Cryptanalysis of Hiji-bij-bij (HBB)

Vlastimil Klíma

LEC s.r.o., Národní 9, Prague, Czech Republic
v.klima@volny.cz

Abstract. In this paper, we show several known-plaintext attacks on the stream cipher HBB which was proposed recently at INDOCRYPT 2003. The cipher can operate either as a classical stream cipher in the "B mode" or as an asynchronous stream cipher in the "SS mode". In the case of the SS mode, we present known-plaintext attacks recovering 128-bit key with the complexity 2^{66} and 256-bit key with the complexity 2^{67}. In the case of B mode with 256-bit key, we show a known-plaintext attack recovering the whole plaintext with the complexity 2^{140}. All attacks need only a small part of the plaintext to be known.

Keywords: Hiji-bij-bij, HBB, stream cipher, synchronous cipher, asynchronous cipher, cryptanalysis, equivalent keys, known-plaintext attack.

1 Introduction

Hiji-bij-bij [1] is a stream cipher with 128-bit or 256-bit secret key (*KEY*) and two modes of operation. In the basic mode (B), it generates a keystream consisting of 128-bit blocks KS_0, KS_1, KS_2... which depend neither on the plaintext M nor on the ciphertext C. The keystream blocks are then XORed with the plaintext blocks (M_0, M_1, M_2...) to produce the ciphertext blocks (C_0, C_1, C_2...). In the self-synchronizing mode (SS), the scheme behaves as an asynchronous stream cipher working with complete 128-bit blocks of the keystream, the plaintext and the ciphertext. In this case, a current keystream block depends on preceding four blocks of the ciphertext. In this paper, we disclose serious weaknesses in both modes.

In the SS mode with 128-bit key, we show a known-plaintext attack. With knowledge of any plaintext block M_i for $i \geq 4$ it is possible to reveal the whole key and to decipher the whole plaintext M with the complexity of order 2^{66}.

In the SS mode with 256-bit key, we are able to decipher the whole plaintext M, except for the first four blocks, with the complexity 2^{67}. We only need to know any plaintext block M_i for $i \geq 4$ and any other 66 plaintext bits, which can be spread in several different blocks M_j for $j \geq 4$, $j \neq i$. Furthermore, if we know any 62 bits from the first four blocks of the plaintext, we are able to decipher the whole plaintext M and to derive the whole key with the complexity 2^{67}.

In the B mode with 256-bit, we are able to decipher the whole message M with the complexity 2^{140} provided that we know any 34 consequent plaintext blocks.

The basic design principle of HBB is to mix a linear and a nonlinear mapping. According to [1], the innovation of HBB is in the design of the linear and the nonlinear mappings. Our attacks show that the way in which these parts are mixed together in HBB contains structural weaknesses. We also show that the nonlinear part doesn't process the 256-bit key quite well, what creates large classes of (weakly) equivalent keys. The main drawback of the linear part is that it provides small amount of key entropy to the mix with the nonlinear part.

In Section 2, we recall necessary description of HBB. In Sections 3 and 4, we present attacks on both operation modes. In Section 5, we make some general notes on the HBB design and then we conclude in Section 6.

2 Description of HBB

HBB uses a linear combiner LC (512-bit register) and a nonlinear combiner NLC (128-bit register). They are both initialized by an encryption key (KEY, 128 or 256 bits long) and then updated regularly. Note that in the case of SS mode the update depends on a ciphertext. The inner states are then processed by linear and nonlinear functions and mixed together to produce a keystream.

2.1 Preliminaries

Here, we will work with strings of various lengths, but mainly with 32-bit strings which we call words. We use an integer in square brackets beside a variable to denote its appropriate word. The words are indexed from 0 upwards. For instance, $LC[0]$ and $LC[15]$ are the first (the leftmost) and the last (the rightmost) words of 512-bit string LC, respectively. We use notation $A \| B$ or (A, B) for a concatenation of strings A and B. Let us denote SideW(LC) and MiddleW(LC) the side and middle words of 512-bit string LC, respectively. More precisely, for $LC = LC[0] \| LC[1] \| ... \| LC[15]$ we have

SideW(LC) = $LC[0] \| LC[7] \| LC[8] \| LC[15]$ and

MiddleW(LC) = $LC[3] \| LC[4] \| LC[11] \| LC[12]$.

The expression non$X$ denotes "bit-by-bit" negation of the string X. We denote the bit length of the string X as $|X|$. $X <<< n$ means cyclic left-shift of the string X by n bits. Bits of a string are numbered from left to right starting from 1 upward and we pick them using braces. For instance, we denote the sequence of the first three bits of the string LC as $LC\{1, 2, 3\}$ whereas $LC\{512\}$ is the last bit of LC. Now, we will define the procedures and functions Exp(), Fold(), NextState(), Evolve(), FastTranspose(), Round(), NLSub(), NLF(), together with the main encryption procedure of HBB.

2.2 Exp(*KEY*) and Fold(*KEY*, *i*)

These functions expand or compress the encryption key KEY in dependence on its length. The functions are defined only for $i \in \{128, 64\}$.

For $|KEY| = 256$, $KEY = KEY[0] \,||\, KEY[1] \,||\, ... \,||\, KEY[7]$, we define

Fold(KEY, 128) = (KEY[0], KEY[1], KEY[2], KEY[3]) ⊕ (KEY[4], KEY[5], KEY[6], KEY[7]),

Fold(KEY, 64) = (KEY[0], KEY[1]) ⊕ (KEY[2], KEY[3]) ⊕ (KEY[4], KEY[5]) ⊕ (KEY[6], KEY[7]),

Exp(KEY) = KEY || nonKEY.

For $|KEY| = 128$, $KEY = KEY[0] \,||\, KEY[1] \,||\, KEY[2] \,||\, KEY[3]$, we define

Fold(KEY, 128) = KEY,

Fold(KEY, 64) = (KEY[0], KEY[1]) ⊕ (KEY[2], KEY[3]),

Exp(KEY) = KEY || nonKEY || nonKEY || KEY.

2.3 Evolve(s, C)

The function Evolve(s, C) is a linear function of the 256-bit input $s = (s_1, s_2, ..., s_{256})$, where C is a 256-bit constant C = (c_1, c_2, ..., c_{256}). Evolve(s, C) returns the 256-bit state which follows the state s with respect to a cellular automata CA. CA is defined by a tridiagonal 256 x 256 constant matrix A, where the main diagonal contains bits $A_{1,1} = c_1$, $A_{2,2} = c_2$, ..., $A_{256,256} = c_{256}$ and the first upper and lower subdiagonal entries of A are all one. All other elements are zero. Thus we have S = Evolve(s, C) = A*s. More precisely

$$\begin{aligned}
S_1 &= c_1 s_1 \oplus s_2,\\
S_2 &= s_1 \oplus c_2 s_2 \oplus s_3,\\
S_3 &= s_2 \oplus c_3 s_3 \oplus s_4,\\
&\ldots\\
S_{255} &= s_{254} \oplus c_{255} s_{255} \oplus s_{256},\\
S_{256} &= s_{255} \oplus c_{256} s_{256},
\end{aligned}$$

where ⊕ denotes addition over $\boldsymbol{Z}_2$.

2.4 NextState(LC)

This is a linear function that divides its input LC into two halves and processes them separately. The linear function Evolve(*, R_0) processes the first half and Evolve(*, R_1) processes the second one, where R_0 and R_1 are two different constants defined in [1]. More precisely, we have NextState(LC) = Evolve(LC[0] || LC[1] || ... || LC[7], R_0) || Evolve(LC[8] || LC[9] || ... || LC[15], R_1).

HBB uses NextState to update the linear combiner LC simply as LC = NextState(LC). Our attacks do not use any property of these constants. We will use intensively the fact that every i-th bit of the new state of LC depends only on the three source bits with indexes $\{i - 1, i, i + 1\}$. It is a consequence of using a cellular automata in the linear part of HBB.

2.5 FastTranspose(*NLC*)

FastTranspose(*NLC*) is carefully chosen bit permutation of 128-bit input *NLC*. Its concrete value is defined in [1], but our attacks do not use any property of it.

2.6 NLSub(*NLC*)

NLSub(*NLC*) is a substitution that is working on bytes. The input *NLC* is divided into 16 bytes that are substituted separately by an S-box. NLSub uses only one S-box which is defined in [1]. Our attacks do not use any property of it.

2.7 F(*NLC*)

The procedure F: $\{0,1\}^{128} \rightarrow \{0,1\}^{128}$ is a nonlinear mapping of 128-bit input *NLC*. Its output $Y = \mathrm{F}(NLC)$ is 128-bit string obtained by the following procedure:

```
1. $NLC = \mathrm{NLSub}(NLC)$,
2. $temp = NLC[0] \oplus NLC[1] \oplus NLC[2] \oplus NLC[3]$,
3. for $i = 0$ to 3 do $NLC[i] = (temp \oplus NLC[i]) <<< (8*i + 4)$,
4. $NLC = \mathrm{FastTranspose}(NLC)$,
5. $Y = \mathrm{NLSub}(NLC)$.
```

We will use the fact that F is highly nonlinear bijective mapping in our attacks. The basic property of HBB is that there is no useful linear approximation of F ([1], Theorem 5).

2.8 Round(*LC*, *NLC*)

The procedure Round() is used to compute the current keystream block *KS* and to update states of the combiners *LC* and *NLC*.

We define (*KS*, *LC*, *NLC*) = Round(*LC*, *NLC*) by the following procedure:

```
1. $LC = \mathrm{NextState}(LC)$,
2. $NLC = \mathrm{F}(NLC)$,
3. $KS = NLC \oplus \mathrm{SideW}(LC)$,
4. $NLC = NLC \oplus \mathrm{MiddleW}(LC)$.
```

2.9 Encryption in the Modes B and SS

In both modes, the cipher is initialized in the same way as follows:

LC = Exp(KEY), F = Fold(KEY, 64), $NLC = F \,\|\, \text{non}F$,

for i = 0 to 15 do ($T_{i \bmod 4}$, LC, NLC) = Round(LC, NLC),

$LC_{-1} = T_3 \,\|\, T_2 \,\|\, T_1 \,\|\, T_0$, $NLC_{-1} = NLC$, $C_{-3} = T_3$, $C_{-2} = T_2$, $C_{-1} = T_1$.

The plaintext message $M = M_0 \,\|\, M_1 \,\|\, \ldots \,\|\, M_{n-1}$ is encrypted to the ciphertext $C = C_0 \,\|\, C_1 \,\|\, \ldots \,\|\, C_{n-1}$ **in the B mode** as follows:

```
for $i$ = 0 to n - 1 do
{
        $(KS_i, LC_i, NLC_i)$ = Round$(LC_{i-1}, NLC_{i-1})$,
        $C_i = M_i \oplus KS_i$,
},
```

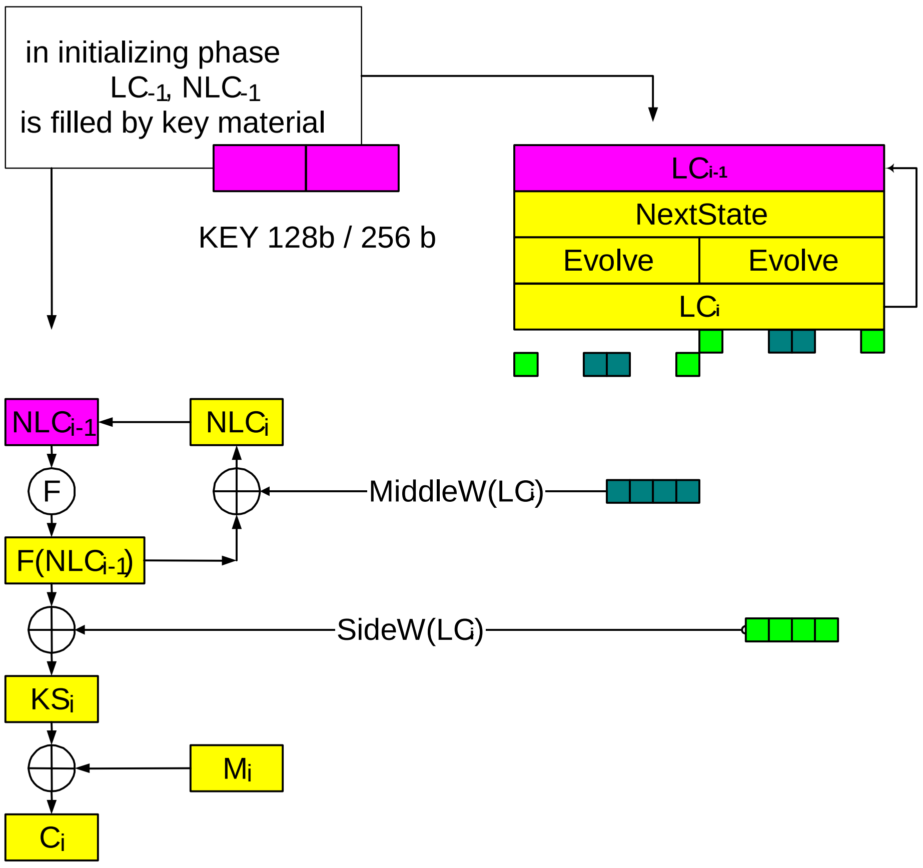

Fig.1: Encryption in the B Mode

The plaintext message $M = M_0 \| M_1 \| \ldots \| M_{n-1}$ is encrypted to the ciphertext $C = C_0 \| C_1 \| \ldots \| C_{n-1}$ **in the SS mode** as follows:

```
for $i$ = 0 to n - 1 do
{
        $(KS_i, LC_i, NLC_i) = \mathrm{Round}(LC_{i-1}, NLC_{i-1})$, note that values of $LC_i$, $NLC_i$ are re-computed in the next steps
        $C_i = M_i \oplus KS_i$,
        $LC_i = \mathrm{Exp}(KEY) \oplus (C_i, C_{i-1}, C_{i-2}, C_{i-3})$,
        $NLC_i = \mathrm{Fold}(KEY, 128) \oplus C_i \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3}$,
}.
```

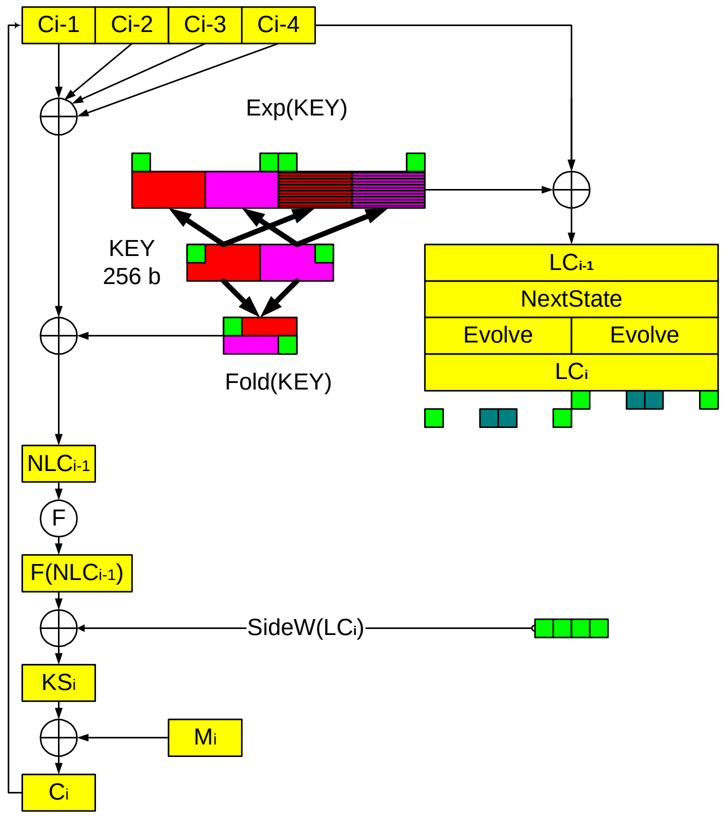

Fig.2: Encryption in the SS Mode

3 Attacks on HBB in the SS Mode

We will show attacks for both key lengths (128 and 256 bits). The attacks are formulated as propositions and their proofs describe particular procedures of the attacks.

3.1 256-bit Key

Proposition 1. *Let us assume HBB in the SS mode with 256-bit key. Furthermore, suppose knowledge of one 128-bit plaintext block M_i for some $i \geq 4$ together with additional 66 plaintext bits which can be spread in several different blocks M_j for $j \geq 4$, $j \neq i$. It is possible then to decipher the whole plaintext M, except for the first four blocks, with the complexity of order 2^{67}. Furthermore, if we know any 62 bits from the first four blocks of the plaintext, then it is possible to decipher the whole plaintext and to derive the whole key with complexity of order 2^{67}.*

Proof. We divide a proof of the proposition into several steps in the following three paragraphs. At first, we reconstruct a part of the plaintext and a part of the encryption key.

3.1.1 The Reconstruction of a Part of the Plaintext and a Part of the Key

From known M_i and C_i, we compute appropriate keystream block $KS_i = M_i \oplus C_i$. According to the definition of encryption of the block M_i for $i \geq 4$, we have

1. $LC_{i-1} = \text{Exp}(KEY) \oplus (C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})$,
2. $NLC_{i-1} = \text{Fold}(KEY, 128) \oplus (C_{i-1} \oplus C_{i-2} \oplus C_{i-3} \oplus C_{i-4})$,
3. $NLC_i = \text{F}(NLC_{i-1})$,
4. $LC_i = \text{NextState}(LC_{i-1})$,
5. $KS_i = NLC_i \oplus \text{SideW}(LC_i)$,
6. $C_i = M_i \oplus KS_i$,

where $\text{Fold}(KEY, 128) = (KEY[0] \oplus KEY[4]) \,\|\, (KEY[1] \oplus KEY[5]) \,\|\, (KEY[2] \oplus KEY[6]) \,\|\, (KEY[3] \oplus KEY[7])$ and $\text{Exp}(KEY) = KEY \,\|\, \text{non}KEY$.

Because $i \geq 4$, only known ciphertext blocks $(C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})$ are involved in the generation of KS_i, whereas the generation of $KS_0, ..., KS_3$ also involves internal variables T_0, T_1, T_2, and T_3, which are not transmitted through a public communication channel.

It follows from the definition of the function NextState that $LC_i = \text{NextState}(LC_{i-1}) = \text{NextState}(\text{Exp}(KEY) \oplus (C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})) = \text{NextState}((KEY \,\|\, \text{non}KEY) \oplus (C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})) = \text{Evolve}(KEY \oplus (C_{i-1}, C_{i-2}), \text{R}_0) \,\|\, \text{Evolve}(\text{non}KEY \oplus (C_{i-3}, C_{i-4}), \text{R}_1)$. Therefore, with respect to the secret key, $LC_i\{1, ..., 32\}$ depends only on 33 key bits $KEY\{1, ..., 32, 33\}$ and $LC_i\{225, ..., 256\}$ depends only on 33 key bits $KEY\{224, 225, ..., 256\}$.

A brute force will determine the aforesaid 66 key bits as follows: Let us guess their particular value, i.e. choose the whole words $KEY[0]$ and $KEY[7]$ and bits $KEY\{33\}$ and $KEY\{224\}$. From the preceding equation, we compute $LC_i\{1, ..., 32\}$ and $LC_i\{225, ..., 256\}$, i.e. the whole words $LC_i[0]$ and $LC_i[7]$. In the same way, we also compute the words $LC_i[8]$ and $LC_i[15]$, because they depend on the same key bits as $LC_i[0]$ and $LC_i[7]$. Thus we know the whole string $\text{SideW}(LC_i) = LC_i[0] \,\|\, LC_i[7] \,\|\, LC_i[8] \,\|\, LC_i[15]$.

Now, we compute $NLC_i = KS_i \oplus \text{SideW}(LC_i)$ and $NLC_{i-1} = \text{F}^{-1}(NLC_i)$. This computation is possible, since we know KS_i and F is an easily invertible bijection. Finally, we obtain $\text{Fold}(KEY, 128) = NLC_{i-1} \oplus (C_{i-1} \oplus C_{i-2} \oplus C_{i-3} \oplus C_{i-4})$, which is a huge amount of information about the secret key.

Now, we will show how to derive KS_j for any $j \geq 4$, $j \neq i$, i.e. how to decipher the whole message M, except for the first four blocks.

Because of the linearity of the NextState function, we have $LC_i \oplus LC_j$ = NextState(Exp(KEY) $\oplus$ ($C_{i\text{-}1}$, $C_{i\text{-}2}$, $C_{i\text{-}3}$, $C_{i\text{-}4}$)) $\oplus$ NextState(Exp(KEY) $\oplus$ ($C_{j\text{-}1}$, $C_{j\text{-}2}$, $C_{j\text{-}3}$, $C_{j\text{-}4}$)) = NextState((Exp(KEY) $\oplus$ (Exp(KEY) $\oplus$ ($C_{i\text{-}1}$, $C_{i\text{-}2}$, $C_{i\text{-}3}$, $C_{i\text{-}4}$) $\oplus$ ($C_{j\text{-}1}$, $C_{j\text{-}2}$, $C_{j\text{-}3}$, $C_{j\text{-}4}$)) = NextState(($C_{i\text{-}1}$, $C_{i\text{-}2}$, $C_{i\text{-}3}$, $C_{i\text{-}4}$) $\oplus$ ($C_{j\text{-}1}$, $C_{j\text{-}2}$, $C_{j\text{-}3}$, $C_{j\text{-}4}$)). Thus we can see that the blocks LC_i and LC_j differ only in a known value. Thus we can easily compute SideW(LC_j) from known SideW(LC_i). Recall that we also know the value of Fold(KEY, 128). Then we obtain

1. $NLC_{j\text{-}1}$ = Fold(KEY, 128) $\oplus$ ($C_{j\text{-}1} \oplus C_{j\text{-}2} \oplus C_{j\text{-}3} \oplus C_{j\text{-}4}$),
2. NLC_j = F($NLC_{j\text{-}1}$),
3. and finally $KS_j = NLC_j \oplus$ SideW(LC_j).

Now, we will decipher all those M_j for which we know some plaintext bits. Comparing known 66 plaintext bits with the deciphered ones, we accept or reject our original choice of the 66 key bits KEY\{1, ..., 33, 225, ..., 256\}.

It should suffice to compare roughly 66 bits of information of a plaintext to determine these bits unambiguously. Provided this hypothesis is true, the complexity of the attack is at maximum 2^{66} *66 HBB encryptions, while the mean value is roughly 2^{67} operations, since the false key should be rejected sooner than after 66 trials.

We will continue with a next part of the proof after presenting the following note about weakly equivalent keys..

3.1.2 Weakly Equivalent Keys

Recall that we have obtained roughly 128 bits of key information Fold(KEY, 128) from the knowledge of 66 key bits (KEY[0], KEY[7], KEY\{33\} and KEY\{224\}), and one plaintext block. Since Fold(KEY, 128) = (KEY[0] $\oplus$ KEY[4]) || (KEY[1] $\oplus$ KEY[5]) || (KEY[2] $\oplus$ KEY[6]) || (KEY[3] $\oplus$ KEY[7]), we know

1. 66 bits KEY\{1\}, KEY\{2\}, ... , KEY\{32\}, KEY\{33\} and KEY\{96\}, KEY\{97\}, ..., KEY\{128\},
2. 66 bits KEY\{129\}, KEY\{130\}, ... , KEY\{160\}, KEY\{161\} and KEY\{224\}, KEY\{225\}, ..., KEY\{256\}, and
3. 62 sums KEY\{34\} $\oplus$ KEY\{162\}, KEY\{35\} $\oplus$ KEY\{163\}, ..., and KEY\{95\} $\oplus$ KEY\{223\}.

We also saw that it was not necessary to determine the partial key bits in the last 62 sums KEY\{34\} $\oplus$ KEY\{162\}, KEY\{35\} $\oplus$ KEY\{163\}, ..., KEY\{95\} $\oplus$ KEY\{223\} to decipher the message M (except for the first four blocks). Therefore, there are at minimum 2^{62} keys, which encrypt the message M (except the first four blocks) identically. We refer these keys to as weakly equivalent keys.

3.1.3 The Reconstruction of the Rest of the Plaintext and the Rest of the Key

We will reconstruct the remaining bits of the key *KEY*{34, ..., 95} by a brute force attack, too. Let us assume knowledge of another 62 bits of the first four blocks of the plaintext and let us guess the value of *KEY*{34, ..., 95}.

Then, we easily derive the remaining key bits *KEY*{162, ..., 223} from the aforesaid knowledge of the 62 sums (c.f. Section 3.1.2). We decipher the first four plaintext blocks and compare the result with the known 62 bits of plaintext. If they do not match, we try another value of *KEY*{34, ..., 95}. If they do, we have determined the correct key bits. This part of the attack increases its complexity by an additive factor of 2^{62} HBB operations, and thus the expected complexity of the whole attack still remains roughly 2^{67} HBB operations, what completes the proof.

3.2 128-bit Key

The attack on 128-bit key is very similar to the previous one. The difference follows from different definitions of the functions Fold(*KEY*, 128) and Exp(*KEY*).

Proposition 2. *Let us assume HBB in the SS mode with 128-bit key. Furthermore, assume knowledge of any plaintext block M_i for $i \geq 4$. Then, it is possible to reveal the whole key and to decipher the whole plaintext M with complexity of order 2^{66}.*

Proof. Here we have Fold(*KEY*, 128) = *KEY* and Exp(*KEY*) = *KEY* || non*KEY* || non*KEY* || *KEY*.

From the linearity of the functions Evolve() and NextState() it follows

LC_i = NextState(LC_{i-1}) = NextState(Exp(*KEY*) $\oplus$ (C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})) = NextState((*KEY* || non*KEY* || non*KEY* || *KEY*) $\oplus$ (C_{i-1}, C_{i-2}, C_{i-3}, C_{i-4})) = NextState((*KEY* || *KEY* || *KEY* || *KEY*) $\oplus$ (C_{i-1} || nonC_{i-2} || nonC_{i-3} || C_{i-4})) = NextState(*KEY* || *KEY* || *KEY* || *KEY*) $\oplus$ NextState(C_{i-1} || nonC_{i-2} || nonC_{i-3} || C_{i-4}). Similarly, LC_j = NextState(*KEY* || *KEY* || *KEY* || *KEY*) $\oplus$ NextState(C_{j-1} || nonC_{j-2} || nonC_{j-3} || C_{j-4}). We can see that blocks LC_i and LC_j differ one from another by a known value, independent of the key.

Furthermore, we have LC_i = NextState(*KEY* || *KEY* || *KEY* || *KEY*) $\oplus$ NextState(C_{i-1} || nonC_{i-2} || nonC_{i-3} || C_{i-4}) = (Evolve(*KEY*[0] || ... || *KEY*[3] || *KEY*[0] || ... || *KEY*[3], R_0) || Evolve(*KEY*[0] || ... || *KEY*[3] || *KEY*[0] || ... || *KEY*[3], R_1)) $\oplus$ NextState(C_{i-1} || nonC_{i-2} || nonC_{i-3} || C_{i-4}). Therefore, LC_i [0] and LC_i [8] depend only on key bits *KEY*{1, ..., 33}, whereas LC_i [7] and LC_i [15] depend only on key bits *KEY*{96, ..., 128}.

Now, we will determine 66 key bits *KEY*{1, ..., 33} and *KEY*{96, ..., 128} by a brute force: Let us guess their value. Then, we can compute LC_i [0], LC_i [7], LC_i [8], LC_i [15], i.e. SideW(LC_i), and Fold(*KEY*, 128) = $NLC_{i-1} \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3} \oplus C_{i-4}$ = $F^{-1}(NLC_i) \oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3} \oplus C_{i-4}$ = $F^{-1}(KS_i \oplus$ SideW(LC_i)) $\oplus C_{i-1} \oplus C_{i-2} \oplus C_{i-3} \oplus C_{i-4}$. Since Fold(*KEY*, 128) is equal to *KEY* here, we just determined the whole secret key *KEY*.

Now, we will compare the computed bits of the *KEY* with the originally chosen bits *KEY*{1, ..., 33} and *KEY*{96, ..., 128}. If they do not match, we proceed by guessing another key value. If we obtain more than one candidate for the key, it is possible to reject false keys by using the knowledge of several additional plaintext bits. Finally, we obtain only one correct key.

Complexity of the attack is roughly 2^{66} HBB operations.

With the knowledge of *KEY* we can decipher the whole message *M* including the first four blocks, what completes the proof.

4 An Attack on HBB in the B Mode

We assume only 256-bit key here, where the following attack decreases HBB strength significantly.

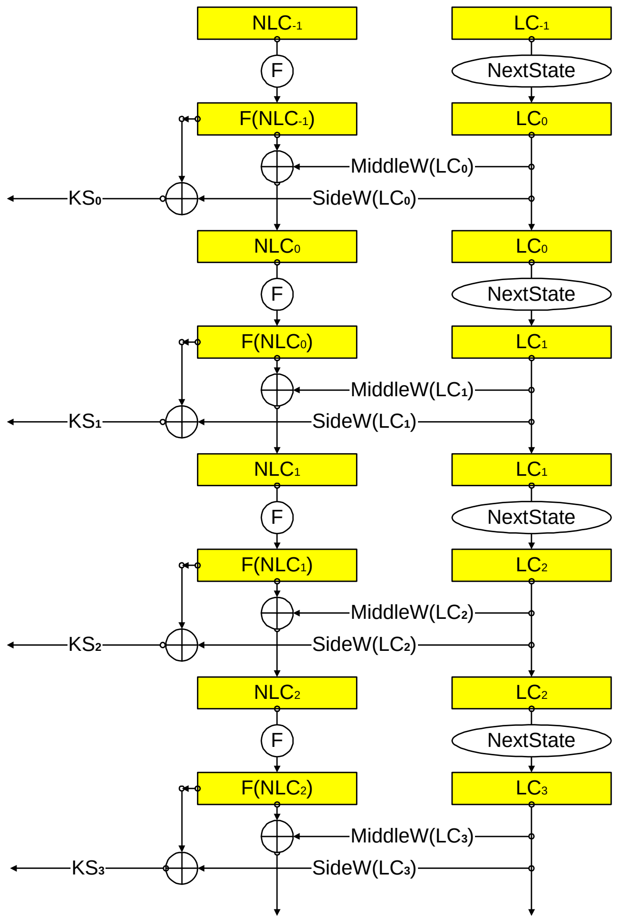

Fig.3: An Attack on HBB in the B Mode

Proposition 3. *Let us assume HBB in the B mode with 256-bit key. Furthermore, suppose knowledge of any 34 consequent blocks of the plaintext. Then, it is possible to decipher the whole plaintext with complexity of order 2^{140}.*

Proof. In the initialization process, the initial values of combiners NLC_{-1} and LC_{-1} are set using a 256-bit *KEY* as described in Section 2.9. Then, the combiners are updated from (NLC_{i-1}, LC_{i-1}) to (NLC_i, LC_i) and the keystream block KS_i is generated for $i = 0, 1, ..., n - 1$ as follows:

1. $LC_i = \mathrm{NextState}(LC_{i-1})$,
2. $NLC_i = \mathrm{F}(NLC_{i-1}) \oplus \mathrm{MiddleW}(LC_i)$,
3. $KS_i = \mathrm{F}(NLC_{i-1}) \oplus \mathrm{SideW}(LC_i)$,
4. $C_i = M_i \oplus KS_i$.

For the sake of simplicity, let us first assume that we know 34 consequent plaintext blocks $M_0, ..., M_{33}$ from the beginning of M, i.e. we know keystream blocks $KS_0, ..., KS_{33}$. We will show how to determine NLC_{-1} and LC_{-1} (which is sufficient for a decryption of the whole M) with complexity 2^{140}.

We will determine 140 bits $LC_0\{1, ..., 32, 33, 34, 35\}$, $LC_0\{222, 223, 224, 225, ..., 256\}$, $LC_0\{257, ..., 288, 289, 290, 291\}$, and $LC_0\{478, 479, 480, 481, ..., 512\}$ by a brute force: Let us guess their value.

Using NextState(), we compute step by step:
$LC_1\{1, ..., 32, 33, 34\}$, $LC_1\{223, 224, 225, ..., 256\}$, $LC_1\{257, ..., 288, 289, 290\}$, $LC_1\{479, 480, 481, ..., 512\}$,
$LC_2\{1, ..., 32, 33\}$, $LC_2\{224, 225, ..., 256\}$, $LC_2\{257, ..., 288, 289\}$, $LC_2\{480, 481, ..., 512\}$, and
$LC_3\{1, ..., 32\}$, $LC_3\{225, ..., 256\}$, $LC_3\{257, ..., 288\}$, $LC_3\{481, ..., 512\}$.

Recall that we determined values of $\mathrm{SideW}(LC_0)$, $\mathrm{SideW}(LC_1)$, $\mathrm{SideW}(LC_2)$, and $\mathrm{SideW}(LC_3)$.

From $NLC_{i-1} = \mathrm{F}^{-1}(KS_i \oplus \mathrm{SideW}(LC_i))$ for $i = 0, 1, 2$, and 3, we compute NLC_{-1}, NLC_0, NLC_1, and NLC_2. Next, we compute

$\mathrm{MiddleW}(LC_0) = NLC_0 \oplus \mathrm{F}(NLC_{-1}) = NLC_0 \oplus KS_0 \oplus \mathrm{SideW}(LC_0) = \mathrm{F}^{-1}(KS_1 \oplus \mathrm{SideW}(LC_1)) \oplus KS_0 \oplus \mathrm{SideW}(LC_0)$,

$\mathrm{MiddleW}(LC_1) = NLC_1 \oplus \mathrm{F}(NLC_0) = NLC_1 \oplus KS_1 \oplus \mathrm{SideW}(LC_1) = \mathrm{F}^{-1}(KS_2 \oplus \mathrm{SideW}(LC_2)) \oplus KS_1 \oplus \mathrm{SideW}(LC_1)$, and

$\mathrm{MiddleW}(LC_2) = NLC_2 \oplus \mathrm{F}(NLC_1) = NLC_2 \oplus KS_2 \oplus \mathrm{SideW}(LC_2) = \mathrm{F}^{-1}(KS_3 \oplus \mathrm{SideW}(LC_3)) \oplus KS_2 \oplus \mathrm{SideW}(LC_2)$.

There are 124 linear relations among bits of $\mathrm{MiddleW}(LC_0)$ and $\mathrm{MiddleW}(LC_1)$ and similarly, there are another 124 linear relations among bits of $\mathrm{MiddleW}(LC_1)$ and $\mathrm{MiddleW}(LC_2)$. These relations are linearly independent, provided the function F^{-1} is carefully designed nonlinear function. Obviously, if F^{-1} does not propagate linear relations among input bits (here $KS_1 \oplus \mathrm{SideW}(LC_1)$, $KS_2 \oplus \mathrm{SideW}(LC_2)$ and $KS_3 \oplus \mathrm{SideW}(LC_3)$) and output bits, the values of $\mathrm{MiddleW}(LC_0)$, $\mathrm{MiddleW}(LC_1)$, and $\mathrm{MiddleW}(LC_2)$ are linearly independent. Therefore, there are 248 relations, which shall hold among computed values of $\mathrm{MiddleW}(LC_0)$, $\mathrm{MiddleW}(LC_1)$, and $\mathrm{MiddleW}(LC_2)$. If they do not hold, the values of $LC_0\{1, ..., 32, 33, 34, 35\}$, $LC_0\{222, 223, 224, 225, ..., 256\}$, $LC_0\{257, ..., 288, 289, 290, 291\}$, and $LC_0\{478, 479, 480, 481, ..., 512\}$ are chosen incorrectly. Then we proceed by guessing next value, until we find the correct one. Complexity of this procedure is roughly 2^{140} HBB operations.

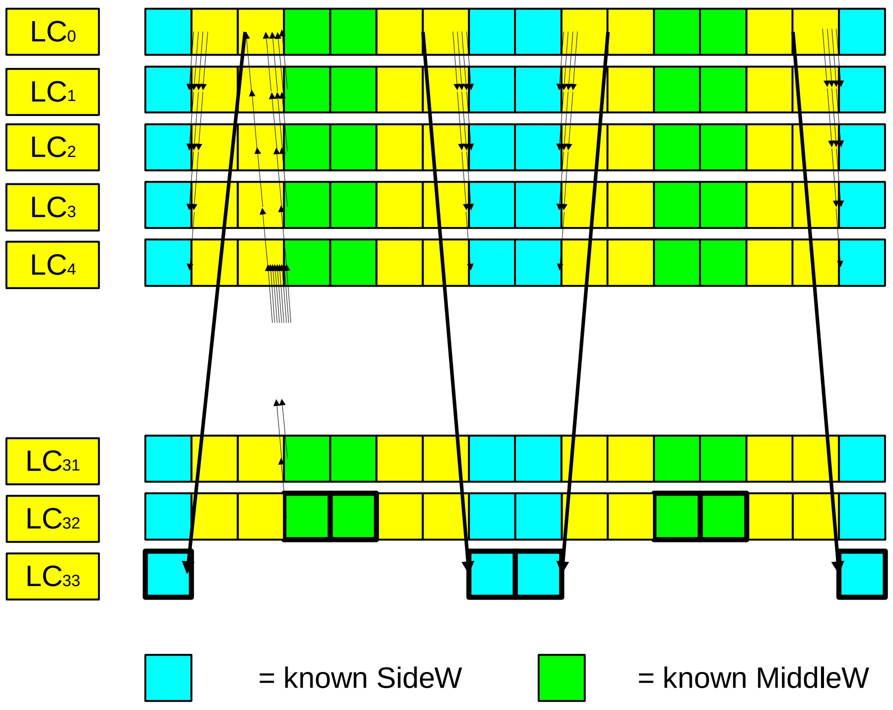

Fig.4: Unveiling LC_0

Now, we will show how to use 30 words KS_4, ..., KS_{33} to increase our knowledge of LC_0 bits. We will work successively in steps for i = 4, ..., 33 according to the following procedure:

On the beginning of i-th step (i = 4, ..., 33), we know

1. $LC_0\{1, ..., 32 + (i - 1)\}$, $LC_0\{225 - (i - 1), ..., 256\}$, $LC_0\{257, ..., 288 + (i - 1)\}$ a $LC_0\{481 - (i - 1), ..., 512\}$, note that it includes SideW(LC_0),
2. $LC_1\{1, ..., 32 + (i - 2)\}$, $LC_1\{225 - (i - 2), ..., 256\}$, $LC_1\{257, ..., 288 + (i - 2)\}$ a $LC_1\{481 - (i - 2), ..., 512\}$, note that it includes SideW(LC_1),
3. ....
4. $LC_{i-1}\{1, ..., 32\}$, $LC_{i-1}\{225, ..., 256\}$, $LC_{i-1}\{257, ..., 288\}$ a $LC_{i-1}\{481, ..., 512\}$, note that it includes SideW(LC_{i-1}),
5. MiddleW(LC_0), MiddleW(LC_1), ..., MiddleW(LC_{i-2}),
6. NLC_{-1}, NLC_0, ..., NLC_{i-2}.

We will try all possible values of 4 bits $LC_0\{32 + i\}$, $LC_0\{225 - i\}$, $LC_0\{288 + i\}$, and $LC_0\{481 - i\}$ successively. From here and from other known bits of LC_0, ..., LC_{i-1} we can compute (using NextState())

$LC_1\{32 + (i - 1)\}$, $LC_1\{225 - (i - 1)\}$, $LC_1\{288 + (i - 1)\}$ and $LC_1\{481 - (i - 1)\}$,
$LC_2\{32 + (i - 2)\}$, $LC_2\{225 - (i - 2)\}$, $LC_2\{288 + (i - 2)\}$ and $LC_2\{481 - (i - 2)\}$,
$LC_3\{32 + (i - 3)\}$, $LC_3\{225 - (i - 3)\}$, $LC_3\{288 + (i - 3)\}$ and $LC_3\{481 - (i - 3)\}$,
...
$LC_{i-1}\{32 + 1\}$, $LC_{i-1}\{225 - 1\}$, $LC_{i-1}\{288 + 1\}$, and $LC_{i-1}\{481 - 1\}$.

From SideW(LC_{i-1}) and new bits of LC_{i-1}, we will compute the whole string SideW(LC_i). Next, we will compute $NLC_{i-1} = \mathrm{F}^{-1}(KS_i \oplus \mathrm{SideW}(LC_i))$ and MiddleW(LC_{i-1}) = $NLC_{i-1} \oplus \mathrm{F}(NLC_{i-2})$.

Between the computed value of MiddleW(LC_{i-1}) and the known value of MiddleW(LC_{i-2}), there are 124 linear relations which shall hold. If they do not hold, we will choose another values of 4 bits $LC_0\{32 + i\}$, $LC_0\{225 - i\}$, $LC_0\{288 + i\}$, and $LC_0\{481 - i\}$, until the correct value is found. After making at most 2^4 trials, we will find the correct value. Complexity of determining these 4 bits is bounded by 2^4 HBB operations.

If we summarize the result at the end of i-th step, we know

1. $LC_0\{1, ..., 32 + i\}$, $LC_0\{225 - i, ..., 256\}$, $LC_0\{257, ..., 288 + i\}$ a $LC_0\{481 - i, ..., 512\}$, including SideW($LC_0$),
2. $LC_1\{1, ..., 32 + (i - 1)\}$, $LC_1\{225 - (i - 1), ..., 256\}$, $LC_1\{257, ..., 288 + (i - 1)\}$ a $LC_1\{481 - (i - 1), ..., 512\}$, including SideW($LC_1$),
3. ....
4. $LC_i\{1, ..., 32\}$, $LC_i\{225, ..., 256\}$, $LC_i\{257, ..., 288\}$ a $LC_i\{481, ..., 512\}$, including SideW(LC_i),
5. MiddleW(LC_0), MiddleW(LC_1), ..., MiddleW(LC_{i-1}),
6. NLC_{-1}, NLC_0, ..., NLC_{i-1}.

Therefore, we fulfilled conditions for a next step. Now, we can continue by a mathematical induction.

Complexity of all 30 steps is bounded by $30*2^4$ HBB operations.

After the 33rd step we know

1. $LC_0\{1, ..., 32, 33, 34, ..., 64, 65\}$, $LC_0\{192, 193, ..., 224, 225, ..., 256\}$, $LC_0\{257, ..., 288, 289, ..., 320, 321\}$, and $LC_0\{448, 449, ..., 480, 481, ..., 512\}$
2. MiddleW(LC_{32}), MiddleW(LC_{31}), ..., MiddleW(LC_0).

From the string LC_0, we are missing only words $LC_0[2]$, $LC_0[5]$, $LC_0[10]$, $LC_0[13]$.

We will show how to determine $LC_0[2]$, provided we know the words $LC_{32}[3]$, $LC_{31}[3]$, ..., $LC_0[3]$. These words are parts of the known strings MiddleW(LC_{32}), MiddleW(LC_{31}), ..., MiddleW(LC_0)).

The whole procedure has 32 steps.

1st step: From $LC_{32}\{97\}$, $LC_{31}\{97, 98\}$, $LC_{30}\{97, 98\}$, ..., $LC_1\{97, 98\}$, and $LC_0\{97, 98\}$, we will successively compute $LC_{31}\{96\}$, $LC_{30}\{96\}$, ..., and $LC_0\{96\}$. For instance, we compute the value of $LC_{31}\{96\}$ using the equation $LC_{31}\{96\} \oplus R_0\{97\}*LC_{31}\{97\} \oplus LC_{31}\{98\} = LC_{32}\{97\}$, where $LC_{31}\{96\}$ is the only unknown.

2nd step: From $LC_{31}\{96\}$, $LC_{30}\{96, 97\}$, $LC_{29}\{96, 97\}$, ..., $LC_0\{96, 97\}$ we will successively compute $LC_{30}\{95\}$, $LC_{29}\{95\}$, ..., $LC_0\{95\}$.

3rd step: From $LC_{30}\{95\}$, $LC_{29}\{95, 96\}$, $LC_{28}\{95, 96\}$, ..., $LC_0\{95, 96\}$ we will successively compute $LC_{29}\{94\}$, $LC_{28}\{94\}$, ..., $LC_0\{94\}$.

...

32nd step: From $LC_1\{66\}$, $LC_0\{66, 67\}$ we will compute $LC_0\{65\}$.

In this way, we obtain the bits $LC_0\{65, 66, ..., 96\}$, including the whole word $LC_0[2]$.

Similarly as for $LC_0[2]$, we can determine words $LC_0[5]$, $LC_0[10]$, $LC_0[13]$ from the knowledge of words 4, 11, and 12 of the strings MiddleW(LC_{32}), MiddleW(LC_{31}), ..., MiddleW(LC_0).

In this way, we compute the remaining unknowns of the string LC_0.

Having obtained LC_0, we can compute LC_{-1}. Recall that we also know NLC_{-1}. Using LC_{-1} and NLC_{-1}, we can decipher the rest of the message.

At the beginning of the attack, we assumed the knowledge of blocks $M_0, ..., M_{33}$. In a more general case when we know blocks $M_i, ..., M_{i+33}$ for some $i > 0$, we can determine NLC_{i-1} and LC_{i-1} by the aforesaid process. Having computed them, we can continue to NLC_{i-2} and LC_{i-2} and further on, until we get to NLC_{-1} and LC_{-1}. Therefore, the particular placement of the known subsequent plaintext blocks is unimportant for the attack.

The total complexity of the attack is $2^{140} + 30*2^4$, i.e. roughly 2^{140} HBB operations.

For a 256-bit key, this reduces the cipher strength significantly, what shows itself as a fault in its design.

5 Notes

The variables T have a very limited impact in the SS mode. These variables are used only to encrypt the first four plaintext blocks. The attacker can ignore these blocks and concentrate solely on the rest of the plaintext.

The description of HBB [1] states that the length of generated keystream is limited to 2^{64} blocks. It should be less. From the birthday paradox, it follows that in such a sequence there are two ciphertext super blocks (C_{i+1}, C_{i+2}, C_{i+3}, C_{i+4}) and (C_{j+1}, C_{j+2}, C_{j+3}, C_{j+4}), having the same 128-bit value $C_{i+1} \oplus C_{i+2} \oplus C_{i+3} \oplus C_{i+4} = C_{j+1} \oplus C_{j+2} \oplus C_{j+3} \oplus C_{j+4}$ with probability around 50%. Therefore, the following 128-bit keystream blocks (in the SS mode) will differ by a known value of SideW[NextState(C_{i+1}, C_{i+2}, C_{i+3}, C_{i+4})] $\oplus$ SideW[NextState(C_{j+1}, C_{j+2}, C_{j+3}, C_{j+4})], and we will obtain 128-bit information about the plaintext, what is not desired property.

6 Conclusion

In this paper, we presented several significant weaknesses of the HBB cipher=. They follow from improper design and combination of its linear and nonlinear part. In the SS mode, we can recover the whole key by a known-plaintext attack with complexity of order 2^{66} for a 128-bit key and with complexity of order 2^{67} for a 256-bit key. In the B mode with 256-bit key, we can exploit knowledge of 34 consequent plaintext blocks to recover the whole plaintext with complexity of order 2^{140}. It follows that the design of HBB is seriously flawed implying that the cipher shall be withdrawn and used only for research purposes.

Acknowledgement

I would like to thank to Tomáš Rosa for helpful suggestions and comments.

7 References

[1] Sarkar, P.: Hiji-bij-bij: A New Stream Cipher with a Self-synchronizing Mode of Operation, in *Proc. Of Progress in Cryptology - INDOCRYPT 2003*, 4th International Conference on Cryptology in India, New Delphi, India, December 2003, Springer-Verlag, Lecture Notes in Computer Science, Vol. 2904, pp. 36 - 51, 2003