Cryptanalysis of SBLH

Goce Jakimovski and Ljupčo Kocarev

Institute for Nonlinear Science
University of California, San Diego
9500 Gilman Drive,
La Jolla, CA 92093-0402
lkocarev@ucsd.edu

Abstract. SBLH is a 256-bit key stream cipher that is used in Business Security's products for voice, fax and data communication. The cipher is claimed to be quite unique and yet very powerful. In this brief report, we suggest a possible chosen ciphertext attack on SBLH. We show that with 2^{24} ciphertext/plaintext pairs, one can successfully recover the active key of length 2^{17} bits.

1 Introduction

SBLH is a patented stream encryption algorithm developed by Business Security [1]. One may summarize the description of the algorithm in [1] using the following quote:

> *The algorithm has been in use in our products nearly ten years, during which it has been extensively analyzed by ourselves, our customers and various governments. All analytical effort pointed to the same conclusion, SBLH is a strong algorithm.*

The structure of the algorithm is very simple. It consists of four building blocks: two memories and two encoders. In early versions of systems built around SBLH, the memories were filled with random data generated by hardware source. In modern versions, a 256-bit key is used by the key expansion algorithm to create the active key. In this paper we show that with 2^{24} ciphertext/plaintext pairs, one can successfully recover the active key in SBLH of length 2^{17} bits. This is the outline of the paper. In the next section we present a general description of the algorithm, while in Section 3 we show how the active key of SBLH can be recovered using a chosen ciphertext attack.

2 General Description of SBLH

SBLH is a stream cipher i.e. the algorithm converts plaintext to ciphertext one bit at a time. It generates a stream of pseudorandom bits, which is XOR-ed with the plaintext to form the ciphertext. The ciphertext is then fed back into the algorithm making the output dependent on both the key and the previous

M. Matsui (Ed.): FSE 2001, LNCS 2355, pp. 144–151, 2002.

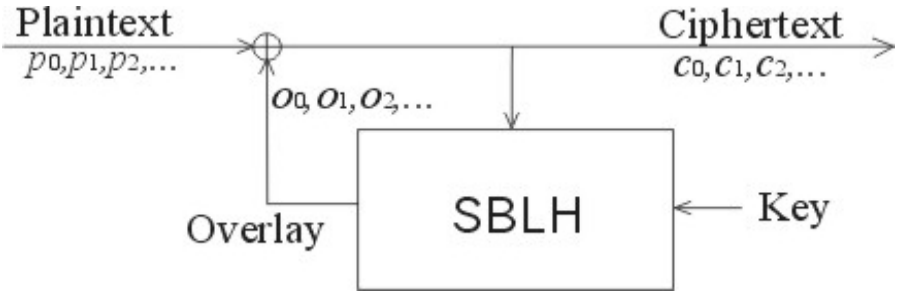

Fig. 1. The encryption process in SBLH

ciphertext. Figure 1 gives an outline of how the encryption process works in SBLH. Overlay is just another word for keystream.

As previously mentioned, each keystream bit is a function of the key and some of the previous ciphertext bits. More specifically each keystream bit is a function of the key and the 64 previous ciphertext bits. Thus, the algorithm is self-synchronizing. If a bit error or a bit slip occurs this will only affect the following 64 bits. After that, the algorithm will be synchronized again. The price we have to pay for self-synchronization is error expansion. Any single error will always affect the following 64 bits. Decryption with SBLH is very similar to encryption. The same keystream used during the encryption is once more added to the ciphertext. This produces the original plaintext. In order to reproduce the keystream used to encrypt, we put the XOR after the data is fed into SBLH. The data that was fed back into SBLH during the encryption is now fed into SBLH during the decryption. The algorithm will therefore produce the same keystream. The decryption process is shown in Figure 2.

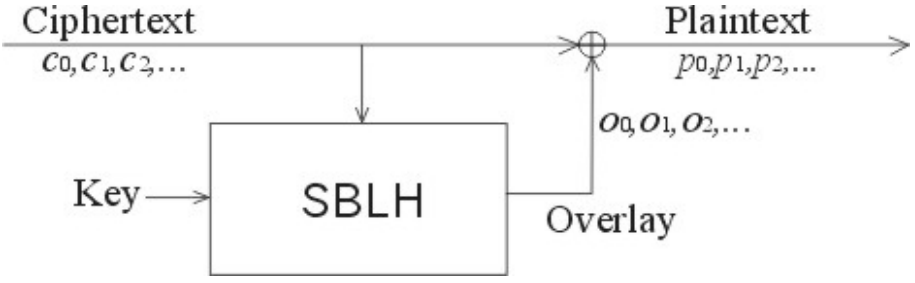

Fig. 2. The decryption process in SBLH

The internal structure of SBLH is quite unique, and the concept is very simple. SBLH is constructed from four simple building blocks, two memories and two encoders. This concept can be used in a wide variety of different algorithms. Key size, error propagation and self-synchronization are some of the parameters

that can be easily adjusted by modifying the basic building blocks of SBLH. Figure 3 describes the structure of SBLH working in encryption mode.

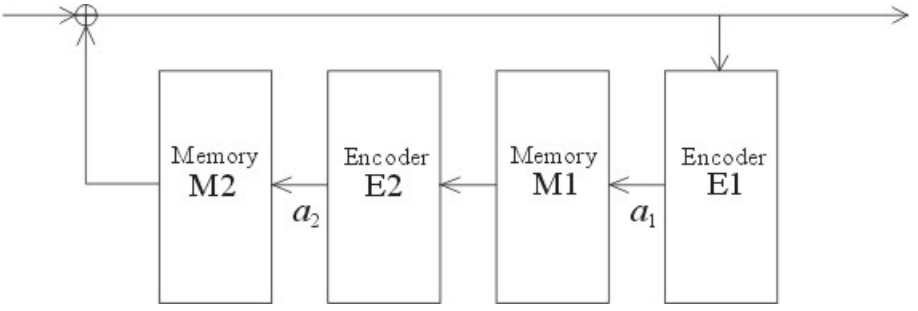

Fig. 3. The structure of SBLH

SBLH takes as input a bit stream (in figure 3 the bit stream is the ciphertext). A bit from the input stream enters the first encoder E_1 where it remains for some time T_1. During this time the bit and $n-1$ other bits currently in E_1 are used by the encoder to produce a sequence of addresses in memory M_1. A equivalent way of modeling E_1 is as a function of n bits, $E_1(x_0...x_{n-1}) = a_1$, where a_1 is an address in memory M_1.

The value stored in M_1 at the address a_1, generated by E_1, is fetched and passed to the second encoder E_2. E_2 is very similar to E_1. The second encoder E_2 is a function of m bits, $E_2(x_0...x_{m-1}) = a_2$, where a_2 is an address in memory M_2. The result of E_2 is used to fetch a value from M_2. The value at address a_2 of memory M_2 is used as overlay to produce the ciphertext. The error propagation and self-synchronization properties of SBLH are completely determined by the depth of the convolutional encoders E_1 and E_2. In this case, the encoder E_1 has 16 bits memory; the output from E_1 is dependent on the last 16 bits. The second encoder E_2 has 48 bits memory. Together E_1 and E_2 have 64 bits of memory, which equals the error propagation/self synchronization of SBLH. The conclusion is that the size of the memory of SBLH matches the sum of the sizes of memories in the encoders E_1 and E_2.

The memories M_1 and M_2 must each be filled with 2^{16} bits of random data. The contents of M_1 and M_2 form the active key. In early versions of systems built around SBLH, the memories were filled with random data generated by hardware source. This made the key length equal to 2^{16} bits. In modern versions the memories are filled with pseudorandom data. The actual key length has been reduced to 256 bits, which is expanded into memories M_1 and M_2. An overview of the key expansion can be found in figure 4.

The expansion starts with 256-bit random key being loaded into a memory. Then the shift-register is loaded with initial value. Before the algorithm can produce any output it has to perform a run-up (the algorithm goes through a sequence of steps without producing any output).

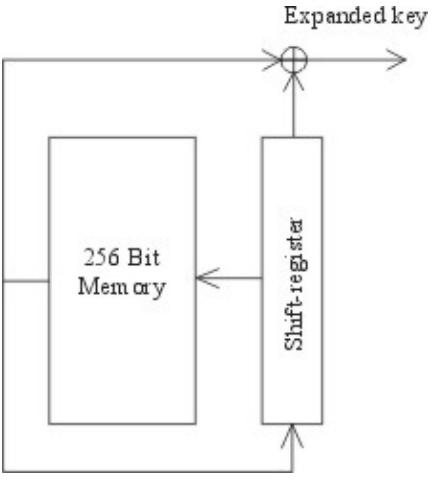

Fig. 4. The key expansion algorithm

3 Chosen Ciphertext Attack

In this section we will describe a chosen ciphertext attack on SBLH. We use the following notation:

c_i – the ciphertext bit in the i-th moment;

$a_{1,i} = E_1(c_{i-15} \dots c_i)$ – output of the encoder E_1 in the i-th moment;

$s_i = M_1(a_{1,i})$ – bit stored at address $a_{1,i}$ in the memory M_1;

$a_{2,i} = E_2(s_{i-47} \dots s_i)$ – output of the encoder E_2 in the i-th moment;

$o_i = M_2(a_{2,i})$ – overlay bit i.e. bit stored at address $a_{2,i}$ in the memory M_2.

Now, let us consider two ciphertext sequences $c'_0, c'_1, \dots c'_n$ and $c''_0, c''_1, \dots c''_n$ $(n \geq 63)$, such that

$$s'_i = s''_i,\ i = n-47, \dots, n-1. \tag{1}$$

If $M_1(a'_{1,n}) = M_1(a''_{1,n})$ i.e. the bit stored at address $a'_{1,n} = E_1(c'_{n-15} \dots c'_n)$ is equal to the bit stored at address $a''_{1,n} = E_1(c''_{n-15} \dots c''_n)$, then $s'_n = s''_n$. Therefore, $a'_{2,n} = E_2(s'_{n-47} \dots s'_n)$ equals $a''_{2,n} = E_2(s''_{n-47} \dots s''_n)$ and $o'_n = M_2(a'_{2,n}) = M_2(a''_{2,n}) = o''_n$. If $M_1(a'_{1,n}) \neq M_1(a''_{1,n})$ i.e. the bit stored at address $a'_{1,n} = E_1(c'_{n-15} \dots c'_n)$ and the bit stored at address $a''_{1,n} = E_1(c''_{n-15} \dots c''_n)$ are different, then the equality $o'_n = o''_n$ will hold with some probability δ. We will assume that δ has some fixed value less than one. If the number of zeroes in the memory M_2 equals the number of ones and $a_{2,i}$ is uniformly distributed, then $\delta = \frac{1}{2}$.

Example 1. The most obvious way to achieve $s'_i = s''_i, i = n-47, \dots, n-1$ is to choose the ciphertext sequences such that $c'_i = c''_i, i = 0, \dots, n-1$. Let $c'_i = c''_i = 1, i = 0, \dots, n-1$, $c'_n = 1$ and $c''_n = 0$. In this case, the condition (1) is satisfied, $a'_{1,n} = E_1(1111111111111111)$, $a''_{1,n} = E_1(1111111111111110)$, $s'_n = M_1(a'_{1,n})$ and $s''_n = M_1(a''_{1,n})$. If $M_1(a'_{1,n}) = M_1(a''_{1,n})$, then $s''_n = s'_n \Rightarrow a''_{2,n} = a'_{2,n} \Rightarrow$

$o''_n = o'_n$. If $M_1(a'_{1,n}) \neq M_1(a''_{1,n})$, then $o''_n = o'_n$ iff $M_2(E_2(s''_{n-47} \dots s''_n)) = M_2(E_2(s'_{n-47} \dots s'_n))$. The last equation is satisfied with some probability δ.

The previously described property of the algorithm can be exploited to check whether the bits at some addresses $A_1 = a'_{1,n}$ and $A_2 = a''_{1,n}$ in the memory M_1 are equal or not. We submit for decryption two ciphertext sequences c'_i and $c''_i, i = 0, \dots 63$, such that $s'_i = s''_i, i = 16, \dots 62$, $E_1(c'_{48} \dots c'_{63}) = A_1$ and $E_1(c''_{48} \dots c''_{63}) = A_2$ (we'll not discuss the existence of the sequences, because for the addresses that will be used in the attack it is easy to construct such sequences). From the ciphertext and corresponding plaintext sequences we compute o'_{63} and o''_{63} . If $o'_{63} \neq o''_{63}$, then $s'_{63} \neq s''_{63}$ i.e. the bit at address A_1 and the bit at the address A_2 are different. If $o'_{63} = o''_{63}$, then we are not sure whether $M_1(A_1) = M_1(A_2)$ or $M_1(A_1) \neq M_1(A_2)$ (as mentioned before, we'll assume that the probability of the event $o'_{63} = o''_{63}$, when $M_1(A_1) \neq M_1(A_2)$ is δ). Hence, if $o'_{63} = o''_{63}$ we repeat the previously described procedure. We repeat the procedure until event $o'_{63} \neq o''_{63}$ happens. If, after large number of repetitions, $o'_{63} = o''_{63}$ is always satisfied, then we assume that $M_1(A_1) = M_1(A_2)$ i.e. the bit at address A_1 in the memory M_1 is equal to the bit at address A_2 in the memory M_1. The probability that we made a mistake is $\delta^t \to 0, t \to \infty$, where t is the number of repetitions of the procedure. We will assume that $M_1(A_1) = M_1(A_2)$ if $o'_{63} = o''_{63}$ during $t = 256$ repetitions.

Example 2. Suppose we want to check whether the bits at addresses A_1 and A_2, where $A_1 = E_1(1111111111111111)$ and $A_2 = E_2(1111111111111110)$, are equal. We submit for decryption the following ciphertext sequences:

$$\underbrace{1111\dots11}_{40bits}1010101011111111111111$$

$$\underbrace{1111\dots11}_{40bits}1010101011111111111110$$

From the resultant plaintexts we compute o'_{63} and o''_{63}. If $o'_{63} \neq o''_{63}$, then we assume that $M_1(A_1) \neq M_1(A_2)$. If $o'_{63} = o''_{63}$, then we submit for decryption two new ciphertext sequences:

$$\underbrace{1111\dots11}_{40bits}1110101011111111111111$$

$$\underbrace{1111\dots11}_{40bits}1110101011111111111110$$

From the resultant plaintexts we compute the new overlay bits o'_{63} and o''_{63}. If $o'_{63} \neq o''_{63}$, then we assume that $M_1(A_1) \neq M_1(A_2)$, while if $o'_{63} = o''_{63}$, we repeat the procedure. If, after 256 repetitions, $o'_{63} = o''_{63}$ is always satisfied, we assume that $M_1(A_1) = M_1(A_2)$.

Let $M_1(A_1) = M_1(A_2)$. We can use this fact to determine the relation between bits at some addresses A'_1 and A'_2. This is done in the next example.

Example 3. Let us assume that during the procedure described in example 2 it was determined that $M_1(A_1) = M_1(A_2)$. We can also determine the relation between $M_1(A'_1)$ and $M_1(A'_2)$, where $A'_1 = E_1(1111111111111110)$ and $A'_2 = E_1(1111111111111100)$. Since $s'_{63} = s''_{63}$ for all ciphertext sequences used in the previous example, we can construct the following procedure for checking whether the bits at addresses A'_1 and A'_2 are equal or not. Submit for decryption two ciphertext sequences:

$$\underbrace{1111\ldots11}_{40bits}10101010111111111111111110$$

$$\underbrace{1111\ldots11}_{40bits}10101010111111111111111100$$

From the resultant plaintexts compute o'_{64} and o''_{64}. If $o'_{64} \neq o''_{64}$, then $M_1(A'_1) \neq M_1(A'_2)$. If $o'_{64} = o''_{64}$, then submit for decryption two new ciphertext sequences:

$$\underbrace{1111\ldots11}_{40bits}11101010111111111111111110$$

$$\underbrace{1111\ldots11}_{40bits}11101010111111111111111100$$

From the resultant plaintexts we compute the new overlay bits o'_{64} and o''_{64}. If $o'_{64} \neq o''_{64}$, then we assume that $M_1(A'_1) \neq M_1(A'_2)$, while if $o'_{64} = o''_{64}$, we repeat the procedure. If, after 256 repetitions, $o'_{64} = o''_{64}$ is always satisfied, we assume that $M_1(A'_1) = M_1(A'_2)$.

In the examples 2 and 3, it was shown how relations between bits of the memory M_1 could be determined. Each new relation that can't be derived from the previous relations reveals one bit of the contents of M_1. More relations we construct, more information about the contents of M_1 we gain. If the number of relations is large enough, we can determine all 2^{16} bits of M_1. In the following, we describe systematic procedure for constructing relations between the bits of M_1.

0. Initialize an array r of $2^{16} \times 2^{16}$ elements. The elements of r contain information about the relations between the bits of M_1. If the relation between the bit at address $E_1(a)$ and the bit at address $E_1(b)$ is not determined, then $r[a][b] = r[b][a] = 2$. If the relation between the bit at address $E_1(a)$ and the bit at address $E_1(b)$ is determined and $M_1(E_1(a)) = M_1(E_1(b))$, then $r[a][b] = r[b][a] = 1$. If the relation between the bit at address $E_1(a)$ and the bit at address $E_1(b)$ is determined and $M_1(E_1(a)) \neq M_1(E_1(b))$, then $r[a][b] = r[b][a] = 0$. In this step, it holds $r[a][b] = 2(a \neq b)$ and $r[a][b] = 1(a = b)$.

1. According to example 2, for all a and b such that $a \oplus b = 0000000000000001$ find out how the bits at addresses $E_1(a)$ and $E_1(b)$ are related. If $M_1(E_1(a)) = M_1(E_1(b))$, then set $r[a][b] = r[b][a] = 1$. If $M_1(E_1(a)) \neq M_1(E_1(b))$, then set $r[a][b] = r[b][a] = 0$.

2. According to example 3, for all a and b such that $r[a][b] = 1$ and $a \oplus b = 0000000000000001$ we can determine the relation between $M_1(E_1(a'))$ and

$M_1(E_1(b'))$, where $a' = (a << 1) \bmod 2^{16}$ and $b' = (b << 1) \bmod 2^{16}$. We note that $a' \oplus b' = 0000000000000010$. If the relation between $M_1(E_1(a'))$ and $M_1(E_1(b'))$ is already determined ($r[a'][b'] \neq 2$), do nothing. If $r[a'][b'] = 2$, then find out how the bits at addresses $E_1(a')$ and $E_1(b')$ are related and update the array r. By updating the array r we mean entering the new relation and all the relations that can be derived using the new and the previous relations.

3. The fourth step is analogous to the previous step. In this step, for all a and b such that $r[a][b] = 1$ and $a \oplus b = 000000000000001\star$ ($\star$ denotes arbitrary value) we can determine the relation between $M_1(E_1(a'))$ and $M_1(E_1(b'))$, where $a' = (a << 1) \bmod 2^{16}$ and $b' = (b << 1) \bmod 2^{16}$. We note that $a' \oplus b' = 00000000000001\star 0$. If the relation between $M_1(E_1(a'))$ and $M_1(E_1(b'))$ is already determined ($r[a'][b'] \neq 2$), do nothing. If $r[a'][b'] = 2$, then find out how the bits at addresses $E_1(a')$ and $E_1(b')$ are related and update the array r.

$\vdots$

16. In a similar way, new relations between bits at addresses $E_1(a')$ and $E_1(b')$ are derived using the relations determined in the previous step. For a' and b' holds that $a' \oplus b' = 1\star\star\star\star\star\star\star\star\star\star\star\star\star\star 0$.

17. In this step, the entries of the encoder E_1 are grouped into classes using the array r. The grouping is performed so that the number of classes is minimal. Let N denotes the number of classes. Each class $K_i, i = 1, \ldots N$ consists of two subsets K_i^1 and K_i^0. The following must hold: (i) $a, b \in K_i^1$ (or $a, b \in K_i^0$) iff $r[a][b] = r[b][a] = 1$, (ii) $a \in K_i^1$ and $b \in K_i^0$ iff $r[a][b] = r[b][a] = 0$, (iii) $a \in K_i$ and $b \in K_j$, $i \neq j$ iff $r[a][b] = r[b][a] = 2$.

18. 2^N possible contents of M_1 are constructed using the classes K_i. For each of these contents, the content of M_2 is determined and tested using the existing ciphertext/plaintext pairs. If the content of M_2 can be uniquely determined i.e. there is no address A such that in one moment $M_2(A) = 0$ and in the other $M_2(A) = 1$, we consider the pair (M_1, M_2) as a possible value of the active key.

The efficiency of the attack depends on the number of classes N. Suppose that the entries of the encoder E_1 are grouped into classes after each step. Let N_i denotes the number of classes after i-th step, let $l_i = \frac{2^{16}}{N_i}$ denotes the average class length and let L_i denotes maximum possible average class length after the i-th step. We note that $L_1 = 2, L_2 = 4, \ldots, L_{16} = 2^{16}$. Figure 5 depicts the values of $D(i) = L_i - \bar{l_i}$ and $d(i) = \frac{D(i)}{L_i}$, where $\bar{l_i}$ is an arithmetic mean of l_i calculated for 2^{10} randomly chosen contents of M_1.

It is obvious that the difference D tends to zero. This is due to the fact that the average number of elements of the classes increases with every step, and thus, the number of relations that can be derived increases too. On the other hand, the required number of new relations decreases with every step. Therefore, the number of contents of M_1 for which $N > 1$ is negligibly small.

When $N = 1$, there are only two possible contents of M_1. Therefore, the active key is one of the two possible pairs (M_1, M_2). Which one, it can be easily checked. The number of relations needed to group the entries of the encoder

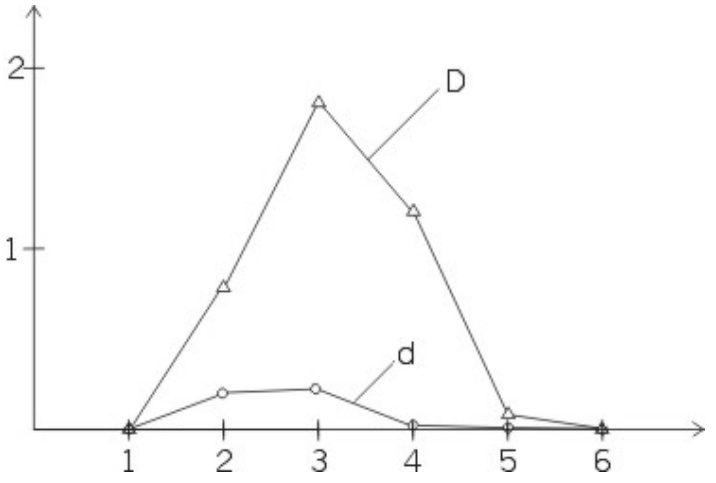

Fig. 5. $D(i)$ and $l(i)$ versus i. See the text for details.

E_1 in one class is $2^{16} - 1$. The number of required ciphertext/plaintext pairs is $(2^{16} - 1) \times 256 \approx 2^{24}$.

References

1. Business Security AB, Lund, Sweden, http://www.bsecurity.se