Hiji-bij-bij : A New Stream Cipher with Self-Synchronizing and MAC Modes of Operation

Palash Sarkar
Cryptology Research Group
Applied Statistics Unit
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 avoids finite field arithmetic. 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 two additional modes of operation – message authentication code (**MAC**) and self synchronizing (**SS**) modes. In the **MAC** mode, the system performs encryption and also generates a MAC "almost for free". 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, message authentication code, 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 recent years 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]) on the memoryless combiner model cannot be directly applied. Some recent block oriented ciphers are SNOW [8], SCREAM [3], MUGI [16] and SSC2 [17] 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 [8] 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. Further, there are design similarities with the stream cipher PANAMA [5]. The innovation we introduce is in the design of the linear and the nonlinear map.

The nonlinear map avoids finite field arithmetic and works only with the bit strings. 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 [13, 14]). 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 identify two additional modes of operation – the message authentication code (**MAC**) and the self-synchronizing (**SS**) modes. In the **MAC** mode, the cipher performs encryption *and* produces a MAC. The generation of the MAC is "almost for free" as in the block cipher encryption mode introduced by Jutla [10]. 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. It is also possible to introduce a more general **IV** mode of operation as in SNOW [8]. We leave this task as future work. See also [9] for certain stream cipher modes of operation.

The speed of software implementation is between 1 and 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 40% 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 since HBB avoids finite field computations, it should be more efficient to implement it in hardware. 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. Finally, Section 7 concludes the paper.

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.

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

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 [14] for an algebraic analysis of CA sequences.

3 HBB Specification

We identify three modes of operation for HBB: Basic (**B**), Message Authentication Code (**MAC**) and Self Synchronizing (**SS**) mode. We use a variable MODE to indicate the mode of operation (**B, MAC** 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 3 do (T_i, LC, NLC) = Round(LC, NLC);
3. $\mathsf{LC}_{-1} = \mathsf{LC} \oplus (T_3||T_2||T_1||T_0)$; NLC_{-1} = NLC; $C_{-3} = C_{-2} = C_{-1} = 0^{128}$;
4. for $i = 0$ to $n-1$ do
5. $(K_i, \mathsf{LC}_i, \mathsf{NLC}_i) = \mathsf{Round}(\mathsf{LC}_{i-1}, \mathsf{NLC}_{i-1})$;
6. $C_i = M_i \oplus K_i$;
7.   Case (MODE)
8.   MODE = **MAC** : $\mathsf{NLC}_i = \mathsf{NLC}_i \oplus M_i$;
9.   MODE = **SS**   : $\mathsf{LC}_i = \mathsf{Exp}(\mathsf{KEY}) \oplus (C_i||C_{i-1}||C_{i-2}||C_{i-3})$;
$\mathsf{NLC}_i = \mathsf{Fold}(\mathsf{KEY}, 128) \oplus C_i \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3}$;
10.   end case;
11. enddo;
12. output $C_0||\ldots||C_{n-1}$;
13. if (MODE = **MAC**)
14. $(*, *, \mathsf{MAC}) = \mathsf{Round}(\mathsf{LC}_{n-1}, \mathsf{NLC}_{n-1})$;
15.   output MAC;
16. endif.

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 computed MAC should equal the received MAC.

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 Exp(KEY) = KEY$||\overline{\text{KEY}}$; and if the key size is 128 bits, then Exp(KEY) = KEY$||\overline{\text{KEY}}||\overline{\text{KEY}}||$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 **MAC** mode the cipher performs encryption and also produces an 128-bit MAC. Note that the generation of the MAC is "almost for free" – a feature which is similar to the IACBC mode proposed by Jutla in [10].

3. In a manner similar to that of SNOW [8], it is possible to define a more general mode of operation – the **IV** mode. This mode uses frequent rekeying of the cipher based on a initialization vector. An application of such a mode is to "jump" to arbitrary points of the keystream. Also the **SS** mode can be considered to be a special case of **IV** mode.

4. 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].

Round(LC, NLC)

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

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 while all other operations treat 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 [6]; which is the inverse mapping over $GF(2^8)$ followed by an affine transformation over $GF(2)^8$. (See [12] for details of the inverse map and [6] 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 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 $0 \leq i < s$. 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 $r \leq s$. 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.

To complete the description of HBB() we have to define the function NextState(). This function is a linear map from 512-bit string 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.

$$(a_0 \dots a_{255})_{16} = \texttt{f9169344ec191960a5bc37331451501906cad5d1663e2bb7d3cd5aaa75d7cca7}$$
$$(b_0 \dots b_{255})_{16} = \texttt{f36f6b011149b6bc986b889d3c224e102e67793a98cf32d94c8de2567aacf82f}$$

The 90/150 rule vector of the two CA corresponding to these two primitive polynomials is found using the algorithm of Tezuka and Fushimi [15] and in hex form is given below.

$$\mathcal{R}_0 : \texttt{2d240f0e5308f30bd460bab9265cffd11279819e92dc69a50b9da4c018b274d5}$$
$$\mathcal{R}_1 : \texttt{91070f8787e737b546f6934aa14b3f26bc87113e6a2e8096da0bd5e7f34e718c}$$

The function EvolveCA() is simple to describe.

EvolveCA(state, rule)
1. state $= (\text{state} \ll 1) \oplus (\text{rule} \wedge \text{state}) \oplus (\text{state} \gg 1)$;
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 Time and Memory Requirements

The total state memory of HBB is 640=(512+128) bits which is 96 bytes. The implementation of byteSub() takes 256 bytes. Some additional 32-bit temporary variables/constants are required; the total memory requirement for variables/constants is around 400 bytes. For fast software implementation, the loops in the subroutines will be "unfolded". The size of the resulting code will be small (around 200 bytes). Hence the whole program and data will comfortably fit into around 650 bytes of memory. We note that the software implementation of both SNOW and SCREAM require 4 look-up tables of size 256 bytes each. Hence the total size of the look-up tables alone is 1 Kbyte. Clearly HBB is smaller.

The speed of implementation in software is slower than SNOW or SCREAM. The speed is between 1 to 2 operations per bit. 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. In software, this operation is relatively expensive and requires almost 40% of the total time. This time can be considerably reduced by using an LFSR over $GF(2^{32})$ to implement LC (as in SNOW). However, we do not do this for primarily two reasons – the hardware implementation of CA is likely to be more efficient; secondly we believe that CA provides a better quality pseudorandom sequence.

The description of HBB is simple and avoids finite field arithmetic and addition modulo 2^{32}. The only place where finite fields occur is in the byteSub() function which in any case is implemented as a look-up table. In fact, at no point of the algorithm do we need to actually use the irreducible polynomial which is used to define $GF(2^8)$. Due to these reasons the hardware implementation of HBB is expected to be more efficient and a throughput of 32 bits per clock cycle looks feasible. However, only detailed hardware design and implementation can properly answer this question.

5 Design Justification

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 **MAC** and the **SS** modes of operation.

5.1 Mixing and Bias

Steps 1 to 5 of the function Round() define a nonlinear map $\Psi : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$. The function Ψ is a nonlinear permutation of the input 128 bits. Let $M^{(0)}$ be the state of NLC before step 1 of Round() and for $1 \leq i \leq 5$, let $M^{(i)}$ be the state of NLC after step i of Round(). (Note that $M^{(1)} = M^{(2)}$.) We consider each $M^{(i)}$ to be a 4×4 matrix of bytes and also a 4×32 matrix of bits. The bytes of $M^{(i)}$ are numbered in a row major fashion and denoted $B_0^{(i)}, \ldots, B_{15}^{(i)}$. Similarly, the bits of $M^{(i)}$ are denoted $b_0^{(i)}, \ldots, b_{127}^{(i)}$. The byte $B_j^{(i)}$ consists of the bits $b_{8*j}^{(i)}, \ldots, b_{8*j+7}^{(i)}$. Given a bit b_k $(0 \leq k \leq 127)$ by $\mathsf{byte}(b_k)$ we denote the byte to which the bit b_k belongs.

Theorem 2 *Let $y_1, \ldots, y_{128}$ be the output bits of the nonlinear map Ψ. Then each y_i depends nonlinearly on all the input 128 bits.*

Proof. The output of Ψ is $M^{(5)}$ and the input is $M^{(0)}$. Since the output of Ψ is $M^{(5)}$, we have $b_k^{(5)} = y_k$ for all $0 \leq k \leq 127$. Consider any bit $b_{k_0}^{(5)}$ of $M^{(5)}$. Let $B_j = \mathsf{byte}(b_{k_0})$. Then $b_k^{(5)}$ depends nonlinearly on all bits of $B_j^{(4)} = (b_l^{(4)}, \ldots, b_{l+7}^{(4)})$, where $l = 8 * j$. The matrix $M^{(4)}$ is obtained from $M^{(3)}$ by an application of FastTranspose() which is a permutation of $\{0,1\}^{128}$. Call this permutation π and for $0 \leq i \leq 7$, let $z_i^{(3)} = \pi^{-1}(b_{l+i}^{(4)})$, where $z_i^{(3)}$ is a bit of $M^{(3)}$. It is easy to verify that $\mathsf{byte}(z_i^{(3)}) = \mathsf{byte}(z_{i+4}^{(3)})$ for $0 \leq i \leq 3$. Let $Z_i^{(3)} = \mathsf{byte}(z_i^{(3)})$. Then each $Z_i^{(3)}$ is a byte of $M^{(3)}$ and these bytes form one complete column of $M^{(3)}$ (considered as a 4×4 matrix of bytes).

The matrix $M^{(3)}$ is obtained from $M^{(2)} = M^{(1)}$ through steps 2 and 3. These two steps define a permutation of $\{0,1\}^{128}$ in the following manner. Consider $M^{(1)}$ to be a 4×4 matrix of bytes. This matrix is multiplied by $J_4 \oplus I_4$, where J_4 is the 4×4 all one matrix and I_4 is the identity matrix of order 4. Then each row of the result is cyclically shifted. The matrix $J_4 \oplus I_4$ is invertible and is its own inverse. Also the cyclic shift of each row is an invertible operation. Since both the operations are invertible, the entire step is also invertible and constitutes a permutation of $\{0,1\}^{128}$. Let $M^{'}$ be the matrix before the row shift operation. Then $M^{(3)}$ is obtained from $M^{'}$ after the row shift operation. Let us denote the permutation defined by the row shift operation to be π_1. For $0 \leq i \leq 7$, Let $w_i^{'} = \pi_1^{-1}(z_i^{(3)})$ and $W_i^{'} = \mathsf{byte}(w_i^{'})$. Then

$$\{W_0^{'}, \ldots, W_7^{'}\} = \{U_{k_1,k_2}, U_{k_1+1,k_2+1}, U_{k_1+2,k_2+2}, U_{k_1+3,k_2+3}, U_{k_1,k_2+1}, U_{k_1+1,k_2+2}, U_{k_1+2,k_2+3}, U_{k_1+3,k_2+4}\}$$

where $U_{i,j}$ is the byte in the ith row and jth column of $M^{'}$, $(0 \leq i, j \leq 4)$ and all the indices are computed modulo 4. The bit $b_{k_0}^{(5)}$ depends nonlinearly on $\{w_0', \ldots, w_7'\}$ and hence on the bytes $\{W_0^{'}, \ldots, W_7^{'}\}$.

The matrix $M^{'}$ is obtained from $M^{(1)}$ by multiplication with the matrix $J_4 \oplus I_4$. Hence $U_{i,j} = \Delta_j^{(1)} \oplus U_{i,j}^{(1)}$, where $U_{i,j}^{(1)}$ is the byte in the ith row and jth column of $M^{(1)}$ and $\Delta_j^{(1)} = \oplus_{i=0}^{3} U_{i,j}^{(1)}$. Hence the set $\{W_0^{'}, \ldots, W_7^{'}\}$ depends upon all the bytes of $M^{(1)}$. Since $M^{(1)}$ is obtained from $M^{(0)}$ by one application of NLSub() each bit $b_k^{(1)}$ of $M^{(1)}$ depends nonlinearly on all bits of byte $B_j^{(0)}$, where $B_j = \mathsf{byte}(b_k)$. Thus the set $\{W_0^{'}, \ldots, W_7^{'}\}$ depends nonlinearly on all bits of $M^{(0)}$. Hence $b_{k_0}^{(5)}$ depends nonlinearly on all the bits of $M^{(0)}$ which proves the result. ∎

Theorem 2 assures us that the mixing done in HBB is total and hence it seems improbable that low diffusion attacks can be applied to HBB. However, we note that it is desirable for Ψ to satisfy other security properties such as strict avalanche criteria and good propagation characteristics. We believe that Ψ indeed possesses such properties though at this point we are unable to prove any such result.

In the following we will use the notion of bias of a binary random variable. Let X be a binary random variable, then we define the bias of X to be $|\Pr[X = 0] - (1/2)|$.

Theorem 3 *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 [12, 6] 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 3 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 MAC and SS modes

The **MAC** mode encrypts the message and also produces a 128-bit MAC in a single pass. The generation of the MAC is "almost for free" and involves only 16 XORs per message block and two extra invocations

of the function Round() for the entire message. Let us denote the computed MAC by NLC_n. Suppose that a message block M_i changes. This changed value is then XORed with NLC_i to obtain the input to Ψ in the next round. Thus the input to Ψ in the next round changes. Since Ψ is a permutation, the output of Ψ will also change leading to a change in the value of NLC_{i+1}. Assuming that Ψ is a "good" nonlinear permutation, it does not seem possible to predict the change in NLC_{i+1} when M_i is changed. The security of the **MAC** mode crucially depends on this fact. In fact, we believe that without knowing the KEY it is not possible to produce two messages which have the same MAC. Under suitable assumptions on Ψ it might be possible to prove this statement in a theoretical framework as in [10]. We leave this task for the future.

The generated MAC is 128 bits long. This length arises naturally from the design of HBB where the nonlinear map works on 128-bit strings. Other lengths of MAC are possible by utilizing other aspects of the Round() function. Next we briefly consider the possibility of MAC generation in other stream ciphers. In SNOW, the nonlinear map works on 32-bit strings, and it should be easy to obtain a 32-bit MAC. However, this is too small for practical applications. The design strategy of SNOW does not suggest any immediate method for generating 128-bit MAC. On the other hand, it should be possible to obtain 128-bit MAC from SCREAM. The only problem is that the round function of SCREAM is quite complicated (it consists of two half round Feistel iterations) and it is difficult to determine the exact method of obtaining the MAC.

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 each 128-bit 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. Again 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. Also HBB provides a good combination of security features and offers certain additional modes of operation. However, the actual security of HBB can be judged only after the research community has carefully analysed it and explored possible weaknesses.

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, J. Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. *Proceedings of ASIACRYPT 2002*, 267-287.

[5] J. Daemen, C. S. K. Clapp. Fast Hashing and Stream Encryption with PANAMA. *Proceedings of Fast Software Encryption 1998*, pp 60-74.

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

[7] H. Dobbertin. Almost Perfect Nonlinear Power Functions on $GF(2^n)$: The Niho Case. *Information and Computation*, 151(1-2): 57-72 (1999).

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

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

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

[11] S. Murphy, M. J. B. Robshaw. Essential Algebraic Structure within the AES. *Proceedings of CRYPTO 2002*, 1-16.

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

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

[14] P. Sarkar. Computing shifts in 90/150 cellular automata sequences. *Finite Fields and their Applications*, to appear.

[15] 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.

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

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