SOBER: A Stream Cipher based on Linear Feedback over $GF(2^8)$[1]

Greg Rose

QUALCOMM Australia, Suite 410, Birkenhead Point, Drummoyne NSW 2137, Australia
ggr@qualcomm.com

Embedded applications such as voice encryption in wireless telephones can place severe constraints on the amount of processing power, program space and memory available for software encryption algorithms. Additionally, some protocols require some form of two-level keying which must be reasonably fast. This paper introduces a mechanism for creating a family of stream ciphers based on Linear Feedback Shift Registers over the Galois Finite Field of order 2^n, where n is chosen to be convenient for software implementation. A particular stream cipher based on this methodology, SOBER, is presented and analysed.

1. Introduction

Encryption of digitised voice data in wireless telephony has been hampered by the lack of computational power in the mobile station. This has led to either weak encryption such as the *Voice Privacy Mask* used in the Time Division Multiple Access standard [9], or hardware generated stream ciphers such as the *A5* [1] cipher used in the European GSM digital phone standard. Hardware based stream ciphers have the disadvantage of incurring additional cost in the manufacture of the mobile station, as well as proving to be both costly and time-consuming in the event that the encryption needs to be changed. Since most of the mobile telephones in use incorporate a microprocessor and memory, a software stream cipher which is fast and uses little memory would be ideal for this application.

Stream ciphers combine the data to be encrypted and a stream of pseudo-random bits generated by the encryption algorithm, usually with the exclusive-or (XOR) operation. Decryption is the process of reproducing the pseudo-random bit stream and removing it from the encrypted data. For the encryption to be secure, it must be computationally difficult to predict the output of the generator.

---

[1] A version of this paper was presented at the 1998 Australian Conference on Information Security and Privacy, published by Springer-Verlag in the Lecture Notes in Computer Science series. This paper contains a much more detailed security analysis, and some of the details of the SOBER stream cipher have been strengthened in the light of that analysis.

Many of the techniques used for generating the stream of pseudo-random bits are based on *Linear Feedback Shift Registers* (LFSRs) over the Galois Finite Field of order 2. The register is updated by shifting it by one bit, and calculating the new bit to be shifted in as a linear function of the other bits. The operations involved, namely shifting and bit extraction, are efficient in hardware but inefficient in software, especially if the length of the shift register exceeds the length of the registers of the processor in question. In addition, only one bit of output is generated for each set of operations, which again is inefficient use of the general purpose CPU. There are a large number of different stream ciphers which utilise LFSRs as their underlying mechanism; for good surveys, see [6] or [8].

The method presented here utilises a number of techniques to greatly increase the generation speed of the pseudo-random stream in software on a microprocessor. However, the fundamental properties of stream ciphers based on LFSRs are preserved so that the extensive study carried out on stream ciphers is still relevant and it incorporates empirical techniques thought to produce good encryption characteristics. The particular algorithm presented, *SOBER*, additionally implements a two level keying scheme which is useful in mobile telephony applications. It has been designed specifically for application within Code Division Multiple Access mobile phones.

2. Fast Linear Feedback Shift Register Byte Generation

Linear feedback shift registers are normally based on a recurrence relation over the Galois Field of order 2 (*GF(2)*). The output sequence (of bits) is defined by

$$s_{n+k} = c_{k-1}s_{n+k-1} + c_{k-2}s_{n+k-2} + \ldots + c_1s_{n+1} + c_0s_n$$

where s_n is the *n*th output of the sequence, the constant coefficients c_i are elements of GF(2), that is, single bits, and *k* is called the *order* of the recurrence relation.

Such registers prove inefficient in software because the general purpose processor, which can manipulate byte- or word-sized objects, is instead forced to perform many operations on single bits.

2.1 LFSR over GF(2^8)

LFSRs can operate over any finite field, and can be made more efficient in software by utilising a finite field more suited to the processor. A particularly good choice for such a field is the Galois Field with 256 elements (*GF(2^8)*). The elements of this field and the coefficients of the recurrence relation occupy exactly one byte of storage and can be efficiently manipulated in software. In the meantime, the order *k* of the recurrence relation which encodes the same amount of state is reduced by a factor of 8.

The field GF(2^8) (and more generally for orders which are a power of two) can be represented (the *standard representation*) as the coefficients modulo 2 of all polynomials with degree less than 8. That is, an element a of the field is represented by a byte with bits $(a_7, a_6, ..., a_0)$ which represents the polynomial

$$a_7x^7 + a_6x^6 + ... + a_1x + a_0$$

The addition operation for such polynomials is simply addition modulo two (XOR) for each of the corresponding coefficients; the additive identity is the polynomial with all coefficients 0. Multiplication in the field is polynomial multiplication with modulo 2 coefficients, with the resulting polynomial being reduced modulo an irreducible polynomial of degree 8. The multiplicative identity element is $(a_7, a_6, ..., a_0) = (0, 0, ..., 0, 1)$. The choice of an irreducible degree 8 polynomial alters the way elements of the group are mapped into encoded bytes on the computer, but does not otherwise affect the actual group operations. For the purpose of this paper, and SOBER, we use the irreducible polynomial

$$x^8 + x^6 + x^3 + x^2 + 1$$

for the modular reduction, but any irreducible monic polynomial of degree 8 would work.

Now that there is a known representation for the 256 elements of the underlying field which can be stored in a single computer byte, the LFSR can be specified in terms of bytes instead of bits, and successive output values will also be bytes rather than bits. Calculating these successive output values requires a number of constant multiplications followed by addition of these terms.

Polynomial multiplication and subsequent modular reduction are complicated operations on a general purpose processor. However, since there are only a small number of field elements the operations can be replaced by table lookups and other simpler operations. For general multiplication within the field, exponentiation and logarithm (index) tables can be used. In the case of $GF^*(2^8)$ the tables are each 256-bytes long and would normally be precomputed and stored in read-only memory. Note that when one of the operands is zero, there is no corresponding entry in the logarithm table, so the operands must first be tested for a zero result.

For the generation of outputs from an LFSR, however, the situation is much simpler. The only multiplications required involve the constant coefficients c_i (above). For efficient implementation, these would be chosen to be 0 or 1 most of the time. The c_i which were other values would have precalculated tables $t_i[]$ stored in read-only memory, where the jth entry of the table would be

$$t_i[j] = j * c_i \; (j = 0 .. 2^8-1)$$

In the example implementation shown below, only two of the coefficients c_i are neither 0 nor 1, and so for the same 2*256 bytes of tables, the multiplications can be done with a single table lookup for each, and no conditional tests.

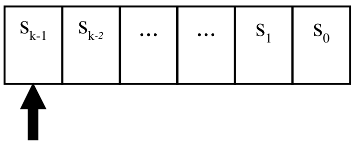
In the discussions below, the operations of addition and multiplication over the Galois Field of order 256 are shown as the symbols ⊕ and ⊗ (respectively) to avoid confusion with the more normal uses of addition and multiplication.

2.2 Circular Buffer or Sliding Window for the Shift Register.

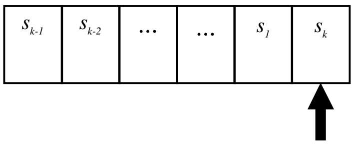
Another important improvement can be made to the software implementation of an LFSR when the elements of the register are bytes or words and not bits. When implemented in hardware, shifting bits is a very simple and efficient operation. In software, and for “registers” larger than the processor registers, it becomes an iterative procedure and very expensive. When the units to be shifted are bytes or words, shifting becomes a little simpler, although it is still iterative.

It is not necessary for a software implementation of a byte- or word-oriented shift register to actually shift at all. Another data structure called a *circular buffer* is often used to implement a shift register. In this implementation, instead of shifting all of the values in memory, a single index or pointer is moved along one memory location. When this index reaches the end of the buffer it is reset to the beginning, and so the buffer acts as if it were a circle and not a straight line.

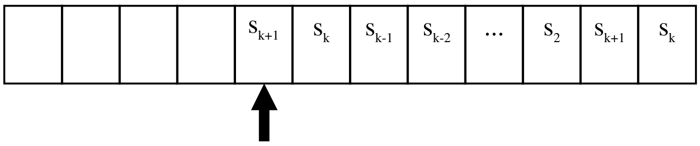
The following diagram shows how a buffer might represent the current state of an LFSR.



When the next value in the sequence is generated, the circular buffer pointer is moved as shown below:



Generating new values of the sequence (as well as other aspects of this cipher that will be seen later), require addressing elements of the buffer, other than the last one, as defined by their offset in the sequence. There are a number of straightforward ways to do this with a circular buffer, such as maintaining multiple indexes or pointers and updating all of them, or calculating offsets modulo k (which can be fairly efficient when k is also a power of 2, but is otherwise usually an expensive operation on microprocessors). SOBER uses yet another technique called a *sliding window* in which a buffer up to twice as long as required is used. When a new value is written into the buffer, it is written into *two* places in the buffer, and all of the intermediate values can be accessed at fixed offsets from the current index or pointer. The pointer starts in the middle of the double length buffer, and when it reaches the end it is reset to the middle again. The following diagram shows what a sliding window implementation would look like.



No matter where in the left half of the buffer the pointer appears, the previous *k-1* elements can be addressed to the right of it. When the pointer reaches the left end of the buffer, both halves have identical contents, and it can be reset to the middle. The right-most half of the register is actually exactly the same as the circular buffer above. Note that with this sliding window approach, the shift register can be arbitrarily long and still be addressed efficiently, while the update time remains constant and short. These attributes are important for this application.

3. Using Linear Feedback Shift Registers for Cryptography

Correct use of LFSRs for stream ciphers can be difficult. Any linearity remaining in the output stream which is derived from the shift register can be exploited to deduce the state of the register at some point. The register can then be driven forwards or backwards, as desired, to recover the output stream. Despite this, a number of successful ciphers do use LFSRs, employing certain useful techniques to overcome the linearity problem. A few of the methods used in these ciphers are of particular interest as they are applicable to LFSRs over larger order fields, including $GF(2^8)$.

1. *Stuttering* (irregular decimation) involves clocking the register a variable and unpredictable number of times between outputs.

2. Using a *non-linear function* of the state of the shift register, instead of its natural "output". This is particularly effective as there are some good choices for non-linear functions which work on byte-sized data on general purpose processors.

3. Using *multiple shift registers* and combining outputs from them (again in a non-linear fashion), or using a *variable feedback polynomial* on one register. Multiple shift registers are useful in hardware, where the cost relates to the total capacity, and operating the shift registers can happen in parallel. In software on a commodity processor, a single larger shift register which can be updated in constant time is preferable.

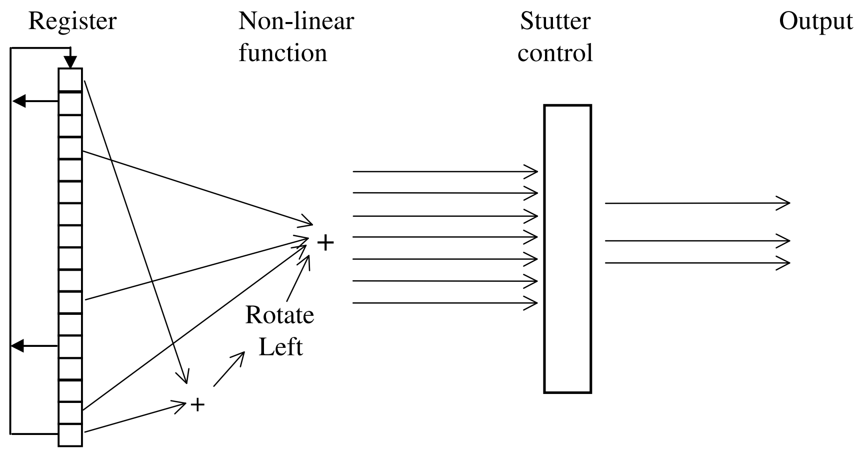
Some other simple design rules for the cryptographic use of LFSRs also apply. The recurrence relation should be chosen carefully so that the output stream is of maximal length. Paar [7] gives an efficient algorithm for finding primitive polynomials over $GF((p^n)^m)$.

4. SOBER: A Stream Cipher based on LFSRs over GF(2^8)

The stream cipher, SOBER, uses the byte operations described above over GF(2^8). Table lookup is used for the constant multiplication required. The sliding window approach is used to allow fast updating of the shift register.

After keying, the shift register is clocked normally, but its output is not used directly. Instead, to disguise the linearity of the register, a non-linear function (with respect to GF(2^8)) of some of the state bytes is evaluated, and the resulting non-linear outputs are decimated in an irregular fashion known as *stuttering*. This results in an average of three octets of output for every seven shifts of the register. Diagrammatically this is shown in Figure 1 below.

Figure 1. Block structure of SOBER



4.1 The Linear Feedback Shift Register

The SOBER linear feedback shift register is a Fibonacci-style register, 17 octets long. This length allows the generator to be in 2^{136}-1 (approximately 8.7×10^{40}) states. The time to update the register using the recurrence relation with given number of non-zero coefficients is constant in a software implementation, irrespective of the length of the register. Therefore, the only cost of having a long register is a few extra bytes of memory. This length was chosen because it conveniently allows for keys of up to 128 bits, although this does not translate to 128 bits of "strength".

Throughout this paper, the elements of the LFSR at a given instant in time, will be referred to as $r_0,\ldots,r_{16}$ respectively. The sequence of outputs generated by cycling the register will be referred to as $s_0,\ldots,s_n$.

The LFSR over $GF(2^8)$ is mathematically equivalent to eight parallel shift registers over GF(2) of length 136, each with the same recurrence relation but different initial states. Ideally, for maximum diffusion and strength against cryptanalysis, the degree 136 polynomial over GF(2) should have approximately half of its coefficients as 1. There are many polynomials over $GF(2^8)$ which have three coefficients which approach this goal, but all three of these coefficients are greater than 1. While in terms of theoretical security this is perfectly acceptable, using such polynomials in practice would require three lookup tables and references, which is less efficient than necessary.

The position of the feedback taps (non-zero coefficients) were chosen in conjunction with the non-linear functions. (This will be explained in more detail in the following section). To avoid one unnecessary table lookup, the binary polynomial chosen must therefore possess a coefficient of 1. Analysis indicates that 65 non-zero binary coefficients is the best possible option, given the feedback structure, which is very close to the goal of 68 non-zero coefficients. In all, there are sixteen primitive polynomials over $GF(2^8)$ meeting this criteria. However, the suitable polynomials actually form two equivalence classes on $GF(2^8)$ where each of the eight polynomials in a class represent one of the possible bit shifts within the byte. The polynomials from both classes (denoted type 1 and type 2) are as follows:

S_n	S_{n+4}	S_{n+15}	Type
99	1	206	1
106	1	201	1
142	1	126	1
148	1	214	1
203	1	146	1
210	1	119	1
213	1	195	1
222	1	136	1
40	1	109	2
45	1	38	2
46	1	159	2
57	1	129	2
110	1	209	2
117	1	63	2
132	1	219	2
140	1	97	2

For our purposes, there appears to be no distinction between equivalence classes, or the polynomials therein, therefore the first polynomial has been selected as the recurrence relation. Thus, SOBER's LFSR is updated according the relation:

$$s_{n+17} = 206 \otimes s_{n+15} \oplus s_{n+4} \oplus 99 \otimes s_n$$ [2]

where addition ($\oplus$) is the exclusive-or (XOR) operation, and multiplication ($\otimes$) is polynomial modular multiplication using byte table lookups on precomputed tables.

4.2 Nonlinear Function of the Shift Register State

Much of the cryptographic security of SOBER resides in a non-linear function used to defend against attacks on the linear feedback or stuttering phase. It is therefore important to make this function as strong as possible.

Both the recurrence relation and the non-linear function access elements of the shift register. The elements used in the non-linear function are carefully chosen such that the distances between them form a "full positive difference set" [3] with maximum span. The feedback positions also form a (non-full) positive difference set, chosen so that only one difference is shared with the non-linear function.

The non-linear function combines five bytes, denoted s_n, s_{n+1}, s_{n+6}, s_{n+13}, and s_{n+16}, using a technique involoving the rotation of partial sums. Although a degree of non-linearity can be achieved using straight arithmetic addition, the least significant bit retains linear properties, and so this method is not sufficient. Numerous possible functions were tested and the results analysed. The output of each non-linear function was compared bitwise to its nearest linear approximation, and the mean absolute deviation and root-mean-square deviation from 0.5 - the theoretically perfect result – were calculated. Of the large number of efficient functions studied, the one for which these deviations were least, was chosen.

$$v_n = ROTL(s_n + s_{n+16}) + s_{n+1} + s_{n+6} + s_{n+13}$$

where the addition is modulo 256, and *ROTL*() is a rotation left by a single bit. (The selection and placement of the actual register elements within this function will be left for discussion in the following section).

It is worth noting that one of the functions tried used a highly non-linear table (the S-box out of SKIPJACK), and the performance of the above function was not much worse than that. Introducing another set of tables and lookups was thought not to be necessary.

---

[2] The equivalent binary polynomial is: $1 + x^{2} + x^{8} + x^{10} + x^{12} + x^{14} + x^{15} + x^{16} + x^{17} + x^{21} + x^{25} + x^{27} + x^{28} + x^{29} + x^{30} + x^{34} + x^{36} + x^{40} + x^{41} + x^{42} + x^{43} + x^{44} + x^{46} + x^{49} + x^{51} + x^{53} + x^{54} + x^{55} + x^{56} + x^{57} + x^{59} + x^{60} + x^{61} + x^{64} + x^{66} + x^{67} + x^{68} + x^{69} + x^{70} + x^{72} + x^{73} + x^{74} + x^{75} + x^{76} + x^{79} + x^{80} + x^{81} + x^{82} + x^{83} + x^{86} + x^{89} + x^{91} + x^{93} + x^{94} + x^{95} + x^{96} + x^{108} + x^{109} + x^{113} + x^{117} + x^{119} + x^{121} + x^{128} + x^{132} + x^{136}$

4.3 Stuttering the Non-Linear Output

It is easily conceivable that this nonlinear value derived from the state of the LFSR could be used to efficiently reconstruct the state. The task is made much more difficult if some of the states are not represented in the output, in a way which is difficult to predict. In this implementation, occasional bytes of nonlinear output are used to determine the inclusion of other bytes in the output stream. When the generator is started, the first output byte is taken to be used as a *stutter control* byte. Each stutter control byte is broken into 4 pairs of bits, with the least significant pair of bits being used first. When the four values have all been used, the next non-linear output from the register is used to form the next stutter control byte, and so on.

There are four possible values for each of the two-bit stutter controls. These determine the output from the stream generator as follows:

(0,0) The register is cycled but no output is produced.

(0,1) The register is cycled, and the nonlinear output XOR the constant '01101001'$_2$ becomes the output of the generator. The register is then cycled again.

(1,0) The register is cycled twice, and the (second) nonlinear output becomes the output of the generator.

(1,1) The register is cycled, and the nonlinear output XOR the constant '10010110'$_2$ becomes the output of the generator.

The constants which are used in the above steps are chosen so that any particular bit of the output has a $^1/_3{}^{rd}$ probability of being inverted.

During study, it became apparent that the stuttering function was very effective at defeating correlation-based attacks, but was only marginally useful in defeating brute force attacks such as those mentioned in [10]. A brute force attack on SOBER, utilising known keystream output, would proceed as follows:

1. First, assumptions are made about the value of one or two of the stutter control bytes, yielding one or two nonlinear equations.

2. Based on these assumptions, the known keystream yields some number of nonlinear equations. The preferred assumptions about the stutter control bytes typically give four equations per assumption.

3. Equations based on the linear feedback function are also added to the known equations, to relate any solutions back to the initial 17-octet state.

4. When the nonlinear function involves 5 unknowns, knowing any 4 of them yields the 5th. Therefore, the attacker can proceed by guessing state variables in such a way as to maximize derivations from known or assumed equations.

The elements of the non-linear function were chosen to maximize the complexity of just such an attack.

Analysis of the impact of the non-linear function on a brute force attack, revealed that the span of the non-linear function, that is, the spread between the first and last elements of the register involved in the function, should be maximized. There are in fact 100 distinct full positive difference sets on 17 elements which include both the first and last elements of the register. A program capable of automating the above attack was then written, and the possible attacks guessing 1 or 2 stutter control bytes were enumerated for all of the 100 possible tap position sets. Note that for this attack the actual nonlinear function used is largely irrelevant. The most powerful attack on a particular choice varied from 2^{80} to 2^{96} in complexity. From the 18 configurations where the minimum attack complexity was 2^{96}, the configuration with the highest average attack complexity ($2^{109.7}$) was chosen. The full positive difference set thus chosen was {0, 1, 6, 13, 16}.

The following table illustrates the approximate complexity of the attack given assumptions of one or two stutter control bytes respectively.

	Plaintext required	Computational complexity
1 stutter control assumed	33 bytes	2^{104} (min) / 2^{112} (avg)
2 stutter controls assumed	65 bytes	2^{96} (min) / 2^{110} (avg)

The overlapping difference between the feedback function taps and the nonlinear function taps is 15 (octets s_0 and s_{15} in the feedback function, and s_1 and s_{16} in the nonlinear function). Therefore it was decided that s_1 and s_{16} should appear on opposite sides of the rotation operation in the non-linear function.

The existence of the attack mentioned should not be viewed as a weakness of SOBER, rather it is an overstatement of the size of keys which can be accepted. On one hand, reducing the number of inputs to the nonlinear function means that more equations are needed before the initial state can be computed, but less values for each equation need to be known before the last input can be deduced. Increasing the number of inputs increases the number of guesses needed for each equation to yield a value, but means that there are more coincidences in the equation set which can be profitably exploited. Using a full positive difference set optimises this tradeoff, as explained by Golic. About 5-7 nonlinear equations provide enough information to solve the initial state of the register; interestingly adding more equations (by guessing more stutter control bytes) does not speed up the attack. These non-linear equations, and a corresponding number of linear feedback equations, give an average number of coincidences in which the same initial state variable appears in multiple equations, increasing the value of guessing that state variable. As a rule of thumb, the equations have about 4.5 unknowns which can be revealed by guessing about 3.5 of them, giving an average complexity of approximately 13 guesses (8 bits each) to recover the 17 initial state bytes. (This is a simplification of the actual argument, but gives insight into the complexity. Further study on this subject is being undertaken.)

Note that the execution time of SOBER on a general purpose CPU is independent of the length of the register. Should more strength than the 96 bits provided for by the current design be required, it is now straightforward to extend the register, and choose suitable feedback and nonlinear functions. SOBER as it currently stands is, however, more than adequate for the intended application in mobile telephony.

4.4 Keying the Stream Cipher

This cipher is designed for applications in wireless telephony. In such applications, packets may be lost due to noise, synchronisation between the Mobile Station (cellphone) and the Base Station may be lost due to signal reflection, or a particular call might be handed off to a different base station as the phone zooms along a freeway. Any loss of synchronisation with a stream cipher is disastrous. One solution, used in the GSM system, is to have each encrypted frame implicitly numbered, and the stream cipher re-keyed for each frame with the secret key and the frame number. SOBER supports such a two tier keying structure.

SOBER accepts one secret key, which should be between 4 to 16 bytes in length. As the shift register state holds 17 bytes of information, there is ample provision for the upper limit of the key material. Initial keying proceeds as follows:

1. The 17 bytes of state information are initialised to the first 17 Fibonacci numbers modulo 256. The register uses these as s_0 to s_{16} respectively. There is no particular significance to these numbers being used, except for the ease of generating them.

2. n is initialised to 0 where n is a counter.

3. Each byte of key material in turn, starting with the first, is added modulo 256 to r_{15}. n is then incremented and the register is cycled. The output of the non-linear function is XORed into r_8. This process is continued until all key material is consumed.

4. The length of the key is added to r_{15}.

5. The output of the non-linear function is XORed into r_8, and the register is cycled, for 11 more steps.

The 17 byte state of the register, s_{40} .. s_{56}, can be saved at this point for later use, and the key discarded. However, for shorter keys, the key could be saved and this procedure repeated as necessary, trading additional computation time for some extra memory.

In addition to the secret key, the secondary key, which is not considered to be secret, is used in wireless telephony applications to generate a unique cipher stream for each frame of data. SOBER accepts a public per-frame key, called *frame,* in the form of a 4-octet unsigned integer and uses a per-frame initialisation similar to the one above. If the use of the cipher makes it unnecessary to utilise per-frame key information, for example, for file transfer over a reliable connection-oriented link, this initialisation step can be omitted entirely.

1. The 17 bytes of state information are initialised to the state saved in the key initialisation step. The register uses these as s_0 to s_{16} respectively.

2. n is initialised to 0.

3. The least significant byte of *frame* is added modulo 256 to r_{15}. n is then incremented, *frame* is shifted right by eight bits, and the register is cycled. The output of the non-linear function is XORed into r_8. This process is repeated four times, until all the bits of *frame* have been used.

4. The output of the non-linear function is XORed into r_8, and the register is cycled, for 11 more steps.

This spread of the elements in the register involved in the feedback and nonlinear functions means that the effect of all key octets has diffused to all octets of the register state after those 11 operations. Note that the use of the non-linear function during key loading not only increases diffusion but destroys the affine relationship that results from using only the linear feedback function for mixing.

The *stutter control* byte must be initialised to be the first non-linear output from the shift register. This is done in the reference implementation below by setting the stutter count variable to zero; the first call to generate a byte of output will perform the actual initialisation.

5. Complexity, Memory and Computation Time.

When the register is cycled but the nonlinear output is not used, the nonlinear function need not be calculated. The bytes used for stutter control themselves never appear in the output stream, and therefore form part of the stuttering. Since the non-linear output bytes can be shown to be extremely close[3] to uniformly distributed, a

---

[3] It is a property of maximal length Linear Feedback Shift Registers that every possible state appears exactly once during each cycle except for the state of all zeros. This yields an

simple analysis shows that for every byte of output from the stream generator, on average the register will be cycled $2^1/_3$ times, and a nonlinear output will be calculated $1^1/_3$ times. On a Sun Ultra workstation at 167MHz, using the code shown below and compiled with *gcc*, bytes can be generated and XORed into a one megabyte buffer at over 3 Mbytes/second, which is about 55 clock cycles per byte.

The underlying LFSR with its maximal length recurrence relation has a period of $(2^8)^{17}-1$ outputs before repeating. Soon after the LFSR outputs repeat, one of the stutter control bytes will repeat, at which time the stream will also begin repeating. It is clearly impossible for any sequence of output longer than 17 octets to repeat unless the contents of the Linear Feedback Shift Register also repeats. Thus the minimum stream output before SOBER repeats is $^3/_7((2^8)^{17}-1)$ bits (approximately 4×10^{40}) octets.

The memory requirements of SOBER are minimal. As shown, static memory for the register and stutter control occupy 37 bytes, and the post-keying state of the register is a further 17 bytes. Using circular buffers, and rekeying as necessary, the total can be reduced to only 20 bytes. There are 512 bytes of pre-initialised tables, and the code compiles to less than 512 bytes on popular architectures.

6. Security Analysis

An unknown plaintext attack on a stream cipher uses statistical abnormalities of the output stream to recover plaintext, or to attack the cipher. The LFSR underlying SOBER has very good statistical properties, which are preserved and enhanced by the subsequent operations, so there is no significant risk of an unknown plaintext attack. This analysis concentrates on vulnerability of SOBER to known-plaintext attacks.

Using $GF(2^8)$ instead of $GF(2)$ in the shift register has very little effect on the properties of the register itself. Herlestam [4] shows that the individual bits of this shift register go through the same sequence as if they were generated by a register over $GF(2)$ with the same total state of 136 bits. The different bit positions in the bytes are merely offsets in the output sequence of that LFSR. Therefore, recovering any linearly independent set of 136 distinct bits, or linear functions thereof, from any known positions in the output sequence yields the state of the register. It is no surprise then that the security of this cipher rests entirely upon the interaction of the nonlinear function and the stuttering.

The carry bits in the addition modulo 256 account for most of the nonlinear behaviour in the function. As there are five quantities being added, carries from lower bits add quite complicated functions of many other bits. If the elements were simply added, there would be no carry input to the least significant bit of the sum, which means that it is equivalent to XOR, and entirely linear. To defeat this, a left rotation of one of the partial sums brings in a bit with a high degree of non-linearity to disguise the linearity of the least significant bit of the remaining sum.

---

extremely tiny bias (2^{-128}) against zero bytes in the register. Ignoring this, it is easy to demonstrate that unbiased input to the non-linear function generates unbiased outputs.

The stuttering of the output provides a degree of uncertainty regarding the position in the register sequence of particular outputs. Some simpler systems using decimation have proven to be insecure. We have no convincing argument that this particular scheme is secure. However, most decimation schemes use the linear state of a register, while this one utilises a much more complicated, nonlinear, state (one of the major benefits of a software implementation).

Associated with the stuttering function, the output bytes are XORed with constants (0x69, 0x96, 0x00). A particular form of enumeration attack is significantly complicated by these trivial computations, as the amount of state information grows exponentially.

Lastly, statistical tests have been used to bolster confidence that nothing entirely stupid has been done. SOBER outputs have been examined using CRYPT-XS [2] , with no anomalous results reported. A total of 27 files each 8192 bits in length were tested together, generated as follows:

- 8 files were generated with the single byte key '0x0*n*', *n* in [1..8], and frame 0.

- 8 files were generated with the single byte key '0x0*n*', *n* in [1..8], and frame 1.

- These files were pairwise XORed, to yield a further 8 files. If there was inadequate avalanche from keying, these files should have shown something wrong.

- Two files were generated containing only the least significant bits of each octet from the original 16 files. These should have shown any problems with linearity in the least significant bits.

- For completeness, the XOR of these latter two files was also tested.

Over 95% of the individual tests were passed at the 95% confidence level. All of the tests passed at the 99% confidence level. Further testing, with CRYPT-XS and Marsaglia's DIEHARD [5] package, is ongoing.

6.1 More detailed security analysis

SOBER was designed to be used in either of two ways.

1. To be keyed once, with a secret key, and then a single stream of output produced. Synchronisation, which is necessary with stream ciphers, would be guaranteed externally.
2. To be keyed once with a secret key, and then to be rekeyed for each frame to the encrypted or decrypted with a non-secret frame counter, as is done in the GSM system.

In the discussion below, these are referred to as single level keying and two level keying, respectively. Wherever the algorithm can perform differently in these two cases, the difference is noted.

6.2 Rate of Output Production

SOBER works, in its last stage, by decimating ("stuttering") an output stream produced by the earlier stages. This is done by using one octet of output (the *stutter control byte*) to determine the behavior of the next four stages. On average, each such byte controls the consumption of 6 values from the earlier stages (minimum 4, maximum 8) and produces 3 output values (minimum 0, maximum 4). Counting the stutter control byte itself, over a long term average SOBER will produce $^{3}/_{7}N$ octets of output for N cycles of the underlying register. (In the published paper, the constant was erroneously specified as $^{4}/_{9}$, which was correct for an earlier version of SOBER and somehow survived many stages of review. All I can say is that I was closer than Painter, whose count of 24 pairs of chromosomes in the human cell survived unchallenged for over 20 years.)

One potential criticism of SOBER is that the time it takes to encrypt a particular amount of data is variable, because of the stuttering. We therefore undertook detailed simulation of the variability in the time it takes to encrypt 34 octets of data (one frame in each direction for our application; the more data is generated, the more even the time taken). The simulation was done by taking 256 random 4-byte keys, and 4096 different frame key inputs for each secret key, and recording the various statistics relating to the production of 34 octets of output. Fortunately SOBER is fast; the generation of the 34 Megabytes of output and associated statistics took just a few CPU seconds on a Sun workstation. The following table summarises the results.

Production of 34 octets of output	Register cycles	Stutter control bytes	Non-linear computations
Mean	79.7	11.7	45.7
Standard Deviation	5.5	1.01	1.01
Maximum / 99.99%	116 / 104	18 / 16	52 / 50
(Max, 99.99%) / Mean (%)	145% / 130%	153% / 136%	113% / 109%

The column for stutter control bytes is included for interest only. On the two architectures we have SOBER timings for (Pentium, Sun UltraSPARC), the algorithm spends roughly equal amounts of time on average computing shift register cycles and non-linear output values, with the rest of the overhead being constant and negligible. (This is obviously architecture dependent.) Assuming equal proportions of time on average, it can be seen that the worst case over 1M frames takes 29% longer than average, and 99.99% of frames take no more than 20% longer than average, to generate. It is straightforward to extrapolate these numbers given detailed performance measurements of particular architectures. In a bench test of SOBER in an actual CDMA mobile this variability was about 1.5% of the available CPU.

When two-level keying is in use, note that the frame key setup takes a constant time (additional to the above) of 19 each of register cycles and nonlinear computations. This works to reduce the variability, while increasing the per-frame computation a further 30%.

We conclude that the variability of time to encrypt frames is not an important issue. Note that Kocher's timing attacks cannot be applied to any synchronous system as the variability is disguised, unless the algorithm is so slow that it sometimes can't encrypt at all.

6.3 Cycle structure of SOBER

Being a stream cipher, SOBER works by generating a pseudo-random output sequence based only on secret internal state. As this state is finite, it follows that the output sequence must be ultimately periodic.

The LFSR upon which SOBER is based has well understood and proven properties. The states of the LFSR can be visualised as a large circle with many equidistant points (representing the discrete states of the LFSR) around the circumference. The stuttering operation causes some of these states to be skipped in the observable SOBER output.

It can be shown that the period of the underlying LFSR in SOBER is 2^{136}-1, and therefore the ultimate period of the output of SOBER is around $^3/_7(2^{136}-1)$. This presupposes that sequences which start with close LFSR states will eventually chance upon the same state to use for stutter control, and once such a collision happens the SOBER outputs will be synchronised. If the number of register cycles before synchronisation is relatively small, SOBER's output will form a single cycle with small tails leading into it. If the number of states before synchronisation is sufficiently large, SOBER would form multiple disjoint cycles. We expected the former situation to be the case, and indeed it is. This behavior is called "the Kruskal Count".

Simulation of 256 distinct, random, 4 byte keys was undertaken. For each key, SOBER was run from the initial state, and the state obtained by *n* cycles of the LFSR, for $n \in \{1..8\}$. The number of output octets before synchronisation of the output streams was recorded. The average was 23.4, with a standard deviation of 25.8. The longest such stream out of the 2048 samples was 166 octets.

Thus we conclude that SOBER output does form a single cycle, and an average tail length of 23.4. This cycle length is approximately 3.73×10^{40}, or just under 2^{135}. $^1/_7{}^{th}$ of the initial states are already in the cycle, that is, have a tail length of 0.

6.4 Regarding keys

SOBER is specified to use keys between 4 and 16 octets (32 to 128 bits). (Note, again, that the effective strength as specified is around 96 bits, not 128.) Keys of shorter lengths are distinct from longer, padded, keys. In fact, the lower bound of 4 octets is not important; we just felt that it was asking for trouble to allow smaller keys. The number of distinct keys, then, is given by: $N_k = \sum_{i=1..16} 2^{8i}$. This is close enough to 2^{128} that for the rest of the discussion we will use that value. Note that the keyspace is 2^{-8} times the state space. The following discussion treats SOBER as if the entire available keyspace is being utilised, when the reality is that, for design purposes, we are likely to be restricted to a very small subset of that space.

6.4.1 Single level keying

When single level keying is used, distinct keys guarantee to produce distinct initial states of the register. This does not, however, necessarily produce a distinct output sequence. When the first stutter control octet is zero, no output is produced and the register is cycled four more times. The output sequence from that initial state is the same as the output sequence from the subsequent state. The probability that the first stutter control octet is zero is 2^{-8}, and the probability that the resulting state is one which can be produced as the initial state from some other (128 bit) key is also about 2^{-8}, meaning that approximately 1 in 2^{16} keys is weak, in the sense that the sequence produced by it is the same as that produced by some other key. If the length of the keys is reduced, the probability of these weak keys is also reduced.

When a long enough output sequence is generated from a single keying operation, there is a possibility of overlap from the end of one sequence and the beginning of another. Because of SOBER's large state space, this is not expected to be a problem. By the birthday paradox, we would expect to need 2^{68} plaintexts longer than 132 octets (${}^{3}/_{7}\ 2^{8} + 23$) to have a significant probability that a single overlap had occurred. This problem exists for many conventional stream ciphers, and has not been perceived as a weakness for A5, for example. We conclude that it is not a weakness for SOBER either.

6.4.2 Two level keying

When two level keying is being used, the situation is slightly different. Now the total key space is 2^{160}, significantly greater than the total state space. Obviously there are now many classes of equivalent (primary and frame) keys, and sets of keys which produce "close" initial states. Some observations, however, can be made.

- For a given secret key, no pair of frame keys can produce the same initial state.
- Assuming frames of a size similar to the observed tail length (34 and 23 octets respectively), if initial states were close and overlap did occur, it will be on average 11 octets, and this will not be significantly more likely than random agreement.
- It is possible that the register can now start in the all-zero state. The probability of this occurring is 2^{-136}, which is so unlikely that we don't consider it worth checking for. If it is deemed necessary, though, it is trivial to check and correct. The probability that a given secret key *can* lead to a zero initial state is 2^{-104}, and for such a key, exactly one frame key will produce the all zero initial state.

6.5 Attacks against Components of SOBER

Unknown plaintext attacks against stream ciphers work by using statistical abnormalities (that is, differences from the distribution of a uniform random variable) of the ciphertext stream to discover information about the plaintext, or to recover the key or past or future values of the encryption stream. Because of the proven properties of the LFSR in SOBER, we know that many aspects of the distribution of its output are essentially ideal. The non-linear stage, and subsequently the stuttering, both work to defeat affine and higher order relationships in the encryption stream. At this time, no exploitable statistical characteristics are known or conjectured.

Known plaintext attacks against a stream cipher essentially start with some amount of encryption stream output, and the goal given such output is to recover the key (or, equivalently in this case, the initial state of the register), or to be able to predict past or future output from the cipher. In the case of SOBER, these are thought to be the same goal. The three major components of SOBER are the shift register, the nonlinear function, and the stuttering. These all contribute to the strength of the cipher in different, complementary, ways.

The Linear Feedback Shift Register itself is not cryptographically strong; its contribution is primarily to provide good statistical properties as mentioned above. As mentioned in the original paper, the individual bit positions of the LFSR act as if they are bits of a 136-bit binary LFSR.

The non-linear function by itself is also not particularly strong. Fast correlation attacks are conjectured to recover the initial state from a few thousand known octets of output from the nonlinear function, and we are aware of an attack which exploited the near-linearity of the least significant bits in the earlier design (in the published paper) to recover the initial state from a few more than 136 consecutive octets. Both of these attacks are completely defeated by the stuttering, and attempting to guess the positions in the output stream very quickly exceeds the brute-force complexity.

The stuttering by itself is not very strong, if it was applied to purely linear output. Edit distance correlation attacks would appear to be applicable, although the large state space makes them somewhat dubious in practice. A divide-and-conquer attack which proceeds by guessing enough stutter control bytes to form 136 linear equations to be solved requires about 2^{34} guesses. Both of these attacks are in turn frustrated by the nonlinearity of the inputs, and are thought to be infeasible. The fact that the stutter control octets are derived from the non-linear function contributes to the strength of the stuttering.

Because there is a single, monolithic shift register in SOBER, general divide and conquer attacks do not seem to be applicable.

7. Conclusion

SOBER is a conservatively designed stream cipher with a very small software footprint, designed for embedded applications in wireless telephony. Software implementations of LFSRs over $GF(2^n)$ can be extremely efficient, allowing well tried design principles to be brought to bear in software ciphers.

References

1. See Ross Anderson's posting on USENET newsgroup *sci.crypt*, "Subject: A5 (Was: HACKING DIGITAL PHONES)", 17 Jun 1994,. Alternatively, S. B. Xu, D. K. He, and X. M. Wang, "An implementation of the GSM General Data Encryption Algorithm A5", *CHINACRYPT '94*, Xidian, China, 11-15 November 1994, pp 287-291 (in Chinese). The latter appears to be based on the same information as Anderson's posting (or possibly the posting itself) as Anderson states that two of the registers have unknown polynomials, but the polynomials are the same in his posting and Xu et.al.

2. W. Caelli, E Dawson, L. Nielsen, H. Gustafson, "CRYPT-X Stastical Package Manual, Measuring the strength of Stream and Block Ciphers", *Queensland Univeristy of Technology*, 1992, ISBN 0-86856-8090.

3. J. Dj. Golic, "On Security of Nonlinear Filter Generators", *Proc. Fast Software Encryption 1996 Cambridge Workshop*, Springer-Verlag 1996.

4. T. Herlestam, "On functions of Linear Shift Register Sequences", in Franz Pichler, editor, *Proc. EUROCRYPT 85*, LNCS 219, Springer-Verlag 1986.

5. G. Marsaglia, "DIEHARD", http://stat.fsu.edu/~geo/diehard.html

6. A. Menezes, P. Van Oorschot, S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1997, Ch 6.

7. C. Paar, Ph.D. Thesis, "Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields", *Institute for Experimental Mathematics, University of Essen*, 1994, ISBN 3-18-332810-0.

8. B. Schneier, "Applied Cryptography Second Edition", *Wiley 1996*, pp. 369-413.

9. TIA/EIA Standard IS-54B, Telecommunications Industry Association, Vienna VA., USA.

10. Codes & Ciphers Ltd, "Some SOBERing Remarks", unpublished report.

Appendix: Excerpt of SOBER Source Code in C

Some source code of SOBER is below. The two multiplication tables, a header file, and a test harness are not shown, but can be obtained from the author or from the Web at `http://www.home.aone.net.au/qualcomm`. Also available at that site is *ssmail* (an enhancement to *Sendmail* which encrypts messages while in transit between consenting agents. It utilises SOBER.)

When initialised with the 8 byte ASCII key "test key", and a frame number of 1, the first 16 bytes of output should be (in hexadecimal) 62 a2 a6 38 ae ed 15 c5 13 7c 45 7b 2c e9 b5 ee.

```
/* Header for SOBER:  Seventeen Octet Byte Enabled
Register stream cipher. */

#define N 17

unsigned char Initial[N];   /* Initial contents of
                              register -- key schedule */
unsigned char R[2*N];       /* Working storage for the
                              shift register */
unsigned char stcnt;        /* stutter control -- when 0
                              next output stored */
unsigned char stctrl;       /* used 2 bits at a time */
int r;                      /* current offset in sliding
                              window buffer */

void sober_key(unsigned char key[], int keylen);
void sober_genbytes(unsigned long frame, unsigned char
buf[], int nbytes);

/* multiplication tables */
#include "tab99.h"
#include "tab206.h"

/*
 * FOLD is how many register cycles need to be performed
 * after combining the last byte of key, before every
 * byte depends on every
 * byte of the key, and is polynomial (and length)
 * dependent.
 */
#define FOLD 11

/* end of SOBER header */

/* cycle the contents of the shift register */

static int cycle(unsigned char *R, int r)
{
    R[r-N] = R[r] = mul99[R[r-N]] ^ R[r-N+4] ^
mul206[R[r-N+15]];
    if (++r == 2*N)
        r = N;
    return r;
}

/* Return a non-linear function of some parts of the
 * register.
 * Two octets are combined using add mod 256, then
 * rotated one place left.
 * Then three more octets are added.
 * The positions of the state bytes form a maximal span
 * full positive
 * difference set, and are 0, 1, 6, 13, 16.
```

```
 */

static unsigned char
nltap(unsigned char *R, int r)
{
    unsigned char   t;

    t = R[r-N] + R[r-N+16];
    t = (t << 1) | (t >> 7);
    return (unsigned char) (t + R[r-N+1] + R[r-N+6] +
R[r-N+13]);
}

/* calculate initial contents of the shift register */

void
sober_key(unsigned char key[], int keylen)
{
    int i;

    /* fill the register with fibonacci numbers */

    R[0] = R[1] = 1;
    for (i = 2; i < N; i++)
        R[i] = R[i-1] + R[i-2];

    /* initialise the pointers and start folding in key*/

    r = N;
    for (i = 0; i < keylen; i++)
    {
        R[r-N+15] += key[i];
        r = cycle(R, r);
        R[r-N+8] ^= nltap(R, r);
    }

    /* also fold in the length of the key */

    R[r-N+15] += keylen;
    for (i = 0; i < FOLD; ++i)
    {
        r = cycle(R, r);
        R[r-N+8] ^= nltap(R, r);
    }

    /* save state */

    for (i = 0; i < N; i++)
        Initial[i] = R[r-N+i];
}

/* Fold in the per-frame key */

void
```

```
sober_setframe(unsigned long frame)
{
    register int    i;

    /* copy initial contents */

    for (i = 0; i < N; i++)
        R[i] = Initial[i];

    /* fold in the frame sequence */

    r = N;
    for (i = 0; i < 4; i++)
    {
        R[r-N+15] += (unsigned char)(frame & 0xff);
        frame >>= 8;
        r = cycle(R, r);
      R[r-N+8] ^= nltap(R, r);
    }

    /* run the register for a while */

    for (i = 0; i < FOLD; i++)
    {
        r = cycle(R, r);
      R[r-N+8] ^= nltap(R, r);
    }
    stcnt = 0;
}


/* XOR pseudo-random bytes into buffer */

void
sober_gen(unsigned char *buf, int nbytes)
{
    unsigned char   *endbuf;

    endbuf = &buf[nbytes];
    while (buf != endbuf)
    {
        stctrl >>= 2;

        /* reprime stuttering if necessary */

        if (stcnt == 0)
        {
            stcnt = 4;
            r = cycle(R, r);
            stctrl = nltap(R, r);
        }
        stcnt--;

        r = cycle(R, r);
```

```
        switch (stctrl & 0x3) {

        case 0: /* just waste a cycle and loop */
            break;

        case 1: /* use the first output from two cycles
*/
            *buf++ ^= nltap(R, r) ^ 0x69;
            r = cycle(R, r);
            break;

        case 2: /* use the second output from two cycles
*/
            r = cycle(R, r);
            *buf++ ^= nltap(R, r);
            break;

        case 3: /* return from one cycle */
            *buf++ ^= nltap(R, r) ^ 0x96;
            break;

        }
    }
}

/* encrypt/decrypt a frame of data */

void
sober_genbytes(unsigned long frame, unsigned char *buf,
int nbytes)
{
    sober_setframe(frame);
    sober_gen(buf, nbytes);
}
```