p) \times 100\% \tag{10}$$

where

$$D(i, j, p) = \begin{cases} 0, & \text{if } C_1(i, j, p) = C_2(i, j, p) \\ 1, & \text{if } C_1(i, j, p) \neq C_2(i, j, p) \end{cases} \tag{11}$$

$$UACI = \frac{1}{L \times C \times P \times 255} \times \sum_{p=1}^{P} \sum_{i=1}^{L} \sum_{j=1}^{C} \left| C_1(i, j, p) - C_2(i, j, p) \right| \times 100\% \tag{12}$$

In the previous equations, i, j and p are the row, column, and plane indexes of the image, respectively. L, C and P are the length, width, and plane sizes of the image respectively. The optimal NPCR and UACI values are 99.61% and 33.46% respectively (Wu et al., 2011).

The HD is defined by:

$$HD(C_1, C_2) = \frac{1}{|Ib|} \sum_{K=1}^{|Ib|} \left(C_1(K) \oplus C_2(K) \right) \tag{13}$$

where $|Ib| = L \times C \times P \times 8$, is the size of the image in bits. The optimum HD value is 50%. A good stream cipher should produce an HD close to 50% (Wang et al., 2014). Table 7 indicates that the NPCR, UACI and HD values of the proposed stream cipher are very close to optimal values. Consequently a high resistance to differential attack is achieved.

Table 7 The NPCR, UACI and HD

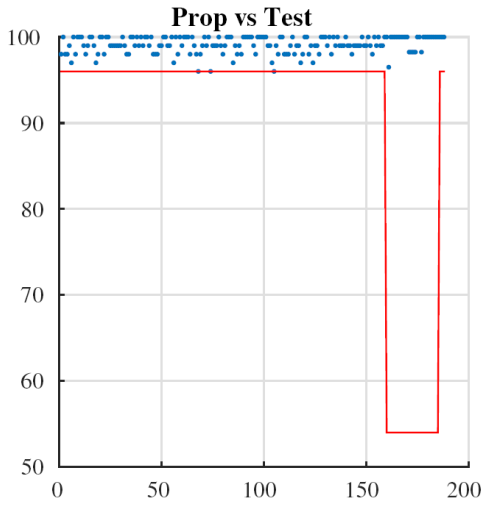
Cryptosystem	*NPCR*	*UACI*	*HD*
Proposed cipher cryptosystem	99.665	33.459	0.499999

5.3 Statistical analysis

5.3.1 NIST test

To evaluate the statistical performances of the Key stream produced, we also used one of the most popular standards for investigating the randomness of binary data, namely the NIST statistical test (Barker and Kelsey, 2012). This test is a statistical package that consists of 188 tests that were proposed to assess the randomness of arbitrarily long binary sequences. We applied the NIST test to many ciphered texts, and all the NIST results obtained, are as expected (good NIST values). In Figure 11 we present one of the NIST result obtained. This means that the ciphered texts have a high randomness.

Figure 11 NIST test key stream results (see online version for colours)



5.3.2 Histogram and chi-square test

A cryptosystem is considered to be strong against statistical attacks, if the histogram of the ciphered text is uniformly distributed. Visually, the uniformity test is necessary, but it is not sufficient. The chi-square test is applied to statistically confirm the uniformity of the histogram:

$$\chi^2_{\exp} = \sum_{i=0}^{Q-1} \frac{(o_i - e_i)^2}{e_i} \tag{14}$$

In equation (14), Q is the number of levels (here $Q = 256$), o_i is the observed occurrence frequency of each colour level (0–255) on the histogram of the ciphered image, and e_i is the expected occurrence frequency of the uniform distribution, given here by $e_i = \frac{L \times C \times P}{Q}$. For a secure cryptosystem, the experimental chi-square value must be less than the theoretical chi-square one, which is 293 in case of $\alpha = 0.05$ and $Q = 256$. In Figures 12, 13 and 14 we give the histograms for the plain/cipher images for lena, Boat and Camera man images on size 512 * 512 * 3. As we can see the histogram of the ciphered image seems to be uniform. To assess the uniformity, we performed the chi square test with the following parameters: alpha = 0.05, and number of classes equal to 256. Experimental value obtained is less than the theoretical one that equal 293. This means that the histogram is uniform (see Table 8).

Figure 12 Histogram of the Lena plain image and its ciphered image, (a) Lena plain image (b) Lena cipher image (c) histogram for plain image (d) histogram for the cipher image (see online version for colours)

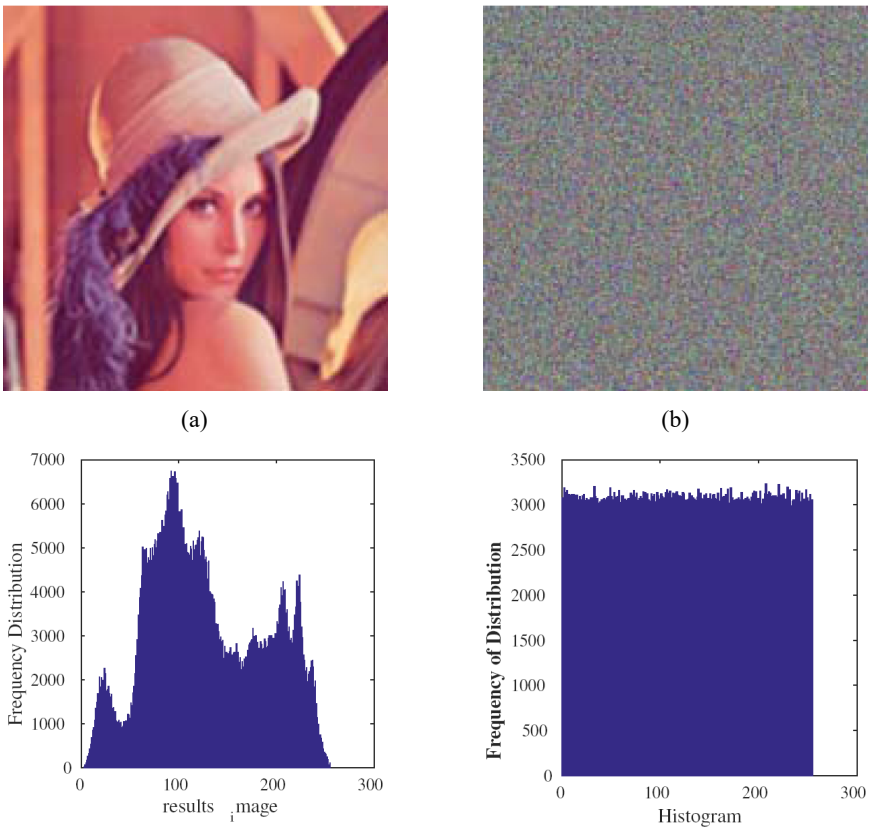
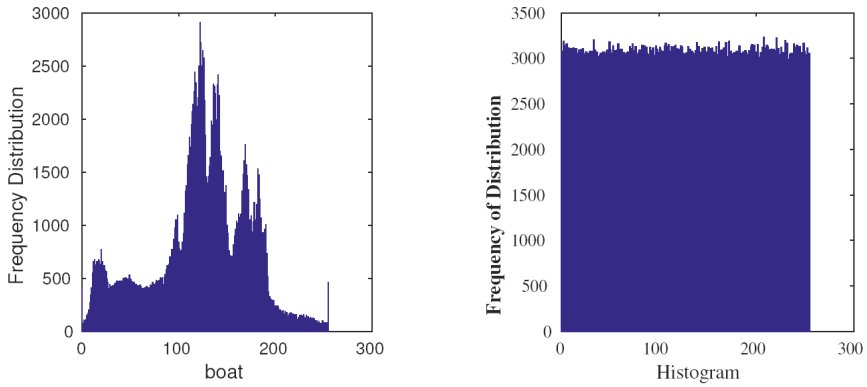


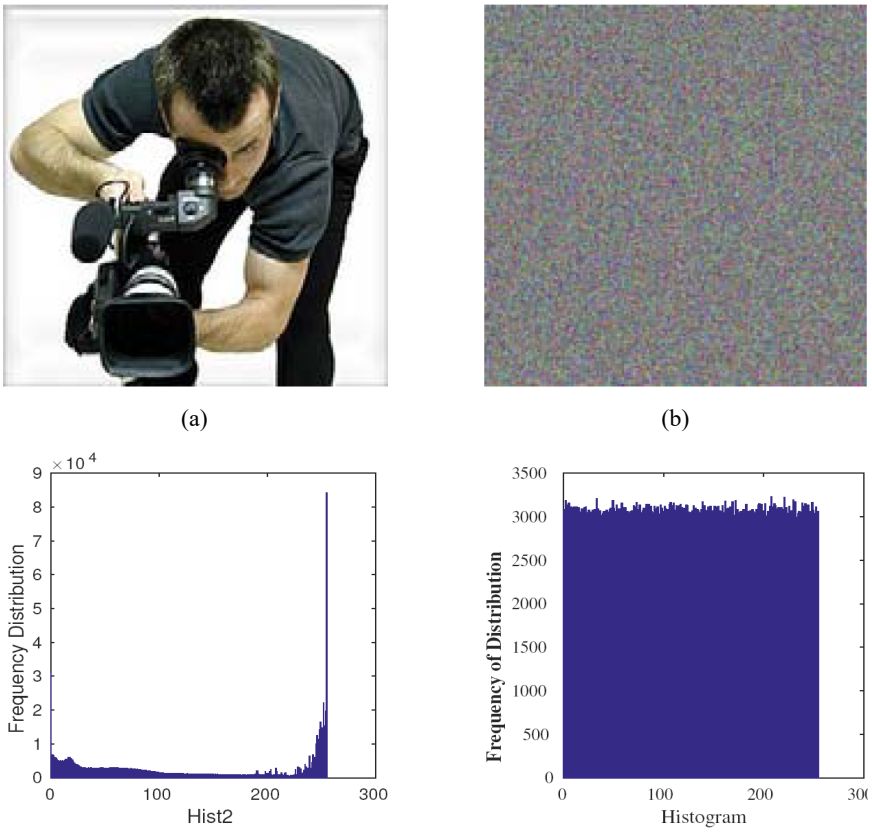
Figure 13 Histogram of the boat plain image and its ciphered image, (a) boat plain image (b) boat cipher image (c) histogram for plain image (d) histogram for the cipher image (see online version for colours)

Figure 13 Histogram of the boat plain image and its ciphered image, (a) boat plain image (b) boat cipher image (c) histogram for plain image (d) histogram for the cipher image (continued) (see online version for colours)



(c) (d)

Figure 14 Histogram of the camera man plain image and its ciphered image, (a) camera man plain image (b) camera man cipher image (c) histogram for plain image (d) histogram for the cipher image (see online version for colours)



(a) (b)

(c) (d)

Table 8 Chi-square value for ciphered Lena, boat and C-man with different sizes

Image	*Experimental value*	*Theoretical value*
Lena 256 × 256 × 3	261.085938	293.247835
Lena 512 × 512 × 3	263.013852	293.247835
Lena 1,024 × 1,024 × 3	270.300127	293.247835
Boat 256 × 256 × 3	260.186354	293.247835
Boat 512 × 512 × 3	266.465369	293.247835
Boat 1,024 × 1,024 × 3	272.669811	293.247835
C-man 256 × 256 × 3	261.339680	293.247835
C-man 512 × 512 × 3	267.317852	293.247835
C-man 1,024 × 1,024 × 3	274.397541	293.247835

5.3.3 Correlation analysis

Correlation analysis is also one of the statistical attacks that are used to cryptanalyse the cryptosystem. The attacker should not have any information of the used secret key or any partial information on the original plain image. This means that the encrypted image should be extremely different from its original version. Correlation analysis is one of the regular and standard methods to measure this property. Indeed, it is well-known that adjacent pixels in the plain images are very redundant and correlated. Thus, in the encrypted images, adjacent pixels should have a redundancy and a correlation as low as possible. The following mathematical equations are used to calculate the correlation coefficient (Song et al., 2013):

$$\rho_{xy} = \frac{\operatorname{cov}(x, y)}{\sqrt{D(x)}\sqrt{D(y)}} \tag{15}$$

where

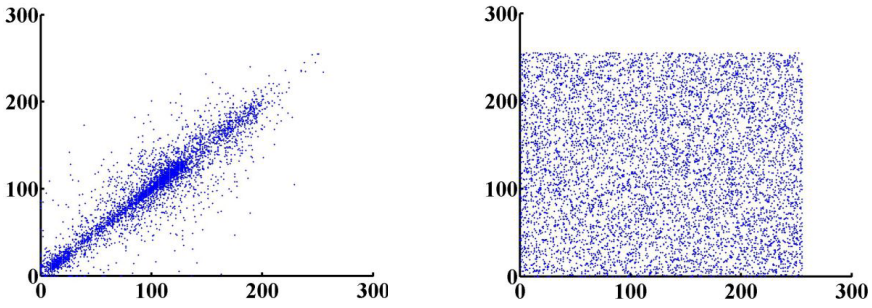
$$\operatorname{cov}(x, y) = \frac{1}{N}\sum_{i=1}^{N}\left(\left[x_i - E(x)\right]\left[y_i - E(y)\right]\right) \tag{16}$$

$$D(x) = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - E(x)\right)^2 \tag{17}$$

$$E(x) = \frac{1}{N}\sum_{i=1}^{N}\left(x_i\right) \tag{18}$$

In the previous equations, x_i and y_i are the values of the two adjacent pixels in the plain image or the corresponding ciphered image.

Figure 15 Correlation of the boat plain image and its ciphered image, (a) plain image correlation of adjacent pixels (b) ciphered image correlation of adjacent pixels (see online version for colours)



To test the security of our proposed stream cipher algorithm, regarding to this type of attack, first N pairs of adjacent pixels in vertical, horizontal, and diagonal directions are selected from the plain image and its ciphered version. Figure 15 shows the correlation curves of the adjacent pixels in the horizontal direction for the plain image and its ciphered one. The values of their corresponding correlation coefficient are 0.96606 and 0.0035. Similar results are obtained for the correlation in vertical and diagonal directions.

6 Conclusions

We proposed a new chaos-based stream cipher, useful for continuous communication as used in network communications. The heart of the system relies on a proposed chaotic generator that is designed and implemented in a secure and efficient manner with a sequential and parallel version. Its structure is modular, generic, and allow the production of high secure sequences. The performance in time for the proposed generator is better than other known PRNG. Also, For very big data size, the obtained performance results are better than other known stream ciphers. The proposed chaotic system is robust against cryptographic attacks. Furthermore, it has strong nonlinearity compared to the other systems. Indeed, the results obtained from the cryptographic analysis and of common statistical tests indicate the robustness of the proposed stream cipher. Our future work will focus on the design of chaos-based joint crypto-compression systems to secure videos: HEVC bitstream and MPEG-4.

References

Abderrahim, N., Benmansour, F. and Seddiki, O. (2014) 'A chaotic stream cipher based on symbolic dynamic description and synchronization', *Nonlinear Dynamics*, Vol. 78, No. 1, pp.197–207.

Abu Taha, M., El Assad, S., Farajallah, M., Queudet, A. and Deforge, O. (2015) 'Chaos-based cryptosystems using dependent diffusion: an overview', *10th International Conference for Internet Technology and Secured Transactions (ICITST)*, IEEE, pp.44–49.

Akhshani, A., Akhavan, A., Mobaraki, A., Lim, S-C. and Hassan, Z. (2014) 'Pseudo random number generator based on quantum chaotic map', *Communications in Nonlinear Science and Numerical Simulation*, Vol. 19, No. 1, pp.101–111.

Arlicot, A. (2014) *Sequences Generator Based Chaotic Maps*, Universit´e de Nantes, Tech. Rep., February.

Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A. et al. (1993) 'Automatic program parallelization', *Proceedings of the IEEE*, Vol. 81, No. 2, pp.211–243.

Barker, E. and Kelsey, J. (2012) *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, NIST SP 800-90 Rev A, Tech. Rep.

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M. et al. (2008) 'Sosemanuk, a fast software-oriented stream cipher', in *New Stream Cipher Designs*, pp.98–118, Springer, Berlin-Heidelberg.

Bernstein, D.J. (2008) 'The salsa20 family of stream ciphers', in *New Stream Cipher Designs*, pp.84–97, Springer, Berlin-Heidelberg.

Biham, E. and Shamir, A. (1991) 'Differential cryptanalysis of des-like cryptosystems', *Journal of CRYPTOLOGY*, Vol. 4, No. 1, pp.3–72.

Boesgaard, M., Vesterager, M., Christensen, T. and Zenner, E. (2005) *The Stream Cipher Rabbit*, ECRYPT Stream Cipher Project Report, Vol. 6.

Butenhof, D.R. (1997) *Programming with POSIX Threads*, Addison-Wesley Professional, Boston

Caragata, D., El Assad, S., Noura, H. and Tutanescu, I. (2010) 'Secure unicast and multicast over satellite dvb using chaotic generators', *International Journal of Internet Technology and Secured Transactions*, Vol. 2, Nos. 3–4, pp.357–379.

Chetto, M., El Assad, S. and Farajallah, M. (2014) 'A lightweight chaos-based cryptosystem for dynamic security management in real-time overloaded applications', *International Journal of Internet Technology and Secured Transactions 7*, Vol. 5, No. 3, pp.262–274.

Cimatti, G., Rovatti, R. and Setti, G. (2007) 'Chaos-based spreading in ds-uwb sensor networks increases available bit rate', *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 54, No. 6, pp.1327–1339.

Dagum, L. and Enon, R. (1998) 'Openmp: an industry standard api for shared-memory programming', *Computational Science & Engineering, IEEE*, Vol. 5, No. 1, pp.46–55.

Desnos, K., El Assad, S., Arlicot, A., Pelcat, M. and Menard, D. (2014) 'Efficient multicore implementation of an advanced generator of discrete chaotic sequences', *International Workshop on Chaos-Information Hiding and Security (CIHS)*.

Diffie, W. and Hellman, M.E. (1979) 'Privacy and authentication: an introduction to cryptography', Proceedings of the IEEE, Vol. 67, No. 3, pp.397–427, IEEE.

El Assad and Farajallah, M. (2016) 'A new chaos-based image encryption system', *Signal Processing: Image Communication*, Vol. 41, No. 11, pp.144–157.

El Assad, S. (2012) 'Chaos based information hiding and security', *International Conference for Internet Technology and Secured Transactions*, IEEE, pp.67–72.

eSTREAM (2008) *eSTREAM: The ECRYPT Stream Cipher Project* [online] http://www.ecrypt.eu.org/stream/ (accessed 10 May 2016).

Farajallah, M., El Assad, S. and Deforges, O. (2016) 'Fast and secure chaos-based cryptosystem for images', *International Journal of Bifurcation and Chaos*, Vol. 26, No. 2, p.21, 1-650-021–1–1-650-021–21.

Gropp, W., Lusk, E., Doss, N. and Skjellum, A. (1996) 'A high-performance, portable implementation of the mpi message passing interface standard', *Parallel Computing*, Vol. 22, No. 6, pp.789–828.

Gupta, S.S., Chattopadhyay, A., Sinha,K., Maitra, S. and Sinha, B.P. (2013) 'High-performance hardware implementation for rc4 stream cipher', *IEEE Transactions on Computers*, Vol. 62, No. 4, pp.730–743.

Gutterman, Z., Pinkas, B. and Reinman, T. (2006) 'Analysis of the linux random number generator', *IEEE Symposium on Security and Privacy*, IEEE, pp.2–16.

Jallouli, O., El Assad, S., Taha, M.A., Chetto, M., Lozi, R. and Caragata, D. (2016) 'An efficient pseudo chaotic number generator based on coupling and multiplexing techniques',*International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2016)*, pp.30040}.

Jo, I-H. and Koh, B-S. (2016) 'Building a common encryption scrambler to protect paid broadcast services', *International Journal of Internet Technology and Secured Transactions*, Vol. 6, No. 3, pp.167–177.

Kircanski, A. and Youssef, A.M. (2009) 'Differential fault analysis of rabbit', *International Workshop on Selected Areas in Cryptography*, Springer, pp.197–214.

Kircanski, A. and Youssef, A.M. (2010) 'Differential fault analysis of hc-128', *International Conference on Cryptology in Africa*, pp.261–278, Springer.

Kocarev, L. (2001) 'Chaos-based cryptography: a brief overview', *Circuits and Systems Magazine*, IEEE, Vol. 1, No. 3, pp.6–21.

Li, L. and Lee, J-H. (2016) 'On the security of a strong provably secure identity-based encryption scheme without bilinear pairing', *International Journal of Internet Technology and Secured Transactions*, Vol. 6, No. 3, pp.178–185.

Li, P., Li, Z., Halang, W.A. and Chen, G. (2007) 'A stream cipher based on a spatiotemporal chaotic system', *Chaos, Solitons & Fractals*, Vol. 32, No. 5, pp.1867–1876.

Lian, S., Sun, J. and Wang, Z. (2005) 'Security analysis of a chaos-based image encryption algorithm', *Physica A: Statistical Mechanics and its Applications*, Vol. 351, No. 2, pp.645–661.

Lian, S., Sun, J., Wang, J. and Wang, Z. (2007) 'A chaotic stream cipher and the usage in video protection', *Chaos, Solitons & Fractals*, Vol. 34, No. 3, pp.851–859.

Lipmaa, H., Wagner, D. and Rogaway, P. (2000) 'Comments to NIST concerning AES modes of operation: Ctr-mode encryption', *CiteSeerX Digital Library*, Vol. 1, pp.1–4.

Lozi, J-P., David, F., Thomas, G., Lawall, J. and Muller, G. (2016) 'Fast and portable locking for multicore architectures', *ACM Transactions on Computer Systems (TOCS)*, Vol. 33, No. 4, p.13.

Lu, H., Wang, S., Li, X., Tang, G., Kuang, J., Ye, W. and Hu, G. (2004) 'A new spatiotemporally chaotic cryptosystem and its security and performance analyses', *Chaos: An Interdisciplinary Journal of Nonlinear Science*, Vol. 14, No. 3, pp.617–629.

Machicao, J., Marco, A.G. and Bruno, O.M. (2012) 'Chaotic encryption method based on life-like cellular automata', *Expert Systems with Applications*, Vol. 39, No. 16, pp.12 626–12 635.

Manifavas, C., Hatzivasilis, G., Fysarakis, K. and Papaefstathiou, Y. (2015) 'A survey of lightweight stream ciphers for embedded systems', *Security and Communication Networks*, Vol. 9, No. 11, pp.1227–1246.

Mar, P.P. and Latt, K.M. (2008) 'New analysis methods on strict avalanche criterion of s-boxes', *World Academy of Science, Engineering and Technology*, Vol. 48, No. 12, pp.150–154.

Masoumi, M., Habibi, P., Dehghan, A., Jadidi, M. and Yousefi, L. (2016) 'Efficient implementation of power analysis attack resistant advanced encryption standard algorithm on side-channel attack standard evaluation board', *International Journal of Internet Technology and Secured Transactions*, Vol. 6, No. 3, pp.203–218.

Masuda, N. and Aihara, K. (2002) 'Cryptosystems with discretized chaotic maps', *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 49, No. 1, pp.28–40.

Maxime, B. (2016) *Comparative Analysis of Estream Ciphers*, Universite de Nantes, Tech. Rep., March.

Pacheco, P. (2011) *An Introduction to Parallel Programming*, 1st ed., Morgan Kaufmann [online] http://amazon.com/o/ASIN/0123742609/ (accessed 11 April 2016).

Rani, M.S. (2011) *An Efficient and Scalable Core Allocation Strategy for Multicore Systems*, PhD dissertation, Florida Atlantic University Boca Raton, FL.

Salehani, Y.E., Kircanski, A. and Youssef, A. (2011) 'Differential fault analysis of sosemanuk', *International Conference on Cryptology in Africa*, Springer, pp.316–331.

Setti, G., Rovatti, R. and Mazzini, G. (2005) 'Chaos-based generation of artificial self-similar traffic', in *Complex Dynamics in Communication Networks*, pp.159–190, Springer., Berlin-Heidelberg.

Siegenthaler, T. (1985) 'Decrypting a class of stream ciphers using ciphertext only', *IEEE Transactions on Computers*, Vol. 100, No. 1, pp.81–85.

Sinnen, O. (2007) *Task Scheduling for Parallel Systems*, Vol. 60, John Wiley & Sons, USA.

Smale, S. (1967) 'Differentiable dynamical systems', *Bulletin of the American Mathematical Society*, Vol. 73, No. 6, pp.747–817.

Song, C-Y., Qiao, Y-L. and Zhang, X-Z. (2013) 'An image encryption scheme based on new spatiotemporal chaos', *Optik-International Journal for Light and Electron Optics*, Vol. 124, No. 18, pp.3329–3334.

Stallings, W. (2006) *Cryptography and Network Security: Principles and Practices*, Pearson Education, India.

Taha, M.A., El Assad, S., Jallouli, O., Queudet, A. and Deforges, O. (2016) 'Design of a pseudo-chaotic number generator as a random number generator', *The 11th International Conference on Communications*, pp.401–404.

Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T. and Nakashima, H. (2007) 'Differential cryptanalysis of salsa20/8', *Workshop Record of SASC*.

Wang, X., Luan, D. and Bao, X. (2014) 'Cryptanalysis of an image encryption algorithm using chebyshev generator', *Digital Signal Processing*, Vol. 25, pp.244–247.

Wang, Y., Liu, Z., Ma, J. and He, H. (2016) 'A pseudorandom number generator based on piecewise logistic map', *Nonlinear Dynamics*, Vol. 83, No. 4, pp.2373–2391.

Wu, H. (2004) 'A new stream cipher hc-256', *International Workshop on Fast Software Encryption*, pp.226–244, Springer.

Wu, H. (2008) 'The stream cipher hc-128', in *New Stream Cipher Designs*, Vol. 2, pp.39–47, Springer, Berlin-Heidelberg.

Wu, Y., Noonan, J.P. and Agaian, S. (2011) 'Npcr and uaci randomness tests for image encryption', *Cyber Journals: Multidisciplinary Journals in Science and Technology, Journal of Selected Areas in Telecommunications (JSAT)*, Vol. 1, No. 2, pp.31–38.