Parallel Bit-Stream Cipher with Cellular Automata Number 30 in Java

Antonio J. Tomeu, Alberto G. Salguero
Dpto. de Ingeniería Informática
Escuela Superior de Ingeniería, Universidad de Cádiz
C/Chile s/n, E-11002
Cádiz (España)
{antonio.tomeu, alberto.salguero}@uca.es

Manuel I. Capel
Dpto. de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática y de Telecomunicaciones, Universidad de Granada
C/Periodista Daniel Saucedo s/n, E-18071
Granada, (España)
manuelcapel@ugr.es

***Abstract*—In previous works we have made the integration of one-dimensional linear cellular automata into bit-stream ciphers. These automata present a high degree of confusion-diffusion information and reasonable entropy levels. Our purpose is to obtain one-time pad ciphers to use in mobile communications. Cellular automaton number 30 is tested and validated to code the cipher. This work shows how to parallelize our encryption algorithm using Subramanian's equation to distribute the computational effort over a multicore CPU. Several standard multicore platforms will be used to verify performance gain.**

Keywords*—cellular automata; bit-stream cipher; entropy; randomness; parallel processing, multicore programming, Vernam cipher, randomness test, complexity, self-organization.

I. INTRODUCTION

The encryption of data is currently a must and unavoidable necessity. Several available encryption paradigms, such as symmetric and asymmetric (or public key), exist to solve this problem. The symmetric one is still widely used due to its low cost and efficiency [10]. When it comes to sending relatively small amounts of informationthe use of standard private key encryption algorithms (DES, AES) becomes inappropriate ([3], [4] and [10]). The same occurs with public key algorithms, which require a lot of time for processing a relatively small amount of information. Commercial approaches such as Pretty Good Privacy (PGP) are useful to reduce this problem.

In these cases, a more appropriate, efficient and quick alternative is to resort to encrypted bitstream approach, with or without using *one-time pad* ([4], [10] and [11]), which uses the encryption protocol just once, and then for next timediscards and replaces it with a different one.

This is accomplished easily and efficiently by using the bitstream encryption cipher ([10] and [18]), also known as Vernam cipher, as the general framework to build our *one-time pad*.

A bit stream cipher encrypts the characters in a plain text message using a sequence of binary digits in the form

$$\gamma_i=\mu_i\oplus\upsilon_i \text{ for } i=1, 2, 3... \quad (1)$$

where μ_i are the bits of the plain text and υ_i are bits of the cipher key, obtained by the equation γ_i, which ciphers the bits using the XOR function $\oplus$ (bitwise addition modulo 2). A bit stream cipher is obtained when υ_i bits are generated randomly and independently.

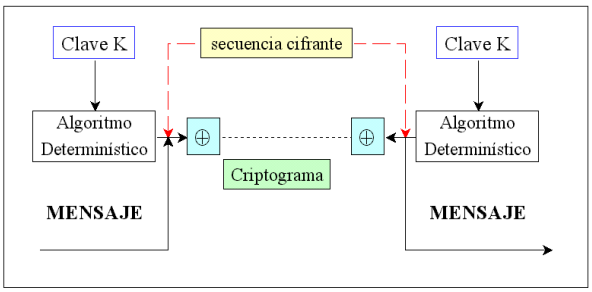

Fig 1[1]: General scheme of a Vernam cipher. Cipher bit sequence must be generated randomly by a reproducible process. This process is parameterized by a private key scheme.

The information retrieval is trivially simple, subject to the availability of the bitstream cipher υ_i, by just doing the following operation,

$$\mu_i=\gamma_i\oplus\upsilon_i \text{ for } i=1, 2, 3... \quad (2)$$

as the boolean function $\oplus$ is an involution.

Shannon proved ([4], [10]) that if a particular condition on the entropy of the key H(K) and message H(M) holds, being

1 Credits of this image are given to Prof. Dr. Jorge Ramió Aguirre and his e-book "Seguridad Informática y Criptografía", at free disposal in the author's website .

M, C and K random variables that respectively denote the flat texts and the encryption key, then the encryption is unconditionally safe if the cipher bit sequence is randomly chosen and independent of the bit- sequences of the message. This condition is given by the inequality,

$$H(K) \geq H(M)$$

This is because H (M|C|) = H(M) and therefore the amount of information I(M;C) must be considered null. Of course, once the *one-time pad* has been established, multiple instances of it can be used and discarded for each connection. It is at that point where the one-dimensional cellular automata, properly used, provides a stream of pseudo-random bits that allows the use of a one-time Vernam cipher, or alternatively the implementation of a reusable cipher stream changing the key of the cipher.

II. ONE-DIMENSIONAL CELLULAR AUTOMATA

A. *Cellular automata model definition*

A one-dimensional cellular automaton [19] is a tuple $<Z^{\infty}, \kappa, \rho, \delta>$ where:

- Z^{∞} is a row (or configuration) of discrete components called cells with possible values within a finite Abelian ring or residual class modulo κ. The evolution of the cellular automaton becomes a sorted and deterministic succession in Z^{∞}.
- ρ is the radius of neighborhood, and is a natural number greater than or equal to one. It determines the range of surrounding cells with which a given cell exchanges information.
- δ is the transition function that is used to recalculate the row of cells to obtain a succession of configurations.

The system evolves by discrete time steps, generating new configurations (or "generations" in bio-inspired terminology) of the cellular automaton by synchronously applying the transition function δ to each of the cells.

This transition function exploits the principle of locality to communicate any cell with each other and exchanges information between them, so that when a particular cell changes its state takes into account the information provided by the surrounding ρ cells left and right. The form of the transition function adopts the scheme shown in equation number 3:

$$\alpha^{(\tau+1)}_{\iota}=\delta(\alpha^{(\tau)}_{\iota-\rho},\ldots, \alpha^{(\tau)}_{\iota},\ldots, \alpha^{(\tau)}_{\iota+\rho}) \quad (3)$$

The symmetry of the neighborhood range of a particular cell is often imposed as a model condition, although this is not strictly necessary.

When the model is used in real problems, it is necessary to impose boundary conditions, being the cylindrical boundary the most common and most suitable shape to our needs. It is also useful to represent the transition functions using the *Wolfram* numerical coding technique, which lets us to specify a particular cellular automaton by a single number ([19]).

In [19] a classification of cellular automata behavior has been carried out. The results obtained are the same whether empirical or analytical analysis are used. All one-dimensional cellular automata behave according to one of the following four possibilities:

- Class 1: evolution leads to a metastable configuration that is perpetuated indefinitely. This configuration destroys all the information encoded in the initial configuration.
- Class 2: evolution leads to a set of simple patterns that are repeated regularly or remain stable.
- Class 3: evolution leads to a pattern with presence of chaos.
- Class 4: evolution leads to isolated structures with highly complex behaviors associated with dendritic-like structures that hold information processing capabilities (comparable to the class of Turing-computable functions, [13]) and sometimes are capable of universal computation.

In figure 2, examples of the four kinds of behavior above described for cellular automata with κ=2 states per cell are shown. Left column shows automata with a random initial configuration, whereas the right column shows automata with all their cells but the central one have a value of 0.

The boundary conditions used for computing the spatiotemporal automata shown in figure 2 are cylindrical.

Since it is intended to achieve a process that forces the confusion-diffusion of the information to be encrypted, it is necessary to choose cellular automata that exhibit chaotic behavior of class 3 ([2], [9]) as potential candidates for the generation of *one-time pad* reusable encryption flow.

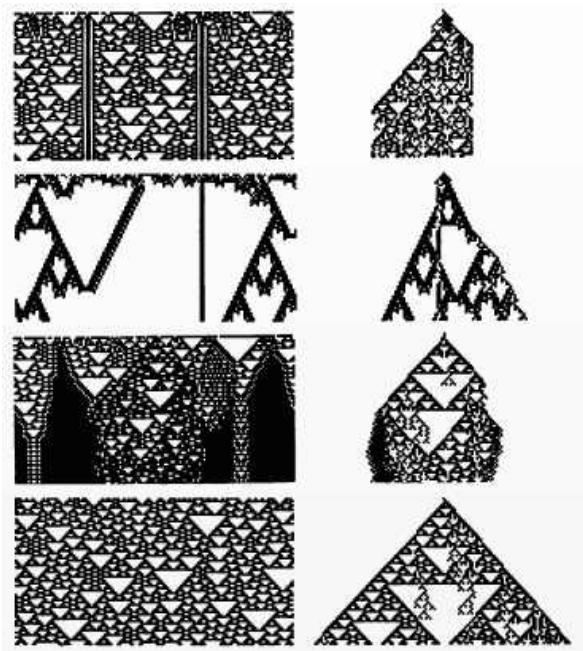

Fig 2: One-dimensional, binary cellular automata behavior classification. Left column shows automata with random initial configuration. Right column shows automata with a specific initial configuration.

B. *Analysis methodology*

Compliance with the *Shannon* criterion to ensure a strong encryption cannot be satisfied by simply identifying a pattern with chaotic behavior. It is necessary to analyze the pattern using an analytical modeling to establish that such a pattern is not spurious, but real ([2]).

For this, the method suggested in [19] and completely developed in [12] and [16] is applied. This methodology characterizes a one-dimensional cellular automaton with cylindrical border as chaotic and member of class 3 when the following three criteria are met:

- Passes five of a battery of six tests of randomness, at least: χ^2, runs, series, permutations, intervals and overlapped triplets.
- Has a configuration entropy value close to the unit.

$$S_t = -\sum_{i=0}^{k-1} p_i^{(j)} \log_k p_i^{(j)} \qquad (4)$$

- Has a temporal entropy defined by the equation

$$S_t = -\sum_{i=0}^{k-1} p_i^{(j)} \log p_i^{(j)} \qquad (5)$$

close to the unit.

With the above criteria, a powerful simulation package (©CaSim) has been designed in order to search for automata with good randomness properties that belong to class 3, which can be used to form *one-time pads*. This package is able to go over the space of the rules of one-dimensional cellular automata for any desired combination of pairs (κ, ρ), and then automatically generating a database of promising automata. It is also possible to search for cellular automata with the right properties in order to integrate them into a reusable stream cipher. In this work, we will follow the latter approach.

Figure 3 shows a snapshot of CaSim simulator running for a cellular automaton with three states per cell.

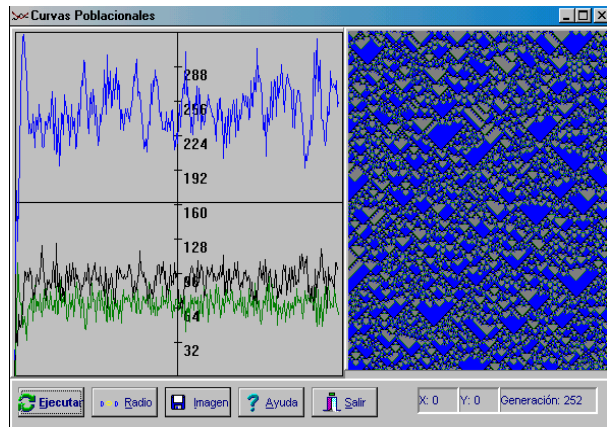

Fig. 3: CaSim Simulator running. Left side shows the evolution of the cell population in each of the possible states. Right side shows the spatiotemporal evolution of the three-state, random initial configuration and cylindrical boundary cellular automaton.

C. *Cellular Automata Number 30*

Given all of the binary one-dimensional cellular automata with neighborhood range equal to unit, the one with additive code number 30 has been chosen as the candidate to form the stream cipher. Its spatiotemporal evolution pattern is described in figure 4. Its transition function is given by

$$\alpha^{(\tau)}_{\iota}=(\alpha^{(\iota-1)}_{k-1} + \alpha^{(\iota-1)}_{k} + \alpha^{(\iota-1)}_{k+1} + (\alpha^{(\iota-1)}_{k}\alpha^{(\iota-1)}_{k+1})) \qquad (6)$$

Additionally, its performance in terms of randomness filtering and parameters entropy is described in table I. It is also widely studied in the literature ([19]). Table I summarizes the means of temporal entropy and configuration parameters.

TABLE I: CAHOTIC PARAMETERS OF CA30

Rule	S (mean)	S_t (mean)
30	0,96	0,98

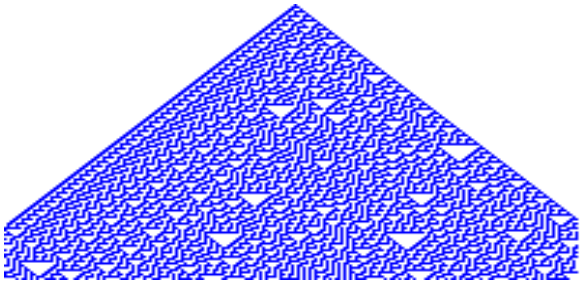

Fig. 4: Rule 30 simulated with CaSim with initial configuration where all cells but central are set to zero.

The same rule, when using a random initial configuration behaves as figure 5 shows.

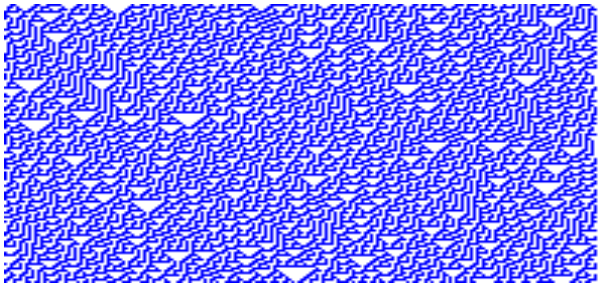

Fig. 5: Rule 30 simulated with CaSim with random initial configuration.

As shown, the behavior of the rule number 30 belongs to class 3 in terms of spatiotemporal evolutionary pattern. It is, therefore, a candidate to be used in the cipher. The next section verifies this assumption.

III. Proposed Bit-Stream Cipher

Once the two-dimensional cellular automaton that will act as a generator of pseudo-random cipher sequences has been chosen, it is necessary to consider how to get the sequences from the automaton. There are several proposals in the literature ([3], [4], [6], [7] and [8]), although the standard in these cases consists on taking the spatiotemporal evolution of the central cell in the array of cells. Its evolution will be used as the bitstream cipher.

A. *Cipher algorithm.*

Let us suppose the existence of a one-dimensional cellular automaton with rule number 30, as its random initial configuration (which will be key encryption). a_k is the central cell of the cellular automaton for a given array cell size. The bitstream encryption algorithm is described by the following equation,

$$\gamma_i=\mu_i \oplus a_k^{(\tau)} \text{ for } i=1, 2, \ldots \quad (7)$$

where the cipher text bits c_i are obtained from the XOR function applied to the plain text bits m_i and the cipher bits $a^{(i)}_k$. The latter is provided by the temporal evolution of the k-th central cell of the cellular automaton. From the above equation it is possible to write the pseudocode of the bitstream encryption algorithm as follows,

```
Bitstream cipher algorithm

INPUT: a message {μi} of n bits and a key χ(0)

OUPUT: cryptogram {γi} of n bits

METHOD:

1.Initialize the cellular automaton with key C(0)

2.For all i=1 to n

      2.1. γi=μi⊕χk(i)

      2.2. χi(i)=(χ(i-1)k-1 + χ(i-1)k + χ(i-1)k+1 + (χ(i-1)k . χ(i-1)k+1)) mod 2
```

The Java class below shows the implementation of the above algorithm to compute the cellular automaton Rule 30. The program used to implement stream ciphers is also shown.

```
import java.util.Random;
  public class aC30{
    int nCelulas;
    int[] estadoAC;
    int[] aux;
    int[] temp;

    public aC30(int nCelulas){
      this.nCelulas = nCelulas;
      Random      p = new Random();
      estadoAC      = new int[nCelulas];
      aux           = new int[nCelulas];

      for(int i=0; i<estadoAC.length; i++)
        estadoAC[i] = p.nextInt(2);
    }

    public int fTransicion(int i){

      if(i==0)
         return((estadoAC[estadoAC.length-1]+estadoAC[i]
           +estadoAC[i+1]+(estadoAC[i]*estadoAC[i+1]))%2);

      else if(i==estadoAC.length-1)
              return((estadoAC[i-1]+estadoAC[i]+estadoAC[0]+
                       estadoAC[i]*estadoAC[0]))%2);

            else return((estadoAC[i-1]+estadoAC[i]
                          +estadoAC[i+1]+
                           (estadoAC[i]*estadoAC[i+1]))%2);
    }


public void nuevoEstadoAC(){
      for(int i=0; i<estadoAC.length; i++)
        aux[i] = fTransicion(i);
      temp     = estadoAC;
      estadoAC = aux;
      aux = temp;
    }

    public int valCelula(int i){
     return(estadoAC[i]);
```

```
    }
}
```

Code 1 listing: Java classto model the cellular automaton 30.

The above class computes successive generations of cellular automata with rule number 30. It can be integrated into a bit stream cipher application as follows,

```
import java.util.Random;

public class cifraVernam{
  public static int nCel      = 1000;
  public static int longTexto;
  public static Random p      = new Random();
  public static int[] textoLlano;
  public static int[] textoCifrado;
  public static aC30 miAC  = new aC30(nCel);

  public static void main(String[] args){
    longTexto = Integer.parseInt(args[0]);
    textoLlano   = new int[longTexto];
    textoCifrado = new int[longTexto];
    for(int i=0; i<textoLlano.length; i++)
      textoLlano[i]=p.nextInt(2);
    long inicTiempo = System.nanoTime();
    for(int i=0; i<textoLlano.length; i++){
      textoCifrado[i]=(textoLlano[i]
                      +miAC.valCelula((int)nCel/2))%2;
    miAC.nuevoEstadoAC();
  }
3  long tiempoTotal = (System.nanoTime()-inicTiempo)/
                                        (long)1.0e9;
  System.out.println("Calculo finalizado en "+tiempoTotal+"
                                              segundos");
  }
}
```

Source code 2: Java class to compute plain text encryption.

The bits of the plain text string have been simulated using a random generator. The bits of the encrypted text have been computed using the cellular automata in *ac30.java*.

B. *Encryption security*

Although it has been considered as a secondary aspect in this work, whose main objective has been the parallelization of the bit stream encryption process using a cellular automaton number 30, security aspects must be also analyzed. A fast parallel encryption is useless if it is not safe. An elemental analysis of the security of the cellular automaton number 30 cipher is important.

The analysis starts studying the randomness of this automaton with respect to the previous tests described above. The following control parameters have been used:

- 1000 cells array.
- Random initial configuration.
- Cylindrical frontier condition.
- 1000 generations.

Table II below shows the results obtained:

TABLE II: CA30 RANDOMNESS TESTS

	χ^2	Runs	Perm.	Int.	Series	Trip.
CA30	√	√	√	√	√	√

A comparison with other usual techniques for obtaining random sequences of digits is shown in table III.

TABLA III: COMPARATIVA CON OTROS GENERADORES

	χ^2	Rachas	Perm.	Int.	Series	Trip.
CA60	√	-	-	-	-	-
LGC	-	√	√	√	√	√
LFSR	√	√	-	-	-	-
√2	√	√	√	√	√	√
e	√	√	√	√	√	√
∏	√	√	-	-	-	-

LCG (Lineal congruential generator); LFSR (Linear feedback shift register); √2, e y ∏, binary expansions of the respective digits.

The comparison between the chosen cellular automata and the other alternatives (some of them commercialized within many commonly used compilers, as LGC) is favorable to the former one. Additionally, the analysis of both the temporal and configuration entropies with rule number 30 automaton, in both cases, provides values close to the unit. The encryption bitstream is thus properly characterized regarding to its randomness.

Finally, it should be noted that a correlation analysis between the plain text and the equivalent cipher text sample shows differences less than 1%.

IV. BITSTREAM CIPHER PARALLELIZATION

The objective of this work is the parallelization of the encryption algorithm assuming be used in multi-core processors platforms. The encryption application should adapt the number of cryptographic tasks according to the number of available cores. The structure of the proposed parallelization strategy is shown in Figure 6.

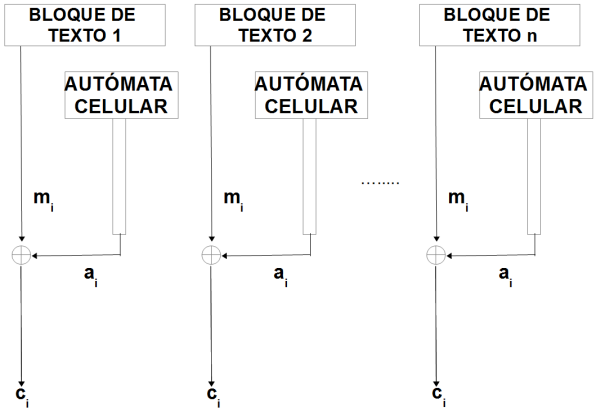

Fig. 6: Parallel encryption architecture with multiple instances of the cellular automaton.

For the development of the cipher, plain text will be divided into equal length blocks of bits ([15]). There will be many instances of cellular automata as blocks. The final encrypted text will be obtained applying the cipher to each plain text block.

A. *Number of tasks*

To decide the number of encryption tasks the known Subramanians' equation is used ([14]). This equation establishes the optimal number of parallel tasks taking into account two parameters:

- The number of available cores, N_{nd}.
- Blocking coefficient of the tasks, C_b.

From the above parameters, this equation establishes that the number of tasks is given by the following expression

$$N_t = N_{nd} / 1 - C_b \qquad (8)$$

where the blocking coefficient roughly quantifies the fraction of time that a particular task is waiting because of inputs/outputs, network delays, or other kind of latency. In this case, given that all tasks perform intensive computation, it is assumed that the blocking coefficient will have a value equal to zero. If a version dedicated to encrypt files is considered, the value of C_b will be in the range [0.5-0.99). The actual value must be empirically estimated.

B. *Parallel tasks structure*

The development of a parallel version of the previous code requires making a partition of the plain text sequence into equal size segments. The number of segments is determined by the above equation. Each of the segments will be encrypted by a dedicated task. Those tasks are independent of each other. Source code 3 shows the class that has been implemented to model such a task.

```
public class tareaVernam implements Runnable{
  private final int linf;
  private final int lsup;
  private final int[] tL;
  private final int[] tC;
  private final aC30Sinc  miac;
  private final int nCel;

  public tareaVernam(int linf, int lsup, int[]tL, int[] tC,
                     aC30Sinc miac, int nCel){
    this.linf = linf;
    this.lsup = lsup;
    this.tL   = tL;
    this.tC   = tC;
    this.miac = miac;
    this.nCel = nCel;
  }
  public void run(){
    for(int i=linf; i<=lsup;i++){
      tC[i]=(int)(tL[i]+miac.valCelula(nCel/2))%2;
      miac.nuevoEstadoAC();
    }
  }
}
```

Source code 3: `Runnable` task for the parallel encryption.

Each of the instances of the above class receives a segment of the plain text bits sequence to be encrypted. This sequence is initialized with the parameters *linf* and *lsup*, along with a reference to an instance of the class *a30Sinc*, which has been already described in this work. In this case, all of the container structures are objects of the class

```
java.util.concurrent.atomic.AtomicIntegerArray
```

which makes the read/write operations of tasks safe when they runin parallel. This is not strictly necessary in our approach for the encryption parallelization. Not using secure data structures for parallelism, the encryption time values offered in the analysis section may be slightly improved ([14]).

The main program determines how many independent cipher tasks are going to run in parallel. For this, it starts by making a system call to determine how many logical cores are available, considering a C_b value equal to zero

```
Runtime.getRuntime().availableProcessors();
```

and then processed using an executor service with fixed capacity, which can be easily created in Java just using the method *newFixedThreadPool(nTareas)* of the class *java.util.concurrency.Executors*. After that, the program just instantiates the tasks and transfer them to the executor in order to be processed:

```
ejecutor.execute(
         new tareaVernam(
                         linf,
                         lsup-1,
                         textoLlano,
                         textoCifrado,
                         new aC30Sinc(Clave),
                        clave));
```

As shown when instantiating a new object of the synchronized cellular automata class, each of the instances that are going to be run in parallel needs to receive its own key. In order encryption be effective and cryptographically robust,

these keys should be different. This ensures that each parallel encryption task generates a different cipher bitstream. The generation of the different keys can be achieved in two ways:

- using different keys explicitly, depending on the number of tasks that are going to be used or
- obtaining the keys from a single key entered by the user, thus using a technique that rotates its bits.

The latter is the usual technique in these situations and it is the one that has been used in the design and implementation of the proposed encryption algorithm.

V. PERFORMANCE ANALYSIS

To carry out the performance analysis of the parallel encryption algorithm we have used the following platforms:

- *Intel Core i7-3632QM* processor with eight logical cores (using Hyper Threading[2]), running at 2,2 GHz, eight GBytes of RAM memory and Windows 8 (64 bits) as the operative system (in figure 7).
- *Intel Core i3-2310M* processor with four logical cores (using Hyper Threading), running at 2,1 GHz, four GBytes of RAM memory, and Windows 8 (32 bits) and Linux Fedora 19 (*Schrödinger's Cat*) as the operative systems (in figure 8).

The obtained results are shown in figures 7 and 8.

As shown in figure 7 and as expected, the encryption time in the parallel version of the algorithm linearly grows. The same occurs with the sequential version, which uses just one core of the processor. The usual improvement gap is present, resulting from the use of all available cores that the platform offers. The obtained improvement is still low and some improvements are further applicable.

These results show the need to always scale the parallel encryption technique depending on the number of available cores, even using the GPU architectures when possible.

Using the second of the platforms, the obtained results are those shown in figure 8. In this case, the same behavior in figure 7 is obtained. The explanation is also the same. The main difference is the performance reduction obtained in the parallel version. This is a consequence of using half of the cores that we previously had. Also, it can be seen how the threads in Linux have a better performance than the threads in Windows, once the JVM threads are mapped to system threads.

Regarding the load of processor cores, it has also been obtained the expected behavior. The parallel version uses the full computation power that the platforms can offer, while the sequential versions use just a fraction of this.

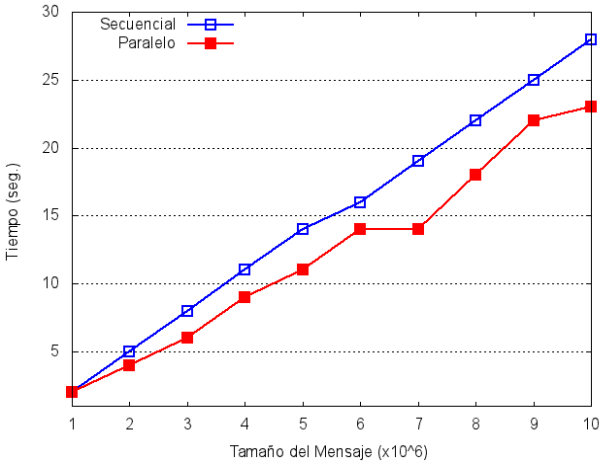

Fig. 7: Encryption time depending on the size of the plain text message (in bits). Sequential and parallel versions of the algorithm running on an Intel i7 platform.

Disk files oriented cipher versions have blocking coefficients values in the range [0.5-0.99), with corresponding associated latencies plus delays derived from the safe writing of the parallel tasks in the encrypted file. In this situation, the usage of the processor by the threads in the proposed parallel algorithm is lower than those obtained here but always superior to sequential versions.

2 Hyper-Threading is the simultaneous multi-threading commercial implementation of the Intel Corp.

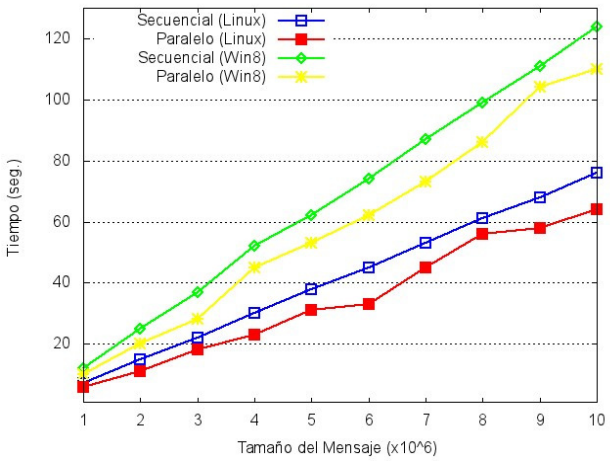

Fig. 8: Encryption time depending on the size of the plain text message (in bits). Sequential and parallel versions of the algorithm running on an Intel i3 platform running on Linux Fedora 19 and Windows 8 operating systems.

VI. CONCLUSIONS AND FUTURE WORK

As expected, the distribution of the workload of the sequential encryption algorithm to all the available cores in the local processor has reasonably improved the encryption time. By calculating subkeys using rotations of the original key, the security is also improved.

To use the new parallel encryption algorithm is possible for application design based on multi-core mobile technology, supported nowadays by Android. This is an interesting future line of research.

On the other hand, the literature provides standard methods for increasing the safety of bitstream cipher by deploying, among others, no linearization techniques ([17]), clock control encryption, linear filters encryption. A possible adaptation of the algorithm presented in this work is to implementthese variations, it is also being necessary to test the security enhancement by means of the same techniques used here. To check how these solutions behave once parallelized is also a must.

Additionally, the other major paradigm currently used in the field of symmetric ciphers is the use of encryption algorithms in blocks of bits, using Feistel networks as their mathematical basis for the encryption [15]. The bit block cipher is a proposal that is well suited to the parallel algorithm proposed here by its nature. Thus it will open a complementary line of research to one presented here, i.e., the parallelization of oriented block ciphers using cellular automata of class 3.

The parallelization technique for the encryption that we have used here also adapts well to a scheme of massive parallelism, like GPU-based architectures. Another of the lines of future work will consists of developing the encryption algorithm proposed on this kind of platforms. Also, a test about the cipher of images using processor cluster is planned.

The integration of a fuzzy cellular automata model, under current analysis ([5], [12]) within the cryptographic protocols with parallelized algorithms is also an interesting direction of study that worths the effort.

ACKNOWLEGMENTS

The authors want to thank the Computing Infrastructure Service at the University of Cadiz who made available the processor cluster.

REFERENCES

[1] O. Bandman, “Mapping Physical Phenomena onto CA-models”, Proceedings of Automata 2008 (Theory and Aplications of Cellular Automata), pp. 381-395, Luniver Press, 2008

[2] A. Adamatzky, B. de Lacy and T. Asai, “Reaction-Difussion Computers”, Elsevier, 2005.

[3] T. Ceccherini-Silberstein & M. Coornaet, “Cellular Automata and Groups”, Springer Monographs in Mathematics, 2010.

[4] T. W. Cusick, C. Ding and A. Renwall, “String Ciphers and Number Theory”, Elsevier Inc., 2004.

[5] A. Deustch and S. Dormann, “Cellular Automaton Modeling of Biological Pattern Formation”, Birkhäuser, 2004.

[6] P. García-Mochales y A. Fúster, “Un Algoritmo para Calcular el Número de Ceros y Unos en Secuencias Filtradas”. Actas de la VI Reunión Española de Criptología, pp. 297-307, 2000.

[7] D. de la Guía y A. Fúster, “Estudio de Autómatas Celulares para Cripografía”. Actas de la IV Reunión Española de Cripología, pp. 167-174, 1996.

[8] A. Fúster. “Cellular Automata in Stream Ciphers”, Contemporary Mathemathics, vol. 477, pp. 1-21, 2009.

[9] E. Jen, “Global Properpties of Cellular Automata”, Journal of Statystical Physics, vol. 43, issue 1-2, pp. 219-242, April 1986,

[10] A. Menezes, P. Van-Oorschot & S. Vanstone. “Handbook of Applied Cryptography”, 1996.

[11] M. Mihaljevic, Y. Zheng and H. Idei, “A Fast and Secure Stream Cipher Based on Cellular Automata Over GF(q)”, en http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.1157&rep=rep1&type=pdf

[12] F. Navarro y A. Tomeu, “Análisis de la Viabilidad de Integración de Autómatas Celulares Difusos en Protocolos de Cifrado de Flujo”, Actas de las II Jornadas Predoctorales, pp. tic1-tic4, 2010.

[13] P. Rendell, “A Fully Universal Turing Machine in Conway's Game of Life”, Journal of Cellular Automata, volume 8, number 1-2, pp. 19-38, 2013.

[14] V. Subramanian, "Programming Concurrency on the JVM: Mastering, Synchronization and Actors". The Pragmatic Bookshelf, 2011.

[15] M. Szaban & F. Seredinsky, "Designing Cryptographically Strong S-Boxes with Use of 1D Cellular Automata", Journal of Cellular Automata, volume , number 1, pp. 91-104, 2011.

[16] A. Tomeu, "Síntesis del Modelo de Autómatas Celulares en Protocolos Criptográficos de Cifrado en Flujo y vía Redes de Feistel". Ed. ProQuest, 2002.

[17] T. Torisawa, T. Komatsuzaki & Y. Saikawa, "Nonlinear Psedurandom Sequences Based on 90/150 LHGCA", Proceedings of 8th International Conference on Cellular Automata for Research and Insdustry, pp. 471-478. Springer, 2008.

[18] G. Vernam, "Cipher Printing Telegrah Systems for Secret Wire and Radio Telegraphic Communications" Journal of the IEEE, vol. 55, pp. 109-115, 1926.

[19] S. Wolfram, "Cellular Automata and Complexity: Collected Papers", Addison Wesley Publishing Company,1993.