SOBER Cryptanalysis

Daniel Bleichenbacher and Sarvar Patel
{bleichen,sarvar}@lucent.com

Bell Laboratories
Lucent Technologies

Abstract. SOBER is a new stream cipher that has recently been developed by Greg Rose for possible applications in wireless telephony [3]. In this paper we analyze SOBER and point out different weaknesses. In the case where an attacker can analyze key streams generated for consecutive frames with the same key we present an attack, that in our implementation requires less than one minute on a 200Mhz Pentium.

1 Overview and Motivation

Encryption schemes that are used in wireless telephony have to meet rather difficult constraints. The schemes have to encrypt rather large amounts of data, but mobile stations often have very limited computational power. Hence the encryption scheme must be quite efficient. Simple schemes, however, are under a big risk of being insecure.

In this paper we analyze the stream cipher SOBER, which was developed by Greg Rose [3]. An improved version of SOBER [4] has recently been submitted for a TIA (Telecommunications industry association) standard. The goal of the TIA standards process is to replace ORYX and CMEA, for which weaknesses have been discovered [6,5].

The outline of the paper is as follows. Section 2 describes the notation used in this paper. We review SOBER in Section 3. Section 4 analyzes SOBER under the assumption that an attacker knows the first 4 bytes of 64 key streams all generated with the same key and the frame numbers 0 through 63. This attack is the most severe attack described in our paper, since it requires only 256 bytes of known key stream and is very efficient. Our implementation often finds the key in less than a minute on 200 Mhz Pentium. That is we reduce the time complexity from 2^{128} for a brute force search to about 2^{28}. Section 5 analyzes SOBER under the more restrictive assumption that an attacker knows only one key stream rather than several as assumed before. Section 6 describes a weakness in the key setup procedure. In particular, we show that using an 80 bit key instead of a 128 bit key can considerably simplify cryptanalysis. In Section 7 we show how to find keys that produce no output. While this is not a severe flaw in SOBER, such keys might be used for a denial of service attack in certain protocols.

L. Knudsen (Ed.): FSE'99, LNCS 1636, pp. 305–316, 1999.

2 Notation, Underlying Mathematical Structure and Basic Operations

Most of our cryptanalysis in this paper is based on heuristic arguments. For example, we often assume that certain values are uniformly distributed. We will call statements based on heuristic arguments *claims* as opposed to statements that are rigorously provable, which are usually called *theorems*. However, we have verified our claims if possible with computer experiments.

SOBER uses operations over the ring $\mathbb{Z}/(256)$ and the field GF_{2^8} and mixes these operations. Hence, the computations depend heavily on the exact representation of elements in these structures.

The field GF_{2^8} is represented as $\mathrm{GF}_2[x]/(x^8+x^6+x^3+x^2+1)$. Addition and multiplication over GF_{2^8} will be denoted by $\oplus$ and $\otimes$ respectively. The symbol '+' will denote addition over the ring $\mathbb{Z}/(256)$ and ' $\wedge$ ' will denote the bitwise logical AND. Elements in $\mathbb{Z}/(256)$ and GF_{2^8} can both be represented with one byte. An implicit conversion ϕ takes place when an element of GF_{2^8} is used for an operation over $\mathbb{Z}/(256)$. This conversion ϕ can be defined by

$$\phi(\sum_{i=0}^{7} c_i x^i) = \sum_{i=0}^{7} c_i 2^i \text{ where } c_i \in \{0,1\},$$

i.e. the coefficients of polynomials in $\mathrm{GF}_2[x]/(x^8+x^6+x^3+x^2+1)$ are interpreted as bits of the binary representation of integers in the interval $[0, 255]$. For the rest of the paper will use ϕ and ϕ^{-1} as implicit conversions between GF_{2^8} and $\mathbb{Z}/(256)$ when necessary, e.g. we can use the constants 141 and 175 to represent the polynomials $x^7+x^3+x^2+1$ and $x^7+x^5+x^3+x^2+1$ respectively.

3 Description of SOBER

SOBER is a stream cipher having a key length up to 128 bits. A stream cipher generally transforms a short key into a key stream, which is then used to encrypt the plaintext. In particular, the ciphertext is usually the bitwise XOR of the key stream and the plaintext. A key stream must not be used to encrypt two plaintexts, since the knowledge of two ciphertexts encrypted with the same key stream would allow to extract the XOR of the two plaintexts. SOBER has a feature, called frames, that allows us to use the same key multiple times without generating the same key stream. This is very useful in applications where a lot of small messages have to be encrypted independently. In particular the key stream generator has two input parameters, namely the key and the frame number. The frame number is not part of the key. It may be just a counter, and is sent to the receiver in clear or is known to him. Hence we have to assume that the frame number is known to an attacker too.

The main reason to chose a linear feedback shift register over GF_{2^8} instead of a LFSR over GF_2 was efficiency [3]. In this paper we consider the security of the cipher. Therefore we will not describe implementation details.

3.1 LFSR

SOBER is based on a linear feedback shift register of degree 17 over GF_{2^8} producing a sequence s_n that satisfies the recurrence relation

$$s_{n+17} = (141 \otimes s_{n+15}) \oplus s_{n+4} \oplus (175 \otimes s_n). \tag{1}$$

During the key and frame setup this recurrence relation is slightly modified to add the key and frame number into the state of the LFSR. In particular the following relation is used.

$$s_{n+17} = (141 \otimes s_{n+15}) \oplus s_{n+4} \oplus (175 \otimes (s_n + a_n)) \tag{2}$$

The value a_n depends on either the key or the frame number and will be described in the next two sections.

3.2 Key Setup

A key consists of 4 to 16 bytes, $K_0, \ldots, K_{\ell-1}$, where ℓ denotes the length of the key in bytes. The LFSR is initialized to the first 17 Fibonacci numbers, i.e.

$$\begin{aligned} s_0 &= s_1 = 1 \\ s_n &= s_{n-1} + s_{n-2} \pmod{256} \text{ for } 2 \le n \le 16 \end{aligned}$$

Then each byte of the key is added to the lowest byte of the LFSR, before the LFSR is cycled, i.e. $a_n = K_n$, so that

$$s_{n+17} = (141 \otimes s_{n+15}) \oplus s_{n+4} \oplus (175 \otimes (s_n + K_n)) \text{ for } 0 \le n \le \ell - 1. \tag{3}$$

Next, we set $a_\ell = \ell$ and $a_n = 0$ for $\ell < n \le 40$ and compute the state $(s_{40}, \ldots, s_{56})$ of the register. This concludes the setup of the key.

Setting $a_\ell = \ell$ guarantees that the state of the LFSR after the initialization is unique for all keys. The key setup procedure is an almost linear operation. In particular, there exists a 17×18 matrix M independent of the key, such that

$$\begin{pmatrix} s_{40} \\ s_{41} \\ \vdots \\ s_{56} \end{pmatrix} = M \begin{pmatrix} 1 \\ s_0 \oplus (s_0 + a_0) \\ s_1 \oplus (s_1 + a_1) \\ \vdots \\ s_{16} \oplus (s_{16} + a_{16}) \end{pmatrix}. \tag{4}$$

3.3 Frame Setup

In some application the same key is used to produce more than one key stream. In order to achieve this a frame number f is used to generate a different key stream for each time the key is used. A frame number is a 32-bit integer. It is added in 11 steps to the LFSR. In particular the n-th step adds the bits 3n,...,3n+7 to the lowest register of the LFSR and cycles the LFSR afterwards. Hence

$$s_{n+17} = (141 \otimes s_{n+15}) \oplus s_{n+4} \oplus (175 \otimes (s_n + a_n)) \quad \text{for } 40 \leq n \leq 50$$
$$\text{where } a_n = \left\lfloor f/8^{n-40} \right\rfloor \bmod 256.$$

The values $s_i : 58 \leq i \leq 96$ are computed with (1). The frame initialization is finished as soon as the state $(s_{80}, \ldots, s_{96})$ is computed.

3.4 Computation of the Key Stream

The sequence v_n is computed in a non-linear way from s_n as follows:

$$v_n = (s_n + s_{n+2} + s_{n+5} + s_{n+12}) \oplus (s_{n+12} \wedge s_{n+13})$$

Assume that the stream cipher is in a state n, i.e. represented by $(s_n, ..., s_{n+16})$. Then the cipher produces the key stream as follows: n is incremented by 1 and v_n is stored in a special register called the *stutter control register*. Hence, v_{81} is the first stutter control byte if a frame number is used. The stutter control register is used to generate 0 to 4 bytes of the key stream as follows: First the stutter control register is divided into 4 bit pairs. For each of these bit pairs, starting with the least significant bits the following is done based on the value of the 2 bits:

bit pair	Action taken
00	Increment n by 1. (No output is produced.)
01	Output the key stream byte $105 \oplus v_{n+1}$ and increment n by 2.
10	Output the key stream byte v_{n+2} and increment n by 2.
11	Output the key stream byte $150 \oplus v_{n+1}$ and increment n by 1.

4 Analysis of the Frames

In this section, we analyze the situation where an attacker knows different key streams generated with one key, but different frame numbers. We might regard the frame number as a part of the key. Hence, the attack described in this section could be classified as an related-key attack. Biham introduced this type of attack and applied it to various ciphers in [1]. We use a the fact that different frames are strongly related to derive an efficient differential attack [2] based on this relation. It is the most serious attack presented in this paper, since both the number of necessary known plaintext bytes and the number of necessary operations to recover the key are small. In particular, we show the following result:

Claim 1. *Given the first 4 bytes of key stream for 64 frames generated with the same key and the frame numbers 0 through 63. Then we expect to find the key in about* $c2^{28}$ *steps, where* c *is the number of steps in the innermost loop.*

Due to the nature of the attack the time complexity can vary and is hard to examine, because the attack is based on a search tree, whose size depends on given input parameter. Our attack is based on the assumption that the values of the first stutter byte are uniformly distributed, and hence that all 4 values for the first 2-bits occur with about the same frequency. We will for example assume that at least 24 out of 64 bit pairs have the value 01 or 11. If that condition is not met, we can run the algorithm again with a lower threshold, and hence a larger tree to search. To verify that our assumptions are reasonable we have implemented the attack. Recovering the key often requires less than minute on a 200Mhz Pentium. We also found that the algorithm indeed shows the expected behavior.

Before we describe the attack, we will define some notations and point out some properties of the frame initialization that will be helpful for the attack.

In order to distinguish the state of LFSR produced with different frame numbers f we will denote the nth value in the sequence s for f by

$$s_n(f).$$

The differential of two such sequences will be denoted by $\Delta_n(f_1, f_2)$, i.e.

$$\Delta_n(f_1, f_2) := s_n(f_1) \oplus s_n(f_2).$$

The j-th byte of the key stream for frame f will be denoted by

$$p_j(f).$$

The 6 least significant bits of the frame number are used twice during the frame setup. They are added to s_{40} and s_{41} during the computation of s_{57} and s_{58}. Therefore, if f_1 and f_2 differ only in the 6 least significant bits, and the attacker can guess s_{40} and s_{41} then he can easily compute $\Delta_n(f_1, f_2)$ for all $n \geq 0$ by

$$\begin{aligned}
\Delta_n(f_1, f_2) &= 0 \text{ for } 0 \leq n \leq 56 \\
\Delta_{n+17}(f_1, f_2) &= (175 \otimes (s_n + a_n(f_1)) \oplus (s_n + a_n(f_2))) \text{ for } n = 40, 41 \\
\Delta_{n+17}(f_1, f_2) &= (175 \otimes \Delta_n(f_1, f_2)) \oplus \Delta_{n+4}(f_1, f_2) \oplus \\
&\qquad (141 \otimes \Delta_{n+15}(f_1, f_2)) \text{ for } n \geq 42
\end{aligned}$$

We can now give a short description of our attack. First we guess s_{40} and s_{41} and compute the Δ's. Then we guess certain well chosen bits of the internal state of the LFSR for the frame 0. Based on these guesses and the precomputed Δ's we compute the corresponding bits for the LFSR state of the other frames. These bits are then used to compute possible output bytes. We don't know the state of the stutter byte, hence we cannot predict with certainty what an output byte should look like. Therefore, we will use a probabilistic approach, i.e. we

reject our guess if considerably less output bytes coincide with our prediction than we would expect for a correct guess. Otherwise we extend our guess with a few more bits and test again. This can be repeated recursively until we have either found a reason to reject that guess or we know enough information about the LFSR state to compute the key. We will now give a more detailed description of the attack.

Step 1: Guessing s_{40} and s_{41}. First we guess the values s_{40} and s_{41}. Next, we compute the values for $\Delta_n(0, f)$ for $82 \leq n \leq 99$ and $0 \leq f \leq 63$.

Different pairs of values of s_{40} and s_{41} may lead to equal values for all $\Delta_n(0, f)$. Since we will only need the $\Delta's$ but not s_{40} and s_{41} we can use this fact to improve the algorithm. In particular the most significant bit of s_{40} has no influence on $\Delta_n(0, f)$. Moreover, the $s_{40} = 0$ and $s_{40} = 64$ give the same Δ's. Hence, it is sufficient to chose s_{40} satisfying $1 \leq s_{40} \leq 127$. Two values for s_{41} are equivalent if either their 3 least significant bits are all zero or if their $k > 3$ least significant bits are equal and the k-th least significant bit is zero. It follows, that it is sufficient to chose s_{41} among the following set:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15,$$
$$25, 26, 27, 28, 29, 30, 31, 57, 58, 59, 60, 61, 62, 63,$$
$$121, 122, 123, 124, 125, 126, 127, 249, 250, 251, 252, 253, 254, 255\}$$

Step 2: Guessing $s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0)$ and $s_{95}(0)$. Now we would like to guess the values for $s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0)$ and $s_{95}(0)$. In order to improve the efficiency, we will start with guessing only the two least significant bits of each byte. We test that guess and if the guess looks promissing then we extend it recursively by guessing the next least significant bits and test again.

Testing is done as follows. Assume that we have guessed the k least significant bits of $s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0)$ and $s_{95}(0)$. This allows us to compute the k least significant bits of

$$s_n(f) = s_n(0) \oplus \Delta_n(0, f)$$

for $n \in \{82, 84, 87, 94, 95\}$. We can now compute the k least significant bits of

$$v_{82}(f) = (s_{82}(f) + s_{84}(f) + s_{87}(f) + s_{94}(f)) \oplus (s_{94}(f) \wedge s_{95}(f))$$

With probability 1/4 the two least significant bits of the first stutter byte v_{81} is 01 and with the same probability it is 11. Hence, with probability 1/4 each the first output byte $p_1(f)$ for frame f is either $v_{82}(f) \oplus 105$ or $v_{82}(f) \oplus 150$. Now, we count the number of frames f for which the k least significant bits of $p_1(f)$ and either $v_{82}(f) \oplus 105$ or $v_{82}(f) \oplus 150$ are equal. We would expect to find about 32 matches on average, if we have guessed the bits for $s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0)$ and $s_{95}(0)$ correctly. In our experiments, we rejected the values for s whenever we found less than 24 matches. The correct solution should be rejected by mistake with a probability less than 3%.

After having guessed and checked all 8 bits, usually only few, typically less than 100 possibilities for the tuple $(s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0), s_{95}(0))$ remain.

We also know the values for the Δ's and have partial knowledge about $s_{40}(0)$ and $s_{41}(0)$. Consequently, the most time consuming part of the analysis is already done.

Steps 3.1 – 3.4: Guessing more of the LFSR state. Next, we extend each of these tuples by using a similar method involving $v_{83}(f), v_{84}(f), v_{85}(f)$ and $v_{86}(f)$. In step 3.i ($1 \leq i \leq 4$) we guess the unknown values out of $s_{82+i}(0), s_{84+i}(0), s_{87+i}(0), s_{94+i}(0)$ and $s_{95+i}(0)$. Then we compute $v_{82+i}(f)$ and compare these values to the known key stream. This comparison of $v_{82+i}(f)$ with the key stream slightly more complex than before, since we have to decide, which byte of the key stream we should use for comparison. Generally, we assume that if there was a match with $v_{82+j}(f)$ for $j < i$ that the corresponding byte was indeed generated using $v_{82+j}(f)$. We will use the following rules in our algorithm:

- If a test $p_j(f) = v_i(f) \oplus x$ where $x \in \{0, 105, 150\}$ was successful then we use $p_{j+1}(f)$ for the next comparison with $v_{i+1}(f)$ otherwise we use $p_j(f)$ and $v_{i+1}(f)$.
- If $p_j(f) = v_i(f) \oplus 105$ then we will not test $v_{i+1}(f)$, since SOBER skips one byte after generating output of the form $v_i(f) \oplus 105$.
- A test $p_j(f) = v_i(f)$ will only be performed if the last test was not successful.

Step 4: Recovering the key At this point we know all the bytes $s_i(0)$ for $82 \leq i \leq 98$ with exception of $s_{92}(0)$ and $s_{93}(0)$. Usually, the previous steps narrow the possibilities to a few hundred cases. Hence we can easily just test each case with all possible values for $s_{92}(0)$ and $s_{93}(0)$. For each combination we compute $s_i(0)$ for $i = 81, 80, \ldots, 16$ using

$$s_i(0) = (175^{-1} \otimes (s_{i+17}) \oplus (141 \otimes s_{i+15}) \oplus s_{i+4}).$$

Finally, we recover the $i - th$ key byte by

$$K_i = 175^{-1} \otimes (s_{i+4} \oplus (141 \otimes s_{i+15})) \oplus s_{i+17}) - s_i.$$

Remember, that s_i for $0 \leq i \leq 16$ is equivalent to the i-th Fibonacci number. We may now do some further test such as decrypting the whole message to check whether we have found the correct key.

A Very Rough Analysis

It is hard to analyze this algorithm analytically. However, we roughly estimate it's complexity as follows. We have to guess s_{40} (7 bits) and s_{41} (5.4 bits) and the 2 least significant bits of $s_{82}(0), s_{84}(0), s_{87}(0), s_{94}(0)$ and $s_{95}(0)$ (10 bits) before we can make the first test in step 2. Each test has to compare with 64 frames. Hence, so far we have a time complexity of approximately $2^{7+5.4+10+6}$, which is slightly more than 2^{28}. Our experiments show that the first test cuts enough nodes in the search tree that the remaining tree is almost negligible. Hence this rough analysis gives an approximation for the actual runtime of the algorithm.

5 Analysis of the Cipher Stream

In this section we analyze the nonlinear output of SOBER and investigate in how difficult it is to compute the internal state of the LFSR given the key stream of the cipher. In particular, we will now show the following result:

Claim 2. *Assume, that 17 consecutive key stream bytes $p_1, \ldots, p_{17}$ are known. Assume further that these bytes are generated from two stutter control bytes and that the value of the first byte is 10111110_2 and the second one is 01010101_2. Then it is possible to find the internal state of the LFSR in $c2^{72}$ steps, where c is a small constant denoting the number of steps for the innermost loop.*

Let the first stutter byte be v_n. To simplify notation we'll use $t_i = s_{n+i}$. From the assumption on the stutter control register the following is known:

$$(t_0 + t_2 + t_5 + t_{12}) \oplus (t_{12} \wedge t_{13}) \equiv 10111110_2 \pmod{256} \quad (5)$$

$$(t_2 + t_4 + t_7 + t_{14}) \oplus (t_{14} \wedge t_{15}) \equiv p_1 \pmod{256} \quad (6)$$

$$(t_3 + t_5 + t_8 + t_{15}) \oplus (t_{15} \wedge t_{16}) \equiv (p_2 \oplus 10010110_2) \pmod{256} \quad (7)$$

$$(t_4 + t_6 + t_9 + t_{16}) \oplus (t_{16} \wedge t_{17}) \equiv (p_3 \oplus 10010110_2) \pmod{256} \quad (8)$$

$$(t_6 + t_8 + t_{11} + t_{18}) \oplus (t_{18} \wedge t_{19}) \equiv p_4 \pmod{256} \quad (9)$$

$$(t_7 + t_9 + t_{12} + t_{19}) \oplus (t_{19} \wedge t_{20}) \equiv 01010101_2 \pmod{256} \quad (10)$$

$$(t_8 + t_{10} + t_{13} + t_{20}) \oplus (t_{20} \wedge t_{21}) \equiv p_5 \oplus 01101001_2 \pmod{256} \quad (11)$$

$$(t_{10} + t_{12} + t_{15} + t_{22}) \oplus (t_{22} \wedge t_{23}) \equiv p_6 \oplus 01101001_2 \pmod{256} \quad (12)$$

The algorithm works as follows. First we guess the 9 bytes

$$t_0, t_4, t_5, t_6, t_{12}, t_{13}, t_{15}, t_{22} \text{ and } t_{23}.$$

We can use these equations to solve for more values t_i as follows:

use equation	to compute
(1)	$t_{17} = (141 \otimes t_{15}) \oplus t_4 \oplus (175 \otimes t_0)$
(6)	$t_2 = (p_1 \oplus (t_{14} \wedge t_{15})) - (t_4 + t_7 + t_{14}) \pmod{256}$
(12)	$t_{10} = (p_6 \oplus 01101001_2 \oplus (t_{22} \wedge t_{23})) - (t_{12} + t_{15} + t_{22}) \pmod{256}$
(1)	$t_{19} = (141 \otimes t_{17}) \oplus t_6 \oplus (175 \otimes t_2)$
(1)	$t_{21} = (141^{-1} \otimes (t_{23}) \oplus t_{10} \oplus (175 \otimes t_6))$
(1)	$t_8 = t_{21} \oplus (141 \otimes t_{19}) \oplus (175 \otimes t_4)$
(11)	t_{20} (see below)
(1)	$t_9 = t_{22} \oplus (141 \otimes t_{20}) \oplus (175 \otimes t_5)$
(10)	$t_7 = (01010101_2 \oplus (t_{19} \wedge t_{20})) - (t_9 + t_{12} + t_{19}) \pmod{256}$
(6)	t_{14} (see below)
(8)	t_{16} (see below)
(7)	$t_3 = (p_2 \oplus 10010110_2 \oplus (t_{15} \wedge t_{16})) - (t_5 + t_8 + t_{15}) \pmod{256}$
(1)	$t_{18} = 141^{-1} \otimes (t_{20} \oplus t_7 \oplus (175 \otimes t_3))$
(1)	$t_1 = 175^{-1} \otimes ((141 \otimes t_{16}) \oplus t_5 \oplus t_{18}$
(9)	$t_{11} = (p_4 \oplus (t_{18} \wedge t_{19})) - (t_6 + t_8 + t_{18}) \pmod{256}$
(1)	$t_{24}, t_{25}, \ldots$

We have now enough information to compute the corresponding key stream and compare it to $p_7, \ldots, p_{16}$. If all the values are equal then we output $t_0, \ldots, t_{16}$ as possible candidate for the internal state of the LFSR. Solving most of these equations is easy, since the equations are either linear over GF_{2^8} or $\mathbb{Z}/(256)$. The only non-linear equations are the equations (6),(8), and (11). These equations have the form

$$(A + X) \oplus (X \wedge B) = C,$$

where A, B, C are given and X is unknown. There are 2^{24} possible combinations for A, B and C, hence it is feasible to precompute the set of solutions of all equations and store the result. Then solving these non-linear equations requires only table look-ups. Moreover, there are totally 2^{24} solutions for the 2^{24} equations. Hence, we expect one solution on average for each of the equations (6),(8), and (11). Therefore we may assume that testing each guess for the 9 bytes $t_0, t_4, t_5, t_6, t_{12}, t_{13}, t_{15}, t_{22}$ and t_{23} takes a constant number c of steps and thus the whole algorithms needs about $c2^{72}$ steps.

If we know enough key stream then we can repeat the attack above for all tuples of 17 consecutive key stream bytes. Since a stutter byte is used for 3 key stream bytes on average, it follows that $n + 17$ consecutive key stream bytes contain on average $n/3$ sequences of 17 bytes of key stream where the first byte was generated with a new stutter bytes. The probability that such a sequence was generated starting with the two stutter bytes 10111110_2 and 01010101_2 is 2^{-16}.

Claim 3. *There exists an algorithm that given $n+16$ consecutive key stream bytes of SOBER, can find the internal state of the LFSR in $cn2^{72}$ steps, with probability about $1 - (1 - 2^{-16})^{n/3}$. where c denotes the number of steps for performing the innermost loop of our algorithm.*

Hence, we expect to find the internal state of SOBER after examining about $n = 3 \cdot 2^{16}$ bytes using about $c2^{89.6}$ steps on average. Respectively, after $c2^{89.1}$ steps we have found the key with probability $1/2$. Even though, this is still a huge number, the security of SOBER is nonetheless much lower than what we would expect from a cryptosystem with 128 bit keys.

Remarks. Variants of the attack described in this section are possible and hence an attacker would have some flexibility. First, it can be noticed that the bytes $p_7, \ldots, p_{17}$ are only used in the last step of the attack to distinguish correct guesses of the LFSR state from wrong guesses. This differentiation would also be possible if other bytes than $p_7, \ldots, p_{17}$ are known. Additionally, statistical information on the values $p_7, p_8, \ldots$ would be already sufficient to complete the attack.

The algorithm we have described is based on the assumption that we can find key stream bytes $p_1, \ldots$ produced with two stutter bytes being 10111110_2 and 01010101_2. These values have the property that the resulting system of equations is easily solvable. However, they are chosen somewhat arbitrarily from a set of equally well suited values. Other values would lead to a different system of

equations, with a different method to solve. For simplicity we haven't described any such alternative attacks here. It should, however, be noted that these alternatives could possibly be used to reduce the number of necessary key stream bytes.

6 Analysis of the Key Setup

In this section, we analyze the key setup. During the key setup a key of length 40 bit to 128 bit is expanded into an initial state of the LFSR register. Not all 2^{136} states of the LFSR are therefore possible. Hence a desirable property of the key setup would be that the knowledge of the key setup procedure is not helpful for cryptanalysis. However, we show that SOBER leaves the LFSR in a state that can be easily described and that this information is helpful for cryptanalysis.

In the following we will restrict ourselves to the analysis of the key setup for 80 bit keys, since this key size is often recommended for application that do not require high security. Similar, attacks can be found for other key sizes too.

Claim 4. *Given 12 consecutive bytes of known plaintext and corresponding ciphertext encrypted with an unknown 80 bit key. Then it is possible to find the key in about* $c2^{68}$ *operations with probability* $9/16$.

The idea behind the attack is as follows: The attacker guesses 4 bits of the first stutter byte. This will give him two equations such as the ones described used in Section 5, if the two bit pairs are different from 00. This happens with probability 9/16. Then he can guess 8 bytes of the internal state of the LFSR, such that he can compute 2 more bytes. The remaining state of the LFSR can now be found by using Equation (4). Finally, the solution has to be verified by computing more key stream bytes and comparing it to some known plaintext.

7 Weak Keys

In this section, we describe how to find keys that generate no key stream. We will call them *weak keys*.

After the key and frame setup it can happen that the LFSR contains only 0's. In that case, all non-linear output bytes and therefore all stutter bytes are equal to zero. Hence the stream cipher will loop forever without generating a single byte of key stream.

The probability that this event occurs is very low, i.e., for a randomly chosen key and frame number this will occur only with probability 2^{-136}.

The existence of such keys is no risk of privacy. However, the party that choses the keys might use this weakness to initiate a denial of service attack against another party. Generally, we recommend to check for such a state of the LFSR. One possible countermeasure is to reinitialize the stream cipher with a different frame number (e.g. $2^{32} - 1$. This would be sufficient countermeasure, since at most one frame for every key can show this behavior.)

Claim 5. *There exists an efficient algorithm that given a frame number f, finds a key K, if such a key exists, such that the LFSR is in the zero state after initializing the key K and the frame number f. The probability that such a key exists for a randomly chosen frame number is larger than* $1/256$.

Assume that we are given a frame number f and that we are looking for a key k such that the LFSR after the initialization is in the zero state. Since each of the steps during the initialization of the frame number is linear, these steps can easily be reversed. Next, we try to find the corresponding key. For simplicity, we are only interested in 128-bit keys. Given the state after the initialization of the key (i.e. n=40), the state of the LFSR after step n=17 can be computed by reversing the effect of cycling the register and hence $s_{17}, \ldots, s_{33}$ can be computed. Since $s_0, .., s_{16}$ are also known, we can now easily compute the i-th key K_i byte from

$$((175 \otimes (s_i + K_i)) \oplus s_{i+4} \oplus (141 \otimes s_{i+15})) = s_{i+17}.$$

Finally we have to check, whether adding the keylength 16 to s_{16} and cycling the register would produce the correct value for s_{33}. We may assume that this will be the case with probability $1/256$. The runtime of this algorithm is negligible. A few pairs of keys and corresponding frame numbers are given below.

key	frame
85FA3E93F1993225E71B13EFC0811DAC	114
34149FED30DEAC25D4FB89A0F0551DA7	12F
FA1F9189BEE0A2128BA818165B83F86E	240
376DA8DCF4632B0FD4A3EB745E3DB584	291
8A7F49B63524B10FE78371BB4E09B57F	2AE
956F2E347D8CC9F50F14C978C68E740B	530
237DE6F06CE5BAEBD58040767F00D31A	5BE
C7DC7B5B6FAFFCF1CFADB819493C4D77	63F
5EE4BB2970166108F78CFE33374CAE94	7E5

8 Conclusion

We have shown different flaws in SOBER. We have implemented the most serious attack and shown that we can often recover the key in less than 1 minute on a Pentium/200Mhz. Greg Rose has developed a newer, still unpublished version of SOBER[4]. This version is based on commissioned cryptanalysis by Codes & Ciphers Ltd. This work is unpublished, but partially mentioned in [4]. It seems that similar attacks to those described in Section 5 and Section 6 have been known found independently to our analysis. This new version of SOBER takes countermeasures against these attack. Moreover, these countermeasures seem to avoid the attack in Section 4 in this paper, though further analysis is still necessary.

References

1. E. Biham. New types of cryptanalytic attacks using related keys. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 398–409, Berlin, 1994. Springer Verlag.
2. E. Biham and A. Shamir. *A Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993.
3. G. Rose. A stream cipher based on linear feedback over $GF(2^8)$. In C. Boyd and E. Dawson, editors, *ACISP'98, Australian Conference on Information Security and Privacy*, volume 1438, page Lecture Notes in Computer Science. Springer Verlag, July 1998.
4. G. Rose. SOBER: A stream cipher based on linear feedback over GF(2^8). (preprint), 1999.
5. D. Wagner, B. Schneier, and J. Kelsey. Cryptanalysis of the cellular message encryption algorithm. In B. S. Kaliski, editor, *CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 526–537. Springer Verlag, 1997.
6. D. Wagner, L. Simpson, E. Dawson, J. Kelsey, W. Millan, and B. Schneier. Cryptanalysis of ORYX. In *Fifth Annual Workshop on Selected Areas in Cryptography, SAC'99*, 1998.