Hiji-bij-bij : A New Stream Cipher with a Self-Synchronizing Mode of Operation

Palash Sarkar
Cryptology Research Group
Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108
e-mail: palash@isical.ac.in

Abstract

In this paper, we present a new stream cipher called Hiji-bij-bij (HBB). The basic design principle of HBB is to mix a linear and a nonlinear map. Our innovation is in the design of the linear and the nonlinear maps. The linear map is realised using two 256-bit maximal period 90/150 cellular automata. The nonlinear map is simple and consists of several alternating linear and nonlinear layers. We prove that the mixing achieved by the nonlinear map is complete and the maximum bias in any non-zero linear combination of the input and output bits of the nonlinear map is at most 2^{-13}. We also identify a self synchronizing (**SS**) mode of operation for HBB. The performance of HBB is reasonably good in software and is expected to be very fast in hardware. To the best of our knowledge, a generic exhaustive search seems to be the only method of attacking the cipher.

Keywords : stream cipher, self-synchronization, cellular automata.

1 Introduction

A stream cipher is a basic cryptographic primitive which has widespread applications in defence and commercial establishments. Like all basic primitives, designing a good stream cipher is a challenging task. In fact, in the past the design of block ciphers has received much more attention than stream ciphers – a fact which is perhaps attributable to the AES selection process. Fortunately, the NESSIE call for primitives rejuvenated the search for good stream ciphers.

In the past two years some new stream ciphers have been proposed. These can be called "second generation" stream ciphers, in the sense that they are block oriented and break away from the bit oriented and memoryless combiner model of classical stream ciphers. One of the advantages of avoiding the memoryless combiner model is the fact that many sophisticated attacks (such as [1, 2, 4]) on the memoryless combiner model cannot be directly applied. Some recent block oriented stream ciphers are SNOW [6], SCREAM [3], MUGI [14] and TURING [10] though not all of these were proposed as candidates for NESSIE evaluation. On the other hand, all the candidates for NESSIE evaluation displayed varying degrees of weaknesses. Some of these were repaired – SNOW [6] for example. The actual challenge in stream ciphers is to match speed to security. It seems difficult to judge the actual nature of the trade-off between speed and security. This question can perhaps be answered by more stream cipher proposals and their analysis. The current paper should be considered to be a contribution in this direction.

In this paper, we present a new stream cipher called Hiji-bij-bij (HBB). The cipher is also a "second generation" block oriented cipher. The basic design strategy is to "mix" a linear map and a nonlinear map. The nonlinear part is based on a round function which is similar to that of block ciphers. This seems to be the common design theme of many current stream ciphers – SNOW and SCREAM for example. The innovation we introduce is in the design of the linear and the nonlinear map.

The nonlinear map is simple and consists of alternating linear and nonlinear layers. Even though the nonlinear map is simple, we are able to prove that complete mixing is obtained. We also show that the maximum bias in any non-zero linear combination of input and output bits of the nonlinear map is at most 2^{-13}. Thus it seems unlikely that linear approximation and low diffusion attacks (such as [2]) can be applied to HBB. The linear map is realized using two maximal length 90/150 cellular automata (CA). The characteristic polynomials of these two CA are primitive of degree 256 and weight (number of non-zero terms) 129 each. This makes it difficult to obtain low degree sparse multiples of these polynomials. (In certain cases, such low degree sparse multiples can be used to attack a cipher.) One reason for choosing CA instead of LFSR is the fact that the shift between two sequences obtained from two cells of a CA can be exponential in the length of the CA (see [11, 12]). For an LFSR, this shift is at most equal to the length of the LFSR.

The basic mode of operation of HBB is as a synchronous stream cipher. We also identify a self-synchronizing (**SS**) mode of operation. The **SS** mode allows the sender and receiver to synchronize in the presence of errors. As soon as four consecutive cipher blocks (each of length 128 bits) are received correctly, the system automatically synchronizes. This is useful in error prone channels. See also [7] for various stream cipher modes of operation.

The speed of software implementation is around 2 operations per bit. The nonlinear map itself is quite fast – 75 operations for 128 bits. The linear map consisting of the next state functions of the two CA is comparatively slower (in software) and requires roughly 62% of the total time. The speed of software implementation of HBB is slower than other comparable stream ciphers such as SNOW or SCREAM. On the other hand, the total memory requirement to implement HBB is less than both SNOW and SCREAM. Also HBB is possibly more efficient to implement in hardware and a speed of 32 bits per clock cycle appears to be achievable. Finally HBB achieves complete diffusion and a maximum bias of 2^{-13}. SCREAM achieves a maximum bias of 2^{-9} and low diffusion; SNOW-2 achieves complete diffusion, but the bias is possibly higher. Thus in certain situations HBB would be preferable to other comparable stream ciphers.

In Section 2 we briefly outline some theory of CA required for understanding the application described here. The complete specification of HBB is given in Section 3. The time and memory requirements of HBB are discussed in Section 4. In Section 5, we provide a brief justification for the design choices of HBB. The features of HBB which can be changed without potentially affecting security are discussed in Section 6. Section 7 concludes the paper. The Appendix describes the origin of the name Hiji-bij-bij and also presents a software implementation of HBB.

2 Cellular Automata Preliminaries

A linear system can be described in terms of a linear transformation. Let $(s_1^{(t)}, \ldots, s_l^{(t)})$ denote the current state of an l-bit linear system. Then the next state $(s_1^{(t+1)}, \ldots, s_l^{(t+1)})$ is obtained by multiplying $(s_1^{(t)}, \ldots, s_l^{(t)})$ with a fixed $l \times l$ matrix M.

A CA is also a linear system which is defined by a matrix M. In case of CA, the matrix M is a tridiagonal matrix. The characteristic polynomial of M is the characteristic polynomial of all the linear recurrences obtained from each cell of the CA. If this characteristic polynomial is primitive, then the linear recurrence has the maximum possible period of $2^l - 1$. Further, it is known that if the characteristic polynomial is primitive, then the first upper and lower subdiagonal entries of M are all one. In this case the CA is called a 90/150 CA. Let $c_1 \ldots c_l$ be the main diagonal entries of M. Then the following relations hold for the state vectors of a 90/150 CA.

$$\left.\begin{array}{rclllll} s_1^{(t+1)} & = & & & c_1 s_1^{(t)} & \oplus & s_2^{(t)}, \\ s_i^{(t+1)} & = & s_{i-1}^{(t)} & \oplus & c_i s_i^{(t)} & \oplus & s_{i+1}^{(t)} \quad \text{for } 2 \le i \le l-1, \\ s_l^{(t+1)} & = & s_{l-1}^{(t)} & \oplus & c_l s_l^{(t)}. & & \end{array}\right\} \tag{1}$$

The vector $(c_1, \ldots, c_l)$ is called the 90/150 rule vector for the CA. A value of 0 indicates rule 90 and a value of 1 indicates rule 150. See [12] for an algebraic analysis of CA sequences.

3 HBB Specification

We identify two modes of operation for HBB: Basic (**B**), and Self Synchronizing (**SS**) mode. We use a variable MODE to indicate the mode of operation (**B** or **SS**) of the system.

We assume that the message is $M_0||M_1||\ldots||M_{n-1}$, where each M_i is an 128-bit block. The secret (or symmetric) key of the system is KEY. We denote the pseudo random key stream by $K_0||\ldots||K_{n-1}$ and the cipher stream by $C_0||\ldots||C_{n-1}$, where each K_i and C_j are 128-bit blocks.

HBB uses 640 bits of internal memory which is partitioned into two portions – a linear core LC of 512 bits and a nonlinear core NLC of 128 bits.

```
HBB($M_0||\ldots||M_{n-1}$, KEY)
1.  LC = Exp(KEY); $F$ = Fold(KEY, 64); NLC = $F||\overline{F}$;
2.  for $i = 0$ to 15 do ($T_{i \bmod 4}$, LC, NLC) = Round(LC, NLC);
3.  $\mathsf{LC}_{-1}$ = LC $\oplus$ ($T_3||T_2||T_1||T_0$); $\mathsf{NLC}_{-1}$ = NLC; $C_{-3} = T_3$; $C_{-2} = T_2$; $C_{-1} = T_1$;
4.  for $i = 0$ to $n-1$ do
5.      ($K_i$, $\mathsf{LC}_i$, $\mathsf{NLC}_i$) = Round($\mathsf{LC}_{i-1}$, $\mathsf{NLC}_{i-1}$);
6.      $C_i = M_i \oplus K_i$;
8.      if (MODE = SS)
            $\mathsf{LC}_i$ = Exp(KEY) $\oplus$ ($C_i||C_{i-1}||C_{i-2}||C_{i-3}$);
            $\mathsf{NLC}_i$ = Fold(KEY, 128) $\oplus C_i \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3}$;
10.     end if;
11. enddo;
12. output $C_0||\ldots||C_{n-1}$;
```

The function HBB() actually describes the encryption algorithm for the cipher. The corresponding decryption algorithm is obtained by the following changes to HBB().

- Change line 6 to "$M_i = C_i \oplus K_i$".
- Change line 12 to "output $M_0||\ldots||M_{n-1}$".

The function HBB() invokes several other functions – Exp(), Fold(), Round() and NLSub(). First we define the function Fold(S, i), where S is a binary string whose length is a positive integral multiple of i. We note that Fold() is described in algorithm form merely for convenience of description. During implementation we can avoid actually implementing the function; the required code will be inserted inline.

```
Fold$(S, i)$
1.  write $S = S_1||S_2||\ldots||S_k$, where $|S| = k \times i$ and $|S_j| = i$ for all $1 \leq j \leq k$;
2.  return $S_1 \oplus S_2 \oplus \ldots \oplus S_k$.
```

Remarks :

1. There are two allowed sizes for the secret key – 128 and 256 bits. If the key size is 256 bits, then $\mathsf{Exp}(\mathsf{KEY}) = \mathsf{KEY}||\overline{\mathsf{KEY}}$; and if the key size is 128 bits, then $\mathsf{Exp}(\mathsf{KEY}) = \mathsf{KEY}||\overline{\mathsf{KEY}}||\overline{\mathsf{KEY}}||\mathsf{KEY}$. It is possible to allow other key sizes between 128 and 256 bits by suitably defining the key expansion function Exp().

2. In the **B** mode, the system operates as a synchronous stream cipher. In the **SS** mode the system operates as a self-synchronizing stream cipher. As soon as four cipher blocks are correctly received the system automatically synchronizes and subsequent decryption is properly done.

3. We assume that a maximum of 2^{64} bits are generated from a single value of the secret key KEY. This is mentioned to provide an upper bound on cryptanalysis.

The Round() function takes as input LC and NLC and returns the next 128 bits of the keystream as output along with the next states of LC and NLC. We consider LC to be an array of length 16 where each element of the array is a 32-bit word. These are formed from the current states of two 256-bit CA; the current state of the first CA is given by LC[0,...,7] and the current state of the second CA is given by LC[8,...,15]. In the description below, the operation $x \lll i$ denotes the left cyclic shift of the binary string x by i. In the following, we write $\mathsf{NLC} = \mathsf{NLC}_0||\mathsf{NLC}_1||\mathsf{NLC}_2||\mathsf{NLC}_3$, where each NLC_i is a 32-bit word.

```
Round(LC, NLC)
1.  NLC = NLSub(NLC);
2.  $\Delta = \mathsf{NLC}_0 \oplus \mathsf{NLC}_1 \oplus \mathsf{NLC}_2 \oplus \mathsf{NLC}_3$, where $|\mathsf{NLC}_i| = 32$;
3.  for $i = 0$ to 3 $\mathsf{NLC}_i = (\Delta \oplus \mathsf{NLC}_i) \lll (8 * i + 4)$;
4.  NLC = FastTranspose(NLC);
5.  NLC = NLSub(NLC);
6.  LC = NextState(LC);
7.  $K_0 = \mathsf{NLC}_0 \oplus \mathsf{LC}_0$; $K_1 = \mathsf{NLC}_1 \oplus \mathsf{LC}_7$; $K_2 = \mathsf{NLC}_2 \oplus \mathsf{LC}_8$; $K_3 = \mathsf{NLC}_3 \oplus \mathsf{LC}_{15}$;
8.  $\mathsf{NLC}_0 = \mathsf{NLC}_0 \oplus \mathsf{LC}_3$; $\mathsf{NLC}_1 = \mathsf{NLC}_1 \oplus \mathsf{LC}_4$; $\mathsf{NLC}_2 = \mathsf{NLC}_2 \oplus \mathsf{LC}_{11}$; $\mathsf{NLC}_3 = \mathsf{NLC}_3 \oplus \mathsf{LC}_{12}$;
9.  return $(K_0||K_1||K_2||K_3, \mathsf{LC}, \mathsf{NLC})$.
```

The 128-bit string NLC is viewed in two ways in the function Round() – as an array of bytes of length 16 and as a 4×32 matrix, where each row of the matrix is a 32-bit word. The operation NLSub(NLC) treats NLC as an array of bytes of length 16 whereas FastTranspose(NLC) treats NLC as a 4×32 matrix.

The call NLSub(NLC) is a nonlinear permutation on NLC. It applies a byte substitution function byteSub() on each byte of NLC. The function byteSub() is the byte substitution function of AES [5]; which is the inverse mapping over $GF(2^8)$ followed by an affine transformation over $GF(2)^8$. (See [9] for details of the inverse map and [5] for details of the byteSub() function.) The function byteSub() is implemented by a look-up table of size 256 bytes.

In the call FastTranspose(NLC), the 128-bit string NLC is considered to be a 4×32 matrix where each row consists of four 32-bit words. We next describe the algorithm FastTranspose() to compute the (partial) transpose M^{t} of a matrix M. Define a set $\mathsf{Masks} = \{m_0, \ldots, m_{2s-1}\}$, where each m_i is a binary string of length 2^s defined as follows: $m_{2i} = (x_i)^{2^{s-i-1}}$; $m_{2i+1} = \overline{m_{2i}}$, where $x_i = 1^{2^i}0^{2^i}$ and $0 \le i < s$. For example, when $s = 3$, $m_0 = 10101010$, $m_1 = 01010101$, $m_2 = 11001100$, $m_3 = 00110011$, $m_4 = 11110000$ and $m_5 = 00001111$. The ith row of the matrix M will be denoted by M_i.

```
FastTranspose(M)
Input : A $2^r \times 2^s$ matrix $M = [A_0, \ldots, A_{2^{r-s}-1}]$, where each $A_i$ is a $2^r \times 2^r$ matrix.
Output : $[A_0^{\mathsf{t}}, \ldots, A_{2^{s-r}-1}^{\mathsf{t}}]$.
1.  for $i = 0$ to $r - 1$ do
2.      for $j = 0$ to $2^i - 1$ do
3.          for $k = 0$ to $2^{r-i-1} - 1$ do
4.              $k_1 = j + k2^{i+1}$; $k_2 = k_1 + 2^i$;
5.              $x_1 = M_{k_1} \wedge m_{2i}$; $x_2 = (M_{k_1} \wedge m_{2i+1}) \ll 2^i$;
6.              $y_1 = (M_{k_2} \wedge m_{2i}) \gg 2^i$; $y_2 = M_{k_2} \wedge m_{2i+1}$;
7.              $M_{k_1} = x_1 \oplus y_1$; $M_{k_2} = x_2 \oplus y_2$;
8.          enddo;
9.      enddo;
10. enddo;
11. return $M$.
```

Proposition 1 *Let M be a $2^r \times 2^s$ matrix with $r \le s$. Then the invocation* FastTranspose(M) *requires $r2^{r+2}$ bitwise operations on strings of length 2^s. Moreover, if $r = s$, then the output is the transpose M^{t} of M.*

The function Round() invokes FastTranspose(M) with a 4×32 matrix M. This requires a total of $2 \times 2^{2+2} = 32$ bitwise operations on strings of length 32. Since a 32-bit string is represented by an unsigned integer, each bitwise operation takes constant time to be executed on a 32-bit architecture.

The algorithm FastTranspose is actually a more general algorithm than is required for HBB. It can also be used in other situations where matrix transpose is required. The structure of the algorithm is similar to the butterfly network used for the fast Fourier transform algorithm.

To complete the description of HBB() we have to define the function NextState(). This function is a linear map from 512-bit strings to 512-bit strings. It is usual to realise such maps using linear finite state machines. We use two 256-bit maximal period 90/150 CA to realise this map. Recall that LC$[0, \ldots, 7]$ and LC$[8, \ldots, 15]$ are the current states of the two CA.

```
NextState(LC)
1. LC$[0, \ldots, 7]$ = EvolveCA(LC$[0, \ldots, 7], \mathcal{R}_0$);
2. LC$[8, \ldots, 15]$ = EvolveCA(LC$[8, \ldots, 15], \mathcal{R}_1$);
3. return LC.
```

The variables $\mathcal{R}_0$ and $\mathcal{R}_1$ are the 90/150 rule vectors for the two CA (see Section 2). The characteristic polynomials $p_0(x)$ and $p_1(x)$ of the two CA are distinct primitive polynomials of degree 256 and weight 129 each. These polynomials are defined below.

$$\begin{aligned} p_0(x) &= a_0 \oplus a_1 x \oplus \cdots \oplus a_{255} x^{255} \oplus x^{256} \\ p_1(x) &= b_0 \oplus b_1 x \oplus \cdots \oplus b_{255} x^{255} \oplus x^{256} \end{aligned}$$

The strings $a_0 \dots a_{255}$ and $b_0 \dots b_{255}$ are binary strings of length 256 and weight 128 each. In hex form these strings are as follows.

$$(a_0 \dots a_{255})_{16} = \texttt{947cde759c5fa5ba1507083b24e588e9c0d519a0094eb0e06ff32adb78e4df95}$$
$$(b_0 \dots b_{255})_{16} = \texttt{e8aa3542c32bd8391048add5e7acc24662a30b76da40dbf53829eebead9f109f}$$

The 90/150 rule vector of the two CA corresponding to these two primitive polynomials is found using the algorithm of Tezuka and Fushimi [13] and in hex form is given below.

$$\mathcal{R}_0 : \texttt{80ffaf46977969e971553bb599be6b2b4b3372952308c787b84c7cce36d501e6}$$
$$\mathcal{R}_1 : \texttt{dd18c62b153df31ac98e86c1910fee242942d51b4201eb3dc1d1a85f57b8919b}$$

The function EvolveCA() is simple to describe.

```
EvolveCA(state, rule)
1. state = (state ≪ 1) ⊕ (rule ∧ state) ⊕ (state ≫ 1);
2. return state;
```

Note that a software implementation of EvolveCA() requires 5 bitwise operation on k-bit strings, where $k = |\text{state}|$. If $k = 32$, then each bitwise operator corresponds to one operation on a 32-bit machine. For our case $k = 256$, and hence more than one operation is required to implement one bitwise operation. In hardware, the next state of each CA can be computed in one clock cycle. A 3-input XOR gate is required to compute the next state of each cell and the computation of the next states of all the cells can be done in parallel.

4 Implementing HBB

In this section, we discuss the software and hardware implementation of HBB. In the following by a word we will mean 4 bytes.

4.1 Software Implementation

The total state memory of HBB is 640=(512+128) bits which is 80 bytes. The implementation of byteSub() takes 256 bytes. Some additional 32-bit internal variables/constants are required. It turns out that a total of 16 words are sufficient to implement HBB. Thus the state memory is 336 bytes and the internal memory is 64 bytes. We note that an efficient software implementation of both SNOW and SCREAM require 4 look-up tables of size 256×4 bytes each. Hence the total size of the look-up tables alone is 4 Kbyte. Clearly HBB is smaller.

For fast software implementation, the loops in the subroutines will be "unfolded". The size of the resulting code is roughly proportional to the number of operations required to implement one invocation of Round(). As we show below, this number is 219 and hence the code will consist of around 250 instructions.

We now turn to estimating the number of operations required to complete one invocation of Round().

Steps 1 and 5 of Round() require 16 look-ups each; Steps 2 and 3 require a total of 11 bitwise operations and Step 4 requires 32 bitwise operations. These steps constitute the nonlinear map of the system and hence the nonlinear map requires 75 operations for computing the next 128 bits of the keystream.

Step 6 consists of applying the next state function to LC. This in turn consists of evolving the two CA once each. In software, this operation is relatively expensive. Below we provide the pseudocode for obtaining the next state of LC. For the sake of convenience we show the next state computation of the first CA only whose state is given by $\mathsf{LC}[0,\ldots,7]$. We write LC_i for $\mathsf{LC}[i]$. We assume that the rule vector of the first CA is given by $R_0,\ldots,R_7$ where each R_i is a 32-bit word. The rule vector for the second CA is given by $R_8,\ldots,R_{15}$.

```
1. $\mathsf{tmp}_0 = ((\mathsf{LC}_0 \ll 1) \oplus (\mathsf{LC}_1 \gg 31)) \oplus (R_0 \wedge \mathsf{LC}_0) \oplus (\mathsf{LC}_0 \gg 1);$
2. $\mathsf{tmp}_1 = ((\mathsf{LC}_1 \ll 1) \oplus (\mathsf{LC}_2 \gg 31)) \oplus (R_1 \wedge \mathsf{LC}_1) \oplus ((\mathsf{LC}_1 \gg 1) \oplus (\mathsf{LC}_0 \ll 31));$
3. $\mathsf{tmp}_2 = ((\mathsf{LC}_2 \ll 1) \oplus (\mathsf{LC}_3 \gg 31)) \oplus (R_2 \wedge \mathsf{LC}_2) \oplus ((\mathsf{LC}_2 \gg 1) \oplus (\mathsf{LC}_1 \ll 31));$
4. $\mathsf{tmp}_3 = ((\mathsf{LC}_3 \ll 1) \oplus (\mathsf{LC}_4 \gg 31)) \oplus (R_3 \wedge \mathsf{LC}_3) \oplus ((\mathsf{LC}_3 \gg 1) \oplus (\mathsf{LC}_2 \ll 31));$
5. $\mathsf{tmp}_4 = ((\mathsf{LC}_4 \ll 1) \oplus (\mathsf{LC}_5 \gg 31)) \oplus (R_4 \wedge \mathsf{LC}_4) \oplus ((\mathsf{LC}_4 \gg 1) \oplus (\mathsf{LC}_3 \ll 31));$
6. $\mathsf{tmp}_5 = ((\mathsf{LC}_5 \ll 1) \oplus (\mathsf{LC}_6 \gg 31)) \oplus (R_5 \wedge \mathsf{LC}_5) \oplus ((\mathsf{LC}_5 \gg 1) \oplus (\mathsf{LC}_4 \ll 31));$
7. $\mathsf{tmp}_6 = ((\mathsf{LC}_6 \ll 1) \oplus (\mathsf{LC}_7 \gg 31)) \oplus (R_6 \wedge \mathsf{LC}_6) \oplus ((\mathsf{LC}_6 \gg 1) \oplus (\mathsf{LC}_5 \ll 31));$
8. $\mathsf{tmp}_7 = (\mathsf{LC}_7 \ll 1) \oplus (R_7 \wedge \mathsf{LC}_7) \oplus ((\mathsf{LC}_7 \gg 1) \oplus (\mathsf{LC}_6 \ll 31));$
9. $\mathsf{LC}_0 = \mathsf{tmp}_0; \mathsf{LC}_1 = \mathsf{tmp}_1; \mathsf{LC}_2 = \mathsf{tmp}_2; \mathsf{LC}_3 = \mathsf{tmp}_3;$
10. $\mathsf{LC}_4 = \mathsf{tmp}_4; \mathsf{LC}_5 = \mathsf{tmp}_5; \mathsf{LC}_6 = \mathsf{tmp}_6; \mathsf{LC}_7 = \mathsf{tmp}_7;$
```

A total of 68 operations are required in the above code. Thus Step 6 requires a total of $68 \times 2 = 136$ operations. Finally Steps 7 and 8 require a total of 8 operations. Thus Round() can be completed using $75 + 136 + 8 = 219$ operations which is also the number of operations required to produce the next 128 bits of the keystream. Generting the cipher requires 4 more bitwise operations and the **SS** mode requires an additional 34 or 36 operations (depending on whether the secret key length is 128 or 256). Thus the **SS** mode is only marginally more expensive than the **B** mode of operation, a feature which makes the cipher attractive in error prone channels. An implementation of the **B** mode of operation of HBB is given in the Appendix and the speed conforms to the calculation made above.

4.2 Hardware Implementation

In this section we briefly describe a strategy for hardware implementation of HBB. The main task is to implement the function Round() which consists of three parts.

Part-A (Steps 1 to 5): These steps update NLC.

Part-B (Step 6): This step updates LC.

Part-C (Step 7 and 8): Generates keystream and updates NLC.

First we note that **Part-A** and **Part-B** can be done in parallel. **Part-C** can only be done after both **Part-A** and **Part-B** have been completed. We now consider the time required for each part in the reverse order.

1. **Part-C :** In Step 7, the next 128 bits of the keystream are generated and in Step 8, the value of NLC is updated. These two steps can be completed in one clock cycle by using 2×128 XOR gates. One set of 128 gates is used to generate the keystream from NLC and LC while the other set of 128 gates is used to obtain the next state of NLC from the current state of NLC and LC. The gate delay and input latching to the gates ensures that there is no read/write conflict for NLC.

2. **Part-B :** Step 6 consists of evolving the two CA exactly once each. In hardware each CA is realised using a 256-bit register and 256 XOR gates. Thus the next state of both CA can be obtained in one clock cycle.

3. **Part-A** (Steps 1 and 5): Each of these two steps perform a nonlinear substitution. To perform each of these steps in one clock cycle we must use 16 different copies of the look-up table implementing byteSub(). Each copy of byteSub() requires 256 bytes and hence the 16 copies will require a total of 4 Kbytes. Given these 16 tables, the invocation NLSub(NLC) can be completed in one clock cycle in the following manner: NLCconsists of 16 bytes and each byte is "fed" into its corresponding copy of byteSub() and the output is used to replace the original byte. Thus Steps 1 and 5 take a total of 2 clock cycles.

4. **Part-A** (Steps 2,3 and 4): These steps perform an affine transformation and a bit permutation on NLC. Let M be the state of NLC before Step 2 and $M^{'}$ be the state of NLC after Step 4. Consider both M and $M^{'}$ to be 4×32 matrix of bits. The combined effect of Steps 2,3 and 4 can be described as follows: First each row of M is replace by a XOR of the other three rows and then the cyclic shifts and the transpose perform a bit permutation. Thus the combined effect of Steps 2,3 and 4 is that each bit of $M^{'}$ is obtained as a XOR of three bits of M. Moreover, given any bit position of $M^{'}$, there are three fixed bit positions of M which are XOR-ed to obtain the value of this position. These three bit positions are known a priori and do not change during the computation of the algorithm. Hence each bit of $M^{'}$ can be obtained in one clock cycle using two 2-input XOR gates and consequently all the 128 bits of $M^{'}$ can be computed in one clock cycle using 256 2-input XOR gates.

To summarise the above, **Part-A** can be completed in 3 clock cycles, **Part-B** in one clock cycle and **Part-C** in one clock cycle. Also **Part-A** and **Part-B** can be computed in parallel. Hence a total of 4 clock cycles is sufficient to complete one invocation of Round(). Thus the next 128 bits of the keystream is produced in 4 clock cycles leading to a throughput of 32 bits per clock cycle (in the **B** mode). However, the above description is somewhat theoretical and only indicates the possibility of 32 bits per clock cycle. A detailed hardware design and performance analysis can properly answer the question of hardware efficiency. This work we leave for the future. The main point that we have tried to make here is that hardware implementation of HBB is reasonably simply and the performance is expected to be attractive.

5 Design Goals

We consider some of the design principles for HBB. The nonlinear map has been designed to provide complete mixing and a low bias. These are proved below. We also discuss the **SS** mode of operation.

5.1 Mixing and Bias

Steps 1 to 5 of the function Round() define a nonlinear bijection $\Psi : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$. Let $M^{(0)}$ be the state of NLC before Step 1 of Round() and let $M^{(5)}$ be the state of NLC after Step 5 of Round(). The state of NLC after Step 1 will be denoted by $M^{(1)}$. The effect of Steps 2 and 3 can be divided into two separate operations – the first operation is to replace each row by the XOR of the other three rows and the second operation is to left shift each row by a specified amount. Let $M^{(2)}$ and $M^{(3)}$ be the states of NLC after the XOR operation and row shiftings respectively. Finally let $M^{(4)}$ be the state of NLC after Step 4. The output of Ψ is $M^{(5)}$ and the input of Ψ is $M^{(0)}$, i.e. $M^{(5)} = \Psi(M^{(0)})$.

We view each $M^{(l)}$ to be a 4×4 matrix of bytes. Thus $b_{i,j}^{(l)}$ denotes the (i,j)th byte of $M^{(l)}$, $0 \leq i,j \leq 3$ and $0 \leq l \leq 5$. Each byte $b_{i,j}^{(l)}$ consists of eight bits. The kth bit of $b_{i,j}^{(l)}$ will be denoted by $b_{i,j,k}^{(l)}$.

The function byteSub() maps eight bits to eight bits. Thus the output of byteSub() consists of eight component Boolean functions each of which take eight bits as input and produce one bit as output. We denote these eight functions by $g_0, \ldots, g_7$.

Our first task is to show how any bit of $M^{(5)}$ depends on all the bits of $M^{(0)}$. To do this, we need to obtain some intermediate results on the effect of the various transformation that constitute Ψ. The proof of the following result is essentially a verification from the respective operations.

Proposition 2 *The following statements hold for $0 \leq i, j \leq 3$ and $0 \leq k \leq 7$. Arithmetic on i, j is computed modulo 4 and arithmetic on k is computed modulo 8.*

1. $b^{(5)}_{i,j,k} = g_k(b^{(4)}_{i,j,0}, \ldots, b^{(4)}_{i,j,7})$.
2. $b^{(4)}_{i,j,k} = b^{(3)}_{k-4\lfloor k/4 \rfloor, j, i+4\lfloor k/4 \rfloor}$.
3. $b^{(3)}_{i,j,k} = b^{(2)}_{i,j+i+\lfloor k/4 \rfloor, k+4}$.
4. $b^{(2)}_{i,j,k} = b^{(1)}_{i+1,j,k} \oplus b^{(1)}_{i+2,j,k} \oplus b^{(1)}_{i+2,j,k}$.
5. $b^{(1)}_{i,j,k} = g_k(b^{(0)}_{i,j,0}, \ldots, b^{(0)}_{i,j,7})$.

Proof. Items (1) and (5) are similar and hence we only prove Item (1). The matrix $M^{(5)}$ is obtained from $M^{(4)}$ using the NLSub() operation, which consists of applying the byteSub() function to each byte of $M^{(4)}$. Thus $b^{(5)}_{i,j} = \mathsf{byteSub}(b^{(4)}_{i,j})$. Since g_k is the kth component function of byteSub(), the kth bit of $b^{(5)}_{i,j}$, i.e., $b^{(5)}_{i,j,k} = g_k(b^{(4)}_{i,j}) = g_k(b^{(4)}_{i,j,0}, \ldots, b^{(4)}_{i,j,7})$.

Matrix $M^{(4)}$ is obtained from $M^{(3)}$ by the partial transpose operation which is an idempotent operation, i.e., both $M^{(4)} = \mathsf{FastTranspose}(M^{(3)})$ and $M^{(3)} = \mathsf{FastTranspose}(M^{(4)})$ hold. The effect of FastTranspose() on a bit $b^{(3)}_{i,j,k}$ is described as follows. If $k < 4$, then this bit becomes $b^{(4)}_{k,j,i}$ and if $k > 4$, then this becomes $b^{(4)}_{k-4,j,i+4}$. Item 2 describes exactly this operation.

Matrix $M^{(3)}$ is obtained from $M^{(2)}$ by the row shift operation. This operation left shift row i in a circular fashion by $4 + 8i$ places. Consequently $M^{(2)}$ is obtained from $M^{(3)}$ by right shifting row i in a circular fashion by $4 + 8i$ places. Now it is not too difficult to verify that the bit $b^{(3)}_{i,j,k}$ is mapped to the bit $b^{(2)}_{i,j+i+\lfloor k/4 \rfloor, k+4}$.

Matrix $M^{(3)}$ is obtained from $M^{(2)}$ by the XOR operation and hence Item 4 follows. This completes the proof of the result. ■

Combining these operations we get the following result.

Theorem 3 *For $0 \leq i, j \leq 3$ and $0 \leq k \leq 7$ we have the following.*

$$
\begin{aligned}
b^{(5)}_{i,j,k} = g_k\ (\ & g_{i+4}(b^{(0)}_{1,j}) \oplus g_{i+4}(b^{(0)}_{2,j}) \oplus g_{i+4}(b^{(0)}_{3,j}), \\
& g_{i+4}(b^{(0)}_{0,j+1}) \oplus g_{i+4}(b^{(0)}_{2,j+1}) \oplus g_{i+4}(b^{(0)}_{3,j+1}), \\
& g_{i+4}(b^{(0)}_{0,j+2}) \oplus g_{i+4}(b^{(0)}_{1,j+2}) \oplus g_{i+4}(b^{(0)}_{3,j+2}), \\
& g_{i+4}(b^{(0)}_{0,j+3}) \oplus g_{i+4}(b^{(0)}_{1,j+3}) \oplus g_{i+4}(b^{(0)}_{2,j+3}), \\
& g_{i}(b^{(0)}_{1,j+1}) \oplus g_{i}(b^{(0)}_{2,j+1}) \oplus g_{i}(b^{(0)}_{3,j+1}), \\
& g_{i}(b^{(0)}_{0,j+2}) \oplus g_{i}(b^{(0)}_{2,j+2}) \oplus g_{i}(b^{(0)}_{3,j+2}), \\
& g_{i}(b^{(0)}_{0,j+3}) \oplus g_{i}(b^{(0)}_{1,j+3}) \oplus g_{i}(b^{(0)}_{3,j+3}), \\
& g_{i}(b^{(0)}_{0,j}) \oplus g_{i}(b^{(0)}_{1,j}) \oplus g_{i}(b^{(0)}_{2,j}) \\
).\ &
\end{aligned}
$$

We would like to show that $b^{(5)}_{i,j,k}$ depends on each of the bits $b^{(0)}_{i,j,k}$. In other words, we would like to show that each bit of the output of Ψ is non-degenerate on all the input bits. (A Boolean function $f(x_1,\ldots,x_n)$ is said to be degenerate on variable x_i, if $f(a_1,\ldots,a_{i-1},0,a_{i+1},\ldots,a_n) = f(a_1,\ldots,a_{i-1},1,a_{i+1},\ldots,a_n)$ for all choices of $(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n) \in \{0,1\}^{n-1}$. Otherwise it is non-degenerate on the variable x_i. The function f is non-degenerate if it is non-degenerate on all the variables.)

We first note that *each of the output component functions $g_0,\ldots,g_7$ of* byteSub*() are non-constant and non-degenerate Boolean functions.* This fact will be used to prove our required result.

Theorem 4 *Let $y_1,\ldots,y_{128}$ be the output bits of the nonlinear map Ψ. Then each y_i depends on all the input 128 bits.*

Proof. First note that all the 16 bytes $b^{(0)}_{i,j}$ occur in the expression for $b^{(5)}_{i,j,k}$ given in Theorem 3. The value of $b^{(5)}_{i,j,k}$ is obtained by invoking g_k on eight bits, where each of these bits is obtained as XOR of three other bits. Let us number the eight input bits of g_k as $a_0,\ldots,a_7$ corresponding to the row in which it occurs in Theorem 3. We take a closer look at how the bytes $b^{(0)}_{i,j}$ influence the a_is.

The bytes $b^{(0)}_{3,j}, b^{(0)}_{0,j+1}, b^{(0)}_{1,j+2}, b^{(0)}_{2,j+3}, b^{0}_{1,j+1}, b^{(0)}_{2,j+2}, b^{(0)}_{3,j+3}, b^{(0)}_{0,j}$ occur exactly once each in the expression for $b^{(5)}_{i,j,k}$. (The other 8 bytes occur exactly twice each.) Further, no two of these bytes occur in any of the a_is and hence it is possible to set up an 1-1 correspondence between the a_is and these bytes. Using this 1-1 correspondence we rename these bytes as $B_0,\ldots,B_7$, i.e., $B_0 = b^{(0)}_{3,j}$, $B_1 = b^{(0)}_{0,j+1}$ and so on.

Now we turn to the actual proof. We want to show that $b^{(5)}_{i,j,k}$ is non-degenerate on $b^{(0)}_{i_1,j_1,k_1}$ for all choices of i_1, j_1, k_1 with $0 \le i_1, j_1 \le 3$ and $0 \le k_1 \le 7$. There are two cases to consider.

Case $b^{(0)}_{i_1,j_j} = B_l$ for some $0 \le l \le 7$: Since g_k is non-degenerate on all variables and hence on the lth variable, there is a combination $c_0,\ldots,c_{l-1},c_{l+1},\ldots,c_7$ such that $g_k(c_0,\ldots,c_{l-1},0,c_{l+1},\ldots,c_7) \neq g_k(c_0,\ldots,c_{l-1},1,c_{l+1},\ldots,c_7)$. We first set all the bits of $b^{(0)}_{i,j}$ which are not in $B_0,\ldots,B_7$ to arbitrary values. This determines two of the bits of each a_i. Denote the XOR of these two bits by d_i. For $0 \le i \le 7$, $i \neq l$, if $d_i = c_i$, then set the bits of B_i such that the third bit of a_i evaluates to 0; and if $d_i \neq c_i$, then set the bits of B_i such that the third bit of a_i evaluates to 1. Since the functions g_i and g_{i+4} are non-constant functions, this can always be done. This ensures that the inputs to g_k other than the lth input has the desired values $c_0,\ldots,c_{l-1},c_{l+1},\ldots,c_7$. Now consider the lth input. The bit d_l has already been fixed. The other bit which determines the value of a_l is obtained by invoking one of g_i or g_{i+4} on the byte B_l. Let us call this function g_p. Since g_p is a non-degenerate function, it is also non-degenerate on the k_1th input. The bit $b^{(0)}_{i_1,j_1,k_1}$ is the k_1th bit of B_l. Hence there is a combination $e_1,\ldots,e_{k_1-1},e_{k_1+1},\ldots,e_7$ such that $g_p(e_1,\ldots,e_{k_1-1},0,e_{k_1+1},\ldots,e_7) \neq g_p(e_1,\ldots,e_{k_1-1},1,e_{k_1+1},\ldots,e_7)$. Thus, fixing the other bits of B_l to the values $e_1,\ldots,e_{k_1-1},e_{k_1+1},\ldots,e_7$ and toggling just the value of $b^{(0)}_{i_1,j_1,k_1}$ we can toggle the value of d_l and consequently toggle the value of the output of g_k. This in turn means that by toggling only the value of $b^{(0)}_{i_1,j_1,k_1}$ and keeping all other bits to some fixed values we can toggle the output of g_k. Thus $b^{(5)}_{i,j,k}$ is non-dengerate on the bit $b^{(0)}_{i_1,j_1,k_1}$ as required.

Case $b^{(0)}_{i_1,j_j} \neq B_l$ for all $0 \le l \le 7$: The analysis of this case is similar to the previous case and hence we do not provide the details.

This completes the proof of the theorem. ■

Theorem 4 assures us that the mixing done in HBB is total and hence it seems improbable that low diffusion attacks can be applied to HBB. In the following we will use the notion of bias of a binary random variable. Let X be a binary random variable. We define the bias of X to be $|\Pr[X = 0] - (1/2)|$.

Theorem 5 *Let $x_1, \ldots, x_{128}$ be the input 128 bits of Ψ and $y_1, \ldots, y_{128}$ be the output 128 bits of Ψ. Let $\alpha \in \{0,1\}^{128}$ and $\beta \in \{0,1\}^{128}$ be nonzero binary strings. Then*

$$\left| \Pr[\langle \alpha, (x_1, \ldots, x_{128}) \rangle = \langle \beta, (y_1, \ldots, y_{128}) \rangle] - \frac{1}{2} \right| \leq \frac{1}{2^{13}}. \tag{2}$$

Proof. The result is obtained from a property of the function byteSub(). It is known [9, 5] that the absolute value of the bias of any nontrivial linear combination of the input and output bits of byteSub() is at most 2^{-4}. This fact and the Piling-Up Lemma (PUL) is used to prove the result.

The essential structure of Ψ is the following. First one round of NLSub() is applied. This is followed by a mixing of the columns of NLC. Then a bit permutation (row shift followed by partial transpose) is applied to NLC which is again followed by an invocation of NLSub(). The mixing of the columns of NLC ensures that each bit after mixing depends upon exactly three bits before mixing. These input three bits are obtained by three separate invocations of byteSub() on three distinct 8-bit strings. Also the output "mixed" bit is subjected to one invocation of byteSub(). Hence any linear combination λ of the input and output of Ψ can be broken up into a linear combination of at least 4 random variables T_1, T_2, T_3 and T_4, where each T_i is a linear combination of the inputs and outputs of one invocation of byteSub(). Further, the invocations of byteSub() for the different T_i's are different. From the property of byteSub(), the bias of each T_i is at most 2^{-4}. Assuming that the T_i's are independent we apply the PUL to obtain the result that the bias of λ is at most $2^3 \times 2^{-16} = 2^{-13}$. ■

Theorem 5 provides a bound on the correlation probability of the best affine approximation of the nonlinear map Ψ. This bound may be further improved by introducing additional layers of NLSub() and an affine transformation. Suppose that the following lines are inserted between Steps 5 and 6 of the function Round().

```
5a. for $i = 1$ to NMIX
5b.         $\Delta = \mathsf{NLC}_0 \oplus \mathsf{NLC}_1 \oplus \mathsf{NLC}_2 \oplus \mathsf{NLC}_3$, where $|\mathsf{NLC}_i| = 32$;
5c.         for $j = 0$ to 3 do $\mathsf{NLC}_i = (\Delta \oplus \mathsf{NLC}_i) \lll (8 * i + 4)$;
5d.         NLC = NLSub(NLC);
5e. enddo;
```

This will introduce an additional NMIX layers of NLSub() and the affine transformation. The effect of this will be to lower the bias even further. The bias will decrease as NMIX increases before finally plateauing off. However, the downside will be more operations per 128 bits of the key stream. The new lines will require $11 * \mathsf{NMIX}$ bitwise operations and $16 * \mathsf{NMIX}$ look-ups. We do not suggest these lines as part of HBB since we believe that a bias of 2^{-13} is low enough to resist cryptanalytic attacks. Hence the slowdown associated with positive values of NMIX can be avoided. However, a user might choose a small value of NMIX without affecting the speed too much.

5.2 Self-Synchronizing Mode

The **SS** mode is a self synchronizing mode. This mode is useful in error prone communication channels. In this mode, the next 128 bits of the key stream depend upon the secret key KEY and the previous four 128-bit cipher blocks. Whenever the receiver correctly receives four consecutive 128-bit blocks of the cipher the next 128 bits of the key stream are correctly generated. From this point onwards as long as the cipher blocks are correctly received, the receiver will be able to correctly generate the key stream. In the **SS** mode the linear part of the state memory LC is updated after generating each 128-bit key block. This frequent re-initialization can be costly, especially in hardware. It is possible to modify the description such that the re-initialization is done after longer intervals. The corresponding synchronization scheme will be different and the exact details will depend on the chosen error model. We leave this as future work.

6 Variability

We briefly discuss the features of HBB which can be varied without potentially changing the security of the system.

1. The choice of the two 256-bit primitive polynomials can be changed. These polynomials were randomly generated and the only consideration was the weight of the polynomials. Thus any two primitive polynomials of degree 256 and weight 129 should be sufficient. Also the weight need not exactly be 129; a value somewhere near 129 should suffice.
2. The choice of the irreducible polynomial to realise $GF(2^8)$ for the map byteSub() does not affect security. This polynomial can be changed.
3. The expansion function $E()$ has been chosen to output a balanced string. There are many other possible choices of $E()$ which will achieve the same effect.

Any one of the above change will change the generated key stream. Hence users who prefer customized systems may incorporate one or more of the above changes to obtain their "unique" system.

7 Conclusion

In this paper we have described a new stream cipher HBB. Compared to existing ciphers, the new cipher has both advantages and disadvantages. For example, it is slower in software implementation, but requires lesser memory. Hardware implementation is expected to be comparable if not more efficient. Further, HBB provides a good combination of security features. However, the actual security of HBB can be judged only after the research community has carefully analysed it and explored possible weaknesses.

Acknowledgements : Thanks to Charanjit Jutla for reading the paper and to Vincent Rijmen and Bill Millan for some comments. Thanks also to Sibsankar Haldar for going through the code of one of the implementations of HBB. Finally, thanks to Chandan Biswas for providing the figure of Hiji-bij-bij.

References

[1] V. V. Chepyzhov, T. Johansson and B. Smeets. A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers. *Proceedings of FSE 2000*, 181-195.

[2] D. Coppersmith, S. Halevi and C. Jutla. Cryptanalysis of stream ciphers with linear masking, *Proceedings of Crypto 2002.*

[3] D. Coppersmith, S. Halevi and C. Jutla. Scream: a software efficient stream cipher, *Proceedings of Fast Software Encryption 2002*, LNCS vol. 2365, 195-209.

[4] N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback, *Eurocrypt 2003*, pp 345-359.

[5] J. Daemen and V. Rijmen. *The design of Rijndael.* Springer Verlag Series on Information Security and Cryptography, 2002. ISBN 3-540-42580-2.

[6] P. Ekdahl and T. Johansson. SNOW - a new stream cipher. *Proceedings of SAC, 2002.*

[7] J. Dj. Golic. Modes of Operation of Stream Ciphers. *Proceedings of Selected Areas in Cryptography 2000*, pp 233-247.

[8] C. S. Jutla. Encryption Modes with Almost Free Message Integrity. *Proceedings of EUROCRYPT 2001*, 529-544.

[9] K. Nyberg. Differentially Uniform Mappings for Cryptography. *Proceedings of EUROCRYPT 1993*, 55-64.

[10] G. Rose and P. Hawkes. Turing, a high performance stream cipher. *Proceedings of Fast Software Encryption, 2003*, to appear. Also available as IACR technical report, http://eprint.iacr.org, number 2002/185.

[11] P. Sarkar. The filter-combiner model for memoryless synchronous stream ciphers. In *Proceedings of Crypto 2002*, Lecture Notes in Computer Science.

[12] P. Sarkar. Computing shifts in 90/150 cellular automata sequences. *Finite Fields and their Applications*, Volume 9, Issue 2, April 2003, pp 175-186.

[13] S. Tezuka and M. Fushimi. A method of designing cellular automata as pseudo random number generators for built-in self-test for VLSI. In *Finite Fields: Theory, Applications and Algorithms*, Contemporary Mathematics, AMS, 363–367, 1994.

[14] D. Watanabe, S. Furuya, H. Yoshida and B. Preneel. A new keystream generator MUGI. *Fast Software Encryption, 2002*, LNCS 2365, Springer, 179-194.

[15] M. Zhang, C. Caroll and A. Chan. The software-oriented stream cipher SSC2. *Fast Software Encryption, 2000*, LNCS 1978, Springer 2001, 31-48.

Figure 1: Hiji-bij-bij

A Who is Hiji-bij-bij?

Hiji-bij-bij is a character in the story Ha-ja-ba-ra-la, a brilliant creation by Sukumar Ray[1]. Ha-ja-ba-ra-la is written in Bangla[2] and falls in the class of nonsense literature along the lines of "Alice's Adventures in Wonderland" by Lewis Carroll. The story is about a small boy's dream wanderings, where he interacts with "unreal" characters. The tale is told in an inimitable manner and is full of satire. Hiji-bij-bij (see Figure 1 for the original sketch of Hiji-bij-bij by its creator) is one of the characters that the boy meets and whose speciality is to laugh until his sides burst at absurd jokes that he himself is constantly cooking up. The word Hiji-bij-bij was probably intended as a pun on the Bangla word hiji-biji, which means nonsense scribblings, especially that of a child. This is perhaps an appropriate name for a stream cipher, which is supposed to produce random "nonsense".

B Software Implementation

A somewhat efficient implementation of HBB is given below. The author is not an expert on software implementation and faster implementations may be possible.

```
/*******************************************************************************************/
/*     An implementation of the basic mode of operation of Hiji-bij-bij.               ***/
/*     usage: <command> <# of key blocks> <key file name>                              ***/
/*     key file name must have two lines: The first line contains the length (in bits)   ***/
/*     of the key and the second line contains the required number of longs (in hex).    ***/
/*     compile:                                                                          ***/
/*              gcc -o HBB -O3 -funroll-loops -fstrength-reduce HBB.c                    ***/
/*******************************************************************************************/
#include <stdio.h>
#include <string.h>
#include <time.h>

/************************** start of constants ******************************/
#define RULE00 0x80ffaf46
#define RULE01 0x977969e9
```

[1]He is the father of Satyajit Ray, a noted Indian film director.

[2]One of the top ten most spoken languages in the world

```
#define RULE02 0x71553bb5
#define RULE03 0x99be6b2b
#define RULE04 0x4b337295
#define RULE05 0x2308c787
#define RULE06 0xb84c7cce
#define RULE07 0x36d501e6

#define RULE10 0xdd18c62b
#define RULE11 0x153df31a
#define RULE12 0xc98e86c1
#define RULE13 0x910fee24
#define RULE14 0x2942d51b
#define RULE15 0x4201eb3d
#define RULE16 0xc1d1a85f
#define RULE17 0x57b8919b

#define MASK0 (0x55555555)
#define MASK1 (0x33333333)

/******************************************************************************/

/************************** start of macros **********************************/

#define FOLD \
{\
\
if (KEYLEN == 256) { \
tmp0 = KEY[0] ^ KEY[2] ^ KEY[4] ^ KEY[6]; \
tmp1 = KEY[1] ^ KEY[3] ^ KEY[5] ^ KEY[7]; \
NLC0.word = tmp0;  NLC1.word = tmp1; \
NLC2.word = ~tmp0; NLC3.word = ~tmp1; \
}\
else { \
tmp0 = KEY[0] ^ KEY[2]; \
tmp1 = KEY[1] ^ KEY[3]; \
NLC0.word = tmp0;  NLC1.word = tmp1; \
NLC2.word = ~tmp0; NLC3.word = ~tmp1; \
}\
}

#define EXP \
{ \
\
if (KEYLEN == 256) { \
state00 = KEY[0]; state01 = KEY[1]; state02 = KEY[2]; state03 = KEY[3]; \
state04 = KEY[4]; state05 = KEY[5]; state06 = KEY[6]; state07 = KEY[7]; \
state10 = ~KEY[0]; state11 = ~KEY[1]; state12 = ~KEY[2]; state13 = ~KEY[3]; \
state14 = ~KEY[4]; state15 = ~KEY[5]; state16 = ~KEY[6]; state17 = ~KEY[7]; \
} \
else { \
state00 = state14 = KEY[0]; state01 = state15 = KEY[1]; \
state02 = state16 = KEY[2]; state03 = state17 = KEY[3]; \
state04 = state10 = ~KEY[0]; state05 = state11 = ~KEY[1]; \
state06 = state12 = ~KEY[2]; state07 = state13 = ~KEY[3]; \
}\
}

#define CLShift(A,I)  (((A) >> (32 - I)) ^ ((A) << (I)))

#define EvolveCA256(ca) \
{\
\
tmp0 = ((state##ca##0 << 1) ^ (state##ca##1 >> 31)) \
      ^ (RULE##ca##0 & state##ca##0) \
                      ^ (state##ca##0 >> 1);\
        tmp1 = ((state##ca##1 << 1) ^ (state##ca##2 >> 31)) \
                      ^ (RULE##ca##1 & state##ca##1)                \
                      ^ ((state##ca##1 >> 1) ^ (state##ca##0 << 31));\
        tmp2 = ((state##ca##2 << 1) ^ (state##ca##3 >> 31)) \
                      ^ (RULE##ca##2 & state##ca##2) \
                      ^ ((state##ca##2 >> 1) ^ (state##ca##1 << 31));\
        tmp3 = ((state##ca##3 << 1) ^ (state##ca##4 >> 31)) \
                      ^ (RULE##ca##3 & state##ca##3) \
                      ^ ((state##ca##3 >> 1) ^ (state##ca##2 << 31));\
        tmp4 = ((state##ca##4 << 1) ^ (state##ca##5 >> 31)) \
                      ^ (RULE##ca##4 & state##ca##4) \
                      ^ ((state##ca##4 >> 1) ^ (state##ca##3 << 31));\
        tmp5 = ((state##ca##5 << 1) ^ (state##ca##6 >> 31)) \
                      ^ (RULE##ca##5 & state##ca##5) \
                      ^ ((state##ca##5 >> 1) ^ (state##ca##4 << 31));\
        tmp6 = ((state##ca##6 << 1) ^ (state##ca##7 >> 31)) \
                      ^ (RULE##ca##6 & state##ca##6) \
                      ^ ((state##ca##6 >> 1) ^ (state##ca##5 << 31));\
        tmp7 = (state##ca##7 << 1) \
                      ^ (RULE##ca##7 & state##ca##7) \
                      ^ ((state##ca##7 >> 1) ^ (state##ca##6 << 31));\
state##ca##0 = tmp0; state##ca##1 = tmp1; state##ca##2 = tmp2; state##ca##3 = tmp3; \
state##ca##4 = tmp4; state##ca##5 = tmp5; state##ca##6 = tmp6; state##ca##7 = tmp7; \
}

#define transpose32(arr0,arr1,arr2,arr3) \
{\
\
tmp0 = arr0 & MASK0; tmp1 = arr0 & ~MASK0; tmp2 = arr1 & MASK0; tmp3 = arr1 & ~MASK0;\
tmp0 = tmp0 << 1; tmp3 = tmp3 >> 1; arr0 = tmp1 ^ tmp3; arr1 = tmp0 ^ tmp2;\
\
tmp0 = arr2 & MASK0; tmp1 = arr2 & ~MASK0; tmp2 = arr3 & MASK0; tmp3 = arr3 & ~MASK0;\
tmp0 = tmp0 << 1; tmp3 = tmp3 >> 1; arr2 = tmp1 ^ tmp3; arr3 = tmp0 ^ tmp2;\
\
tmp0 = arr0 & MASK1; tmp1 = arr0 & ~MASK1; tmp2 = arr2 & MASK1; tmp3 = arr2 & ~MASK1;\
tmp0 = tmp0 << 2; tmp3 = tmp3 >> 2; arr0 = tmp1 ^ tmp3; arr2 = tmp0 ^ tmp2;\
\
tmp0 = arr1 & MASK1; tmp1 = arr1 & ~MASK1; tmp2 = arr3 & MASK1; tmp3 = arr3 & ~MASK1;\
tmp0 = tmp0 << 2; tmp3 = tmp3 >> 2; arr1 = tmp1 ^ tmp3; arr3 = tmp0 ^ tmp2;\
}
```

```
#define Round \
{ \
\
NLSub; \
tmp0 = NLC0.word ^ NLC1.word ^ NLC2.word ^ NLC3.word;\
NLC0.word = CLShift((tmp0^NLC0.word),4);\
NLC1.word = CLShift((tmp0^NLC1.word),12);\
NLC2.word = CLShift((tmp0^NLC2.word),20);\
NLC3.word = CLShift((tmp0^NLC3.word),28);\
transpose32(NLC0.word,NLC1.word,NLC2.word,NLC3.word);\
NLSub;\
EvolveCA256(0);\
EvolveCA256(1);\
KEYSTREAM0 = NLC0.word ^ state00;\
KEYSTREAM1 = NLC1.word ^ state07;\
KEYSTREAM2 = NLC2.word ^ state10;\
KEYSTREAM3 = NLC3.word ^ state17;\
NLC0.word = NLC0.word ^ state03;\
NLC1.word = NLC1.word ^ state04;\
NLC2.word = NLC2.word ^ state13;\
NLC3.word = NLC3.word ^ state14;\
}

#define NLSub \
NLC0.bytes.byte0 = byteSub[NLC0.bytes.byte0];\
NLC0.bytes.byte1 = byteSub[NLC0.bytes.byte1];\
NLC0.bytes.byte2 = byteSub[NLC0.bytes.byte2];\
NLC0.bytes.byte3 = byteSub[NLC0.bytes.byte3];\
NLC1.bytes.byte0 = byteSub[NLC1.bytes.byte0];\
NLC1.bytes.byte1 = byteSub[NLC1.bytes.byte1];\
NLC1.bytes.byte2 = byteSub[NLC1.bytes.byte2];\
NLC1.bytes.byte3 = byteSub[NLC1.bytes.byte3];\
NLC2.bytes.byte0 = byteSub[NLC2.bytes.byte0];\
NLC2.bytes.byte1 = byteSub[NLC2.bytes.byte1];\
NLC2.bytes.byte2 = byteSub[NLC2.bytes.byte2];\
NLC2.bytes.byte3 = byteSub[NLC2.bytes.byte3];\
NLC3.bytes.byte0 = byteSub[NLC3.bytes.byte0];\
NLC3.bytes.byte1 = byteSub[NLC3.bytes.byte1];\
NLC3.bytes.byte2 = byteSub[NLC3.bytes.byte2];\
NLC3.bytes.byte3 = byteSub[NLC3.bytes.byte3];\

/****************** end of macros ************************************/

typedef union {
unsigned long word;
struct {
unsigned char byte0,byte1,byte2,byte3;
} bytes;
} NLCWORD;

int KEYLEN; /* either 256 or 512 */
unsigned long KBLK; /* number of keystream blocks to be generated */
unsigned long KEY[8]; /* secret key read from file */
```

```
/*  the S-box */
static const unsigned char byteSub[256] =
{
0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
0xca,0x82,0xc9,0x7b,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
0x53,0xd1,0x00,0xed,0x20,0xfc,0xd1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};
/**************/

main(int argc, char *argv[]) {

register NLCWORD NLC0,NLC1,NLC2,NLC3; /* nonlinear core */

/* linear core -- state vectors for the two CA (numbered 0 and 1) */
register unsigned long state00=0, state01=0, state02=0, state03=0;
register unsigned long state04=0, state05=0, state06=0, state07=0;
register unsigned long state10=0, state11=0, state12=0, state13=0;
register unsigned long state14=0, state15=0, state16=0, state17=0;
/*************************************************/
register unsigned long KEYSTREAM0,KEYSTREAM1,KEYSTREAM2,KEYSTREAM3;
/* 128 bits of the keystream */
register unsigned long tmp0=0,tmp1=0,tmp2=0,tmp3=0,tmp4=0,tmp5=0,tmp6=0,tmp7=0;

unsigned long T[4][4],C[4],M[4];
int i;
clock_t t1,t2;
FILE *fp;

if (argc < 3) {printf("Usage: %s <# of key blocks> <key file name>\n",argv[0]); exit(0);}
KBLK = atol(argv[1]); printf("KBLK = %ld\n", KBLK);
fp = fopen(argv[2],"r");
if (fp == NULL) {printf("Cannot open %s\n",argv[2]); exit(0);}
fscanf(fp,"%d\n",&KEYLEN);
if ((KEYLEN != 128)&&(KEYLEN != 256)) {printf("invalid length of secret key\n"); exit(0);}
for(i=0;i<KEYLEN/32;i++) fscanf(fp,"%x",&KEY[i]);
KEYSTREAM0 = KEYSTREAM1 = KEYSTREAM2 = KEYSTREAM3 = 0;
/********* start of key setup ***************/
EXP FOLD
for(i=0;i<=12;i++) Round
T[0][0] = KEYSTREAM0; T[0][1] = KEYSTREAM1; T[0][2] = KEYSTREAM2; T[0][3] = KEYSTREAM3;
Round
T[1][0] = KEYSTREAM0; T[1][1] = KEYSTREAM1; T[1][2] = KEYSTREAM2; T[1][3] = KEYSTREAM3;
Round
T[2][0] = KEYSTREAM0; T[2][1] = KEYSTREAM1; T[2][2] = KEYSTREAM2; T[2][3] = KEYSTREAM3;
Round
T[3][0] = KEYSTREAM0; T[3][1] = KEYSTREAM1; T[3][2] = KEYSTREAM2; T[3][3] = KEYSTREAM3;
state00 = state00 ^ T[3][0]; state01 = state01 ^ T[3][1];
state02 = state02 ^ T[3][2]; state03 = state03 ^ T[3][3];
state04 = state04 ^ T[2][0]; state05 = state05 ^ T[2][1];
state06 = state06 ^ T[2][2]; state07 = state07 ^ T[2][3];
state10 = state10 ^ T[1][0]; state11 = state11 ^ T[1][1];
state12 = state12 ^ T[1][2]; state13 = state13 ^ T[1][3];
state14 = state14 ^ T[0][0]; state15 = state15 ^ T[0][1];
state16 = state16 ^ T[0][2]; state17 = state17 ^ T[0][3];
/********* end of key setup ****************/

t1 = clock();
/********* start of actual key generation **********/
for(i=0;i<KBLK;i++) {
Round /* actual key generation; */
/********* start of simulation of encryption *************/
C[0] = M[0] ^ KEYSTREAM0;
C[1] = M[1] ^ KEYSTREAM1;
C[2] = M[2] ^ KEYSTREAM2;
C[3] = M[3] ^ KEYSTREAM3;
/********* end of simulation of encryption *************/
}
/********* end of actual key generation **********/
t2 = clock();
printf("CLK_TCK = %d, CLOCKS_PER_SEC = %d\n", CLK_TCK, CLOCKS_PER_SEC);
printf("clocks elapsed = %d, time elapsed = %f\n",
t2-t1,(float)(t2-t1)/(float)CLOCKS_PER_SEC);
}
```