The t-Class of SOBER Stream Ciphers

Greg Rose, Philip Hawkes

QUALCOMM Australia, Suite 410, Birkenhead Point, Drummoyne NSW 2047, Australia
{ggr, phawkes}@qualcomm.com

This paper proposes the t-class of SOBER stream ciphers: t8, t16 and t32. t8, t16 and t32 offer 64-, 128- and 256-bit key strength respectively. The t-class ciphers are based on the same principles as the original SOBER family: SOBER [17], SOBER-II [18], S16 and S32 [19], utilising the structure SOBER-II and S16 are based. The t-class ciphers are software stream ciphers designed for software implementation. Changes between the t-class and the original SOBER family are centred around constructing a stronger non-linear filter and more secure key loading. Also, t32 is now based on the same structure as t8 and t16 (S32 had a different structure). Much of the analysis of SOBER-II and S16 applies to the t-class; this paper discusses the applications of such analyses to the t-class and introduces further analyses. The output streams from these ciphers have proven to perform well in all statistical tests.

1. Introduction

The t-class of SOBER ciphers: t8, t16, and t32; are fast, software-oriented stream ciphers designed for implementation on a general purpose CPU. t8, t16 and t32 is designed for keys up to 64, 128 and 256 bits in length respectively. The cipher t*w*, $w \in \{8,16,32\}$, is designed for *w*-bit processors, uses operations on *w*-bit words and accepts a secret session key with up to *w* bytes. For example, t8 is designed for 8-bit processors, the operations in t8 act byte-wise and the session key can be up to 8 bytes (64 bits) in length. Table 1 compares the t-class ciphers. The t-class ciphers are based on the same principles as the original SOBER family: SOBER [17], SOBER-II [18], S16 and S32 [19].

Cipher	Maximum Key Length (bits)	Encryption Speed (Mbps)	
		Unix 200	Pentium 233
t8	64	~25	~24
t16	128	~44	~25
t32	256	~76	~80

Table 1. Maximum key length and encryption speed for the t-class ciphers. The encryption speed was measured on a 200MHz Sun Ultra Sparc and a 233MHz Pentium using unoptimised code.

The SOBER family originated with the design of SOBER [17]: a software stream cipher designed to meet the needs of embedded applications such as voice encryption in wireless telephones which place severe constraints on the amount of processing power, program space and memory available for software encryption algorithms. Since most of the mobile telephones in use incorporate a microprocessor and memory, a software stream cipher that is fast and uses little memory would be ideal for this application. 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. As discussed in Section 2, such ciphers are difficult to implement efficiently on a general purpose CPU. SOBER family ciphers overcome this dilemma by utilising a LFSR defined over $GF(2^w)$ and a number of techniques to greatly increase the generation speed of the pseudo-random stream in software on a microprocessor. Following various analyses [1,3,5,9] of SOBER, the algorithm was updated to *SOBER-II* [18], and two SOBER variants: *S16* and *S32*; were proposed [19]. S16 is a straightforward enhancement of SOBER to 16-bit wide arithmetic, while S32 is an extension of the design principles of SOBER to 32-bit wide arithmetic. Further analyses of SOBER-II are found in [2,4,10].

These analyses of SOBER-II found attacks that are specific to the overall structure of the cipher, rather than exploiting a weakness of the individual components used in the cipher. It is relatively simple to determine the security of the cipher formed by replacing the 8-bit words by 16-bit or 32-bit words. In designing the t-class ciphers we aimed to provide ciphers which utilized key of up to 64, 128 and 256 bits in length, and for which there is no know attack with complexity less than an exhaustive key search. The best attack on SOBER-II has a complexity in excess of a 64-bit search; the best attack on a 16-bit (32-bit) version of this cipher would then have complexity in excess of 128-bit (256-bit) key search. Therefore, we decided that the t-class should preserve the overall structure used in SOBER-II. As SOBER-II has been the subject to public scrutiny for some time, this also gives us some confidence in the structure being used.

The t-class stream ciphers have four components: *key loading*, an LFSR, a *non-linear filter* (*NLF*) and *stuttering*, as shown in Figure 1. The key loading sets the 17 words in the register of the LFSR to an *initial state* derived from the key. In some cases a re-synchronisation key or frame key is used during key loading. This is discussed in further detail in Section 5. The LFSR uses a *linear feedback function* to construct a stream of words from the initial state; this stream of words is the *LFSR stream* $\{s_n\}$. The process of producing a new word in the LFSR stream is called a *cycle* of the LFSR. The purpose of the NLF is to disguise the linearity in the LFSR stream. After every cycle of the LFSR, the NLF combines words from the register in a non-linear function; the outputs form the *NLF stream* $\{v_n\}$. The stuttering uses occasional NLF stream words to select NLF stream words to be used in the key stream $\{z_j\}$.[1]

The t-class cipher differ from the original SOBER family in the following areas:

- t32 now uses the same structure as the rest of the t-class, while S32 had a different structure to SOBER-II and S16.

[1] This process is also known as *decimation.*

- The NLF has been significantly strengthened with the use of a nonlinear *S*-box and the inclusions of a key-dependent constant as an additional variable.
- The key loading has changed in two ways.
  - The frame is now loaded as if it were a 4-octet key.
  - The key loading has been strengthened to eliminate algebraic relationships that existed between words of the initial state of the LFSR in SOBER-II.

The paper is set out as follows. First, the LFSRs used in the t-class are defined in Section 2. Section 3 describes the NLF and explains how the overall structure of the t-class was found. Section 4 describes the stuttering and gives some results regarding the period of the key stream. The key loading is described in Section 5. Section 6 considers the memory requirements and performance results for the t-class, and an overall security analysis is provided in Section 7.

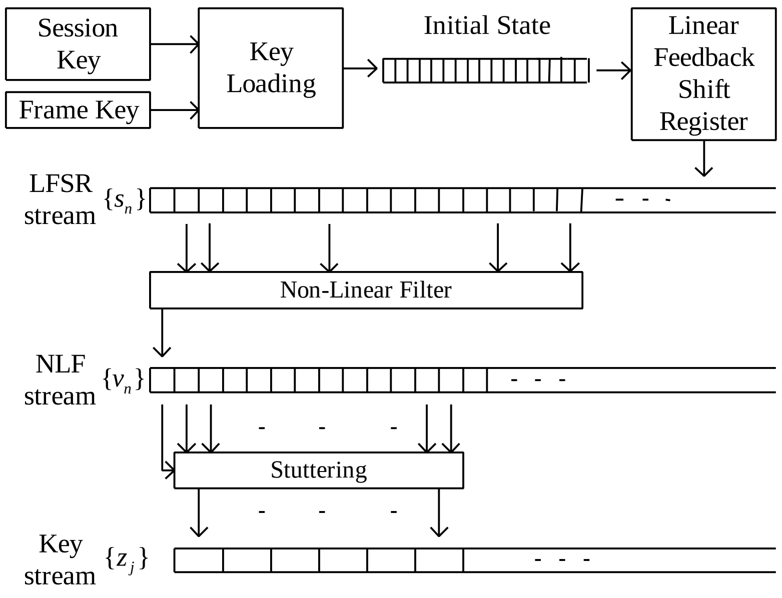

Figure 1. The components of the t-class SOBER stream ciphers.

2 The Linear Feedback Shift Register

A linear feedback shift register (LFSR) is typically based on a recurrence relation over the Galois Field of order 2 (*GF*(2)). The output sequence (of bits) is defined by

$$s_{n+k} = c_{k-1}s_{n+k-1} + c_{k-2}s_{n+k-2} + \cdots + c_1 s_{n+1} + c_0 s_n \,, \tag{2.1}$$

where s_n is the *n*-th output of the sequence, the constant coefficients c_i, $0 \le i \le k-1$, are elements of $GF(2)$, that is, single bits, and *k* is called the *order* of the recurrence relation.

The LFSR is typically represented by the polynomial

$$p_1(x) = x^k + c_{k-1}x^{k-1} + c_{k-2}x^{k-2} + \cdots + c_1 x + c_0\,, \tag{2.2}$$

where $p_1(x)$ indicates that multiplication and addition are over $GF(2^1)$. 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 that utilise LFSRs as their underlying mechanism; for good surveys, see [12] or [14].

2.1 LFSR over GF(2^w)

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. Particularly good choices for such a field are the Galois Field with 2^w elements (*GF(2^w)*), where *w* is related to the size of items in the underlying processor, usually bytes or 16- or 32-bit words. The elements of this field and the coefficients of the recurrence relation occupy exactly one unit of storage and can be efficiently manipulated in software. In the meantime, the order *k* of the recurrence relation that encodes the same amount of state is reduced by a factor of *w*.

The field $GF(2^w)$ can be represented (the *standard representation*) as the modulo 2 coefficients of all polynomials with degree less than *w*. That is, an element *a* of the field is represented by a *w*-bit word with bits $(a_{w-1}, a_{w-2}, \ldots, a_1, a_0)$, which represents the polynomial

$$a_{w-1}x^{w-1} + a_{w-2}x^{w-2} + \cdots + a_1 x + a_0\,.$$

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 w. The multiplicative identity element is $(a_{w-1}, a_{w-2}, \ldots, a_1, a_0) = (0, 0, \ldots, 0, 1)$. The choice of an irreducible degree w polynomial alters the way elements of the group are mapped into encoded words on the computer, but does not otherwise affect the actual group operations. The original SOBER family and the SOBER t-class ciphers use the irreducible polynomials shown in Table 2.[2]

[2] The polynomials used here are those chosen by *Mathematica* for its default representation of the finite fields. The fields resulting from different choices of irreducible polynomials are isomorphic, and so the representation is largely irrelevant to the subsequent analysis.

w	Irreducible Polynomial used in tw	Hexadecimal Representation
8	$x^8+x^6+x^3+x^2+1$	0x14D
16	$x^{16}+x^{14}+x^{12}+x^7+x^6+x^4+x^2+x+1$	0x150D7
32	$x^{32}+(x^{24}+x^{16}+x^8+1)(x^6+x^5+x^2+1)$	0x165656565

Table 2: The irreducible polynomials used in tw, $w \in \{8,16,32\}$.

Now that there is a known representation for the elements of the underlying field which can be stored in a single computer unit, the LFSR can be specified in terms of bytes or words instead of bits, and successive output values will also be those units rather than bits. The feedback function is still of the form of equation (2.1), however the values s_n and the coefficients c_i are elements of $GF(2^w)$, rather than bits, and addition and multiplication performed over $GF(2^w)$ as described above. Cycling the LFSR requires a number of constant multiplication operations followed by XOR of these terms. Note that multiplication by 0 simply means ignoring the corresponding element of the register, while multiplication by 1 requires no computation, as the output (of the multiplication) is the same as the input. Thus, to enable an efficient implementation, the multiplication constants c_i, $0 \le i \le k-1$, are chosen to be 0 or 1 most of the time. Multiplication with any other constants c_i is implemented using pre-calculated tables stored in read-only memory, see [16,18] for details.

The t-class ciphers use LFSRs of the form

$$s_{n+17} = a\, s_{n+15} \quad \oplus \quad s_{n+4} \quad \oplus \quad b\, s_n\,, \tag{2.3}$$

where $\oplus$ denotes addition over $GF(2^w)$ (equivalent to w-bit XOR), multiplication is performed over $GF(2^w)$, and a, b are non-zero. This is the same form as used in SOBER-II and S16, indeed the feedback function for t8 (t16) is identical to that used in SOBER-II (S16). The feedback functions are shown in Table 3. The LFSR is represented by a polynomial over $GF(2^w)$:

$$p_w(x) = x^{17} \oplus a\, x^{15} \oplus x^4 \oplus b\,, \tag{2.4}$$

where the subscript w indicates that the addition and multiplication is over $GF(2^w)$.

w	Feedback Function for tw
8	$s_{n+17} = \text{0xCE}\, s_{n+15} \quad \oplus \quad s_{n+4} \quad \oplus \quad \text{0x63}\, s_n$
16	$s_{n+17} = \text{0xE382}\, s_{n+15} \quad \oplus \quad s_{n+4} \quad \oplus \quad \text{0x67C3}\, s_n$
32	$s_{n+17} = s_{n+15} \quad \oplus \quad s_{n+4} \quad \oplus \quad \text{0xC2DB2AA3}\, s_n$

Table 3: The feedback functions for the t-class, where $\oplus$ denotes w-bit XOR and $\otimes$ denotes multiplication over $GF(2^w)$.

The LFSR over the field $GF(2^w)$ is mathematically equivalent to w parallel shift registers over $GF(2)$ of length equivalent to the total state 13w, each with the same recurrence relation but different initial state [10]. Let the polynomial $p_1(x)$ represent the LFSR over $GF(2)$. The multiplication constants chosen minimise the number of coefficient not equal to one, such that the following properties are satisfied.

- **The LFSR has maximum length period**. The period has a maximum length of $(2^{13w}-1)$ when $p_1(x)$ is a *primitive* polynomial of degree 13*w*, that is, it divides x^d+1, for $d=2^{13w}-1$, but not for any *d* that divides $(2^{13w}-1)$.
- **Approximately half of the coefficients of $p_1(x)$ are 1**. This condition is ideal for maximum diffusion and strength against cryptanalysis.

The LFSR stream is never required in its entirety. To cycle the LFSR, only those elements $s_n,\ldots,s_{n+16}$ are required. After *n* cycles of the LFSR, it is only necessary to preserve the elements $s_n,\ldots,s_{n+16}$: the value of these elements is called the *state* of the register. The state of the register corresponding to the elements $s_n,\ldots,s_{n+16}$ at any given instant in time is denoted $r_0,\ldots,r_{16}$. When the LFSR is cycled, the values in positions r_0, r_4 and r_{15} (s_n, s_{n+4} and s_{n+15}) are used to determine s_{n+17}. The values in positions $r_1,\ldots,r_{16}$ are shifted to the positions $r_0,\ldots,r_{15}$ respectively (that is, $r_i=r_{i+1}$, $15\geq i\geq 0$) and r_{16} is set to the value of s_{n+17}. The NLF then uses the values in the register as input to the nonlinear function (see Figure 2), as described in the following section. The implementation of the LFSR can be further enhanced using techniques discussed in [16].

3 The Non-linear Filter

Much of the cryptographic security of the t-class SOBER family resides in the non-linear filter (NLF) used to defend against attacks on the linear feedback or stuttering phase. It is therefore important to make this function as strong as possible without compromising the performance of the cipher.

The NLF takes the values from certain positions in the register as inputs; as in SOBER-II and S16, the t-class ciphers use the register elements $r_0, r_1, r_6, r_{13}, r_{16}$ as inputs. As the LFSR is cycled before the NLF is applied, the NLF stream word v_n depends on LFSR stream words $s_{n+1}, s_{n+2}, s_{n+7}\ s_{n+14}$ and s_{n+17}. The t-class NLF also uses a key-dependent constant word called *Konst* as an input value. This value is derived immediately after the secret session key is loaded, and remains the same even if the frame number changes.

In designing the NLF we gave ourselves certain restrictions. The processors for which the t-class ciphers are designed are likely to have restrictions on the amount of ROM available. Thus these ciphers can afford to use a S-box in the NLF, although it would be preferable for the S-box to use only a small amount of memory; we restrict the S-box to containing only 256 entries where each entry is a *w*-bit word. The NLF should also be balanced (that is, every output word occurs with equal probability).

Furthermore, we place the requirement if any five of the six inputs (including *Konst*) are fixed then every value for the remaining input corresponds to a unique output.

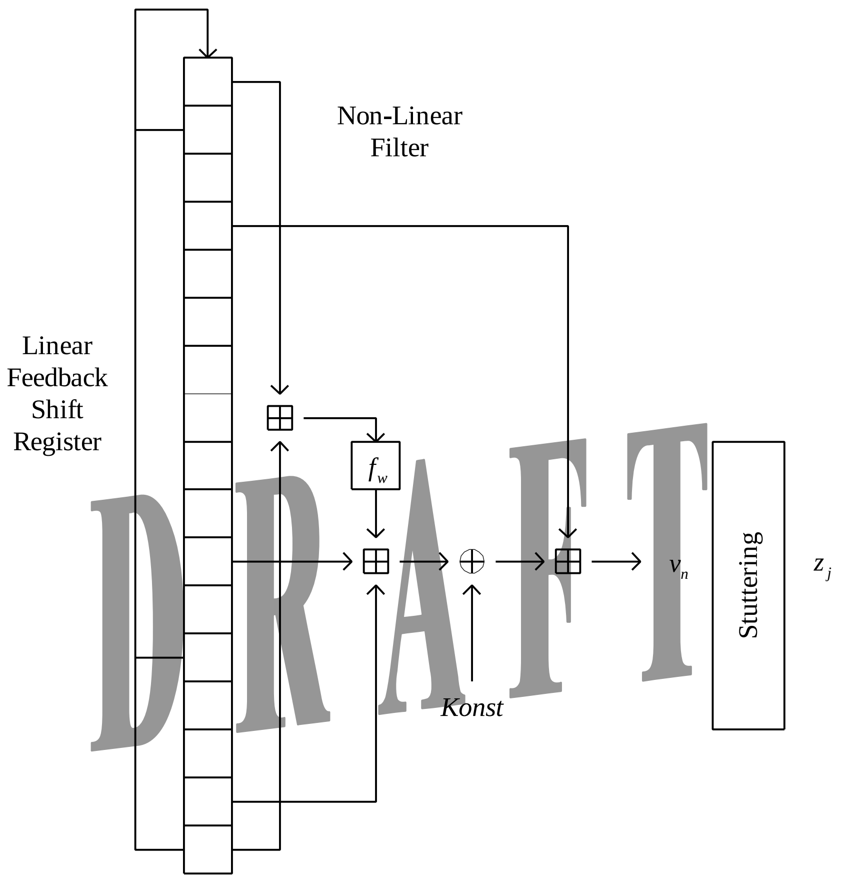

Figure 2. The block structure of the t-class ciphers

The NLF chosen for the t-class is of the form

$$v_n = ((f_w(r_0 + r_{16}) + r_1 + r_6) \oplus Konst) + r_{13}, \qquad (3.1)$$

as shown in Figure 2, where addition is modulo 2^w, and the function f_w changes for each value of *w*. In terms of LFSR stream words,

$$v_n = ((f_w(s_{n+1} + s_{n+17}) + s_{n+2} + s_{n+7}) \oplus Konst) + s_{n+14}\,. \qquad (3.2)$$

The function f_w serves three purposes. Firstly, it removes the linearity in the least significant bit and adds significantly non-linearity to the remaining bits. Secondly, it ensures that the addition of r_0 and r_{16} does not commute with the addition of r_1 and r_6. Thirdly it ensures that every bit of the output of the NLF depends on every bit of r_0 and r_{16}.

XORing the value *Konst* into the NLF has two purposes. Firstly, it increasing the complexity of any attack (excluding exhaustive key search) as there are now 2^{16} possible NLF functions. Secondly, the *Konst* is XORed (rather than added) so as to lower the probability of the addition of r_{13} commuting with the addition of r_1 and r_6. There is still a small probability that the operations will commute, but this probability is low and relies on the value of *Konst*. This issue requires further analysis.

3.1 The Function f_8 used in t8

The function f_8 used in t8 simply uses the *S*-box (or *f*-table) used in the Skipjack cipher.[3] That is, if *SJ* denotes the SkipJack *S*-box, then $f_8(X) = SJ(X)$. This *S*-box was chosen as it is high nonlinear, has been available for public scrutiny for some time. We also wish to foster confidence that we have not introduced a deliberately weak *S*-box.

3.2 The Function f_w used in t*w*, $w \in \{16, 32\}$

In t16 and t32, the f_w function has three parts, as shown in Figure 3.

The input to the S-box should depend on every bit of r_0 and r_{16}, so the eight most significant bits (MSBs) of $(r_0 + r_{16})$ are extracted to be the input (reference) to the 256 entry S-box. We call this operation *most significant byte extraction*, and denote the operation by *MSBE*.

The *S*-boxes for t16 and t32, denoted SB_{16} and SB_{32} respectively, are a combination of the Skipjack *S*-box and an *S*-box tailor-designed by the Information Security Research Centre (ISRC) at the Queensland University of Technology [6]. The eight MSBs of the output of SB_{16} and SB_{32} are defined by the Skipjack *S*-box. The remaining $(w-8)$ least significant bits (LSBs) of the output of SB_{16} and SB_{32} are defined by the *S*-boxes constructed by the ISRC. These *S*-boxes were constructed as $(w-8)$ mutually uncorrelated, balanced and highly non-linear single bit functions. The *S*-boxes SB_{16} and SB_{32} can be found at http://www.home.aone.net.au/qualcomm.

The output of $f_w(X)$, $w \in \{16, 32\}$, is determined as follows. Following MSBE, the most significant byte of *X* becomes the input to SB_w. The most significant byte is then removed from *X*, and this value is XORed with the *w*-bit output of the *S*-box. Thus, the most significant byte of the output of f_w, $w \in \{16, 32\}$, is the output of the Skipjack *S*-box, while the least significant $(w-8)$ bits are obtained by XORing the $(w-8)$ bits of the output of SB_w with the $(w-8)$ least significant bits of the input. The function f_w is defined this way to ensure that it is a highly non-linear

[3] See FIPS 185- Escrowed Encryption Standard at the following web page: `http://www.itl.nist.gov/fipspubs/fip185.htm`

permutation, while using only a single, small *S*-box. The function f_8 can be considered of the same form, noting that $w-8=0$.

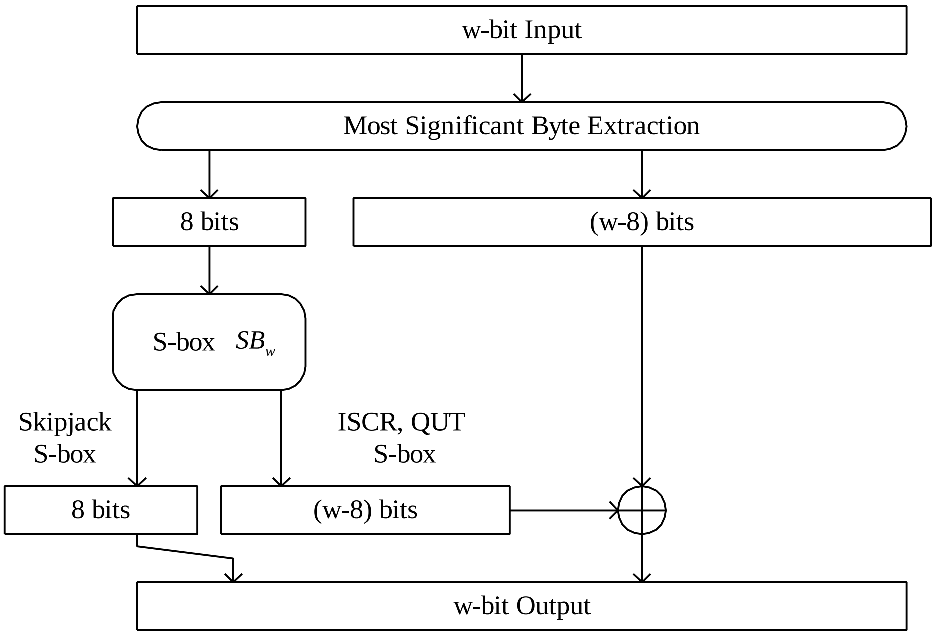

Figure 3. The function f_w used in t*w*, $w \in \{16.32\}$

3.3 Guess-and-Determine Attacks

Guess-and-Determine (GD) attacks are based on the assumption that the attacker knows part of a sequence of NLF outputs, and the attacker knows how many times the LFSR has been cycled between the NLF outputs. The stuttering destroys knowledge about the number of LFSR cycles; to account for this an attacker must *assume* the value of the *dibits* (two bit blocks) used to control the stuttering. The resulting attack is called an *assume-guess-and-determine* (AGD) attack. AGD attacks on SOBER-family stream cipher have been observed by Blackburn *et al.* [1], Bleichenbacher and Patel [3], Bleichenbacher, Patel and Meier [4] and Hawkes [10]. The two latter papers present attacks against the structure used in SOBER-II, S16 and the t-class ciphers; the attacks from both papers are of equal complexity. In this section, we focus only on the GD attack component of an attack, AGD attacks are discussed following the description of the stuttering.

Observation 1. Given the values of any three of the LFSR stream words $s_n, s_{n+4}, s_{n+15}, s_{n+17}$, equation (2.3) can be used to determine the value of the remaining word. ■

Observation 2. Suppose an attacker knows the value of v_t. Given the values of *Konst* and any four of the LFSR stream words $s_{n+1}, s_{n+2}, s_{n+7}, s_{n+14}, s_{n+17}$, equation (3.2) can be used to determine the value of the remaining LFSR stream word from the value of v_n. ■

Suppose an attacker observes a portion of the NLF stream and knows the relative juxtaposition of the NLF words. The basic format of a GD attack is as follows:

1. **Guess** *Konst* and some LFSR stream words.
2. **Determine** a full state of the register by determining further LFSR stream words through applications of Observations 1 and 2.
3. **Test** if this guess is correct by producing more of the NLF stream and comparing this with the observed NLF stream. If the NLF streams are not the same, then return to Stage 1.

Example 1. The GD component of the attack of Bleichenbacher, Patel and Meier [4] against SOBER-II also applies to the t-class. Suppose that an attacker observes an NLF stream which contains the values of v_{n+i}, $i \in \{-1,0,1,2,4,9,11,12,14\}$, for some n. The GD attack proceeds as follows:

1. Guess the values of *Konst* and s_{n+i}, $i \in \{1,5,6,10,14,15,16,21,25\}$; a total of 10 words.
2. Determine the values of s_{n+i}, $0 \le i \le 16$, using the process shown in Table 4. Table 4 uses the following notation. If applying Observation 1 using values for $s_{n+a}, s_{n+b}, s_{n+c}$ to determine s_{n+d} then we write this as $a,b,c \to d$. If applying Observation 2 using v_{n+x} and values for $s_{n+a}, s_{n+b}, s_{n+c}, s_{n+d}$ and *Konst* to determine s_{n+e} then we write this as $a,b,c,d \to_x e$. Following this stage the attacker has determined the state of full register: s_{n+i}, $0 \le i \le 16$.
3. Test if the guess is correct by generating the NLF stream (using this register state) and comparing with other NLF stream words. If the NLF streams are not the same, then return to Stage 1. ■

	Action		Action		Action
1	$1,5,16 \to 18$	9	$16,27,29 \to 12$	17	$13,17,28 \to 30$
2	$6,10,21 \to 23$	10	$12,23,25 \to 8$	18	$15,19,30 \to 32$
3	$10,14,25 \to 27$	11	$12,18,25,28 \to_{11} 13$	19	$1,2,14,17 \to_{0} 7$
4	$5,6,18,21 \to_{4} 11$	12	$1,6,11,16 \to_{-1} 0$	20	$1,8,15 \to 3$
5	$10,11,16,18 \to_{9} 26$	13	$13,14,26,29 \to_{12} 19$	21	$3,7,18 \to 20$
6	$11,15,26 \to 28$	14	$8,19,21 \to 4$	22	$3,4,16,19 \to_{2} 9$
7	$15,16,21,28 \to_{14} 31$	15	$0,4,15 \to 17$		
8	$14,18,31 \to 29$	16	$6,17,19 \to 2$		

Table 4. The "determining" steps in the GD attack on the t-class ciphers.

The complexity of an attack is proportional to the number of guesses. If $(x-1)$ LFSR stream words and the value of *Konst* are guessed in the Stage 1, then the number of possible guesses is $(2^{16})^x$. The GD attack in Example 1 has a complexity of 2^{10w}; for example, the attack on t16 has a complexity of 2^{160}. This is the lowest known complexity for a GD attack on a t-class cipher. In some GD attacks, a "divide-and-conquer" approach can be used to reduce the complexity. This is achieved by guessing *Konst* and a subset of *y* LFSR words ($y < x-1$), determining as many LFSR stream words as possible, and using these LFSR stream words to determine an NLF output which can be compared against the observed NLF stream. If the NLF output disagrees with the observed NLF output then this indicates that the guess (for *Konst* and the subset of *y* LFSR stream words) is incorrect. The attacker then eliminates such guesses, and only proceeds to guess the remaining $(x-y-1)$ words if the NLF output agrees. This process reduces the overall number of guesses. Such an approach was used to find a GD attack against unstuttered SOBER-II with a complexity of 2^{72} [10]; this GD attack on the t-class ciphers would have a complexity of 2^{10w}.

3.4 Extended Guess-and-Determine Attacks

Extended guess-and-determine (EGD) attacks use the same approach as GD attacks, however EGD attacks use additional linear relationships between LFSR stream words that result from the linear feedback function. While GD attacks have been known since late 1998, EGD attacks have only been noticed since August 1999, so less is known about these attacks.

Consider the polynomial $p_w(x)$ in equation (2.4) corresponding to the feedback function in equation (2.3). Let $q(x) = \sum_{i\geq 0} g_i x^i$ be some polynomial over $GF(2^w)$ and the product

$$r(x) = p_w(x)\cdot q(x) = d_k x^k + d_{k-1}x^{k-1} + \cdots + d_1 x + d_0, \tag{3.1}$$

where addition and multiplication is still over $GF(2^w)$. The polynomial product $r(x)$ corresponds to the following relationship between LFSR stream words:

$$d_k s_{t+k} = d_{k-1}s_{t+k-1} + \cdots + d_1 s_{t+1} + d_0 s_t .$$

In particular, this equation gives a relationship between those values of s_{t+i} such that $d_i \neq 0$.

Example 2. Consider the polynomial $p_w(x)$. If we set $q(x) = p_w(x)$ then the product is $r(x) = p_w(x)^2 = x^{34} + a^2 x^{30} + x^8 + b^2$. This polynomial indicates that $s_{t+34} = a^2 s_{t+30} + s_{t+8} + b^2 s_t$; which gives a relationship between the LFSR stream words s_{t+i}, $i \in \{0,8,30,34\}$. ■

EGD attacks are GD attacks that, in addition to exploiting the NLF and LFSR feedback function, also exploit these extra linear relationships in the same way that

Observation 1 exploits the feedback function in a GD attack. We are only concerned with linear relationships that will give information that could not have been derived using the original polynomial in equation (2.4). This motivates the following definitions.

Definition 1. A polynomial is *exploited* if it is used (as in Observation 1) to determine an LFSR stream word from the other LFSR stream words. ■

Definition 2. Let $r_1(x),\ldots,r_a(x)$ be polynomials over $GF(2^w)$. Suppose the polynomial $r(x)$ represents a linear relationship between s_{t+i}, $i \in S$ for some set *S*. Then $r(x)$ is said to be *resolvable* with respect to $r_1(x),\ldots,r_a(x)$ if it is possible to determine s_{t+j} from $\{s_{t+i}, i \in S\setminus\{j\}\}$ by exploiting the polynomials $r_1(x),\ldots,r_a(x)$. ■

Example 3. Consider $r(x) = p_w(x)\cdot q(x)$ in a t-class cipher where $q(x) = x^2 + \alpha$. When expanded, $r(x) = x^{19} + \alpha^2 x^{15} + x^6 + \alpha x^4 + \beta x^2 + \alpha\beta$, which is a relationship between the LFSR stream words s_{t+i}, $i \in \{0,2,4,6,15,19\}$. However, $r(x)$ is resolvable with respect to $p_w(x)$. For any $j \in \{0,2,4,6,15,19\}$, it is possible to determine s_{t+j} from the remaining values of s_{t+i} using the relationship between $\{s_t, s_{t+4}, s_{t+15}, s_{t+17}\}$ and $\{s_{t+2}, s_{t+6}, s_{t+17}, s_{t+19}\}$. ■

Example 4. The polynomial $r(x) = p_w^2(x)$ in the t-class is not resolvable with respect to $p(x)$. ■

A set of polynomials such that no polynomial is resolvable with respect to the others is called an *EGD basis*. The first step in looking for EGD attacks is to find an EGD basis: in particular looking for polynomials that are sparse (that is, they provide a relationship between only a few words). Then we must find the best EGD attack (that is, the attack with the lowest complexity) using this EGD basis. Recall that an EGD attack uses not only the EGD basis, but the NLF equation as well. The following example demonstrates why EGD attacks must be considered as a serious threat.

Example 5. We examined a SOBER-like cipher with an LFSR polynomial $p_w(x) = x^{13} + x^4 + x^1 + \lambda$, and an NLF of the form $v_t = f(s_t, s_{t+5}, s_{t+10}, s_{t+11})$. The best GD attack against the 16-bit word version of this cipher has a complexity of 2^{128} (ignoring stuttering). However, there also exist EGD attacks, using $p_w(x)$ and $p_w^2(x)$ as an EGD basis, which have a complexity of only 2^{112} (ignoring stuttering). ■

3.2.1 EGD Attacks on the t-class Ciphers

To date, we have found the following EGD basis for the t-class ciphers:

$$\begin{aligned} r_1(x) &= p_w(x) = x^{17} + \alpha x^{15} + x^4 + \beta, \\ r_2(x) &= p_w^2(x) = x^{34} + \alpha^2 x^{30} + x^8 + \beta^2, \\ r_3(x) &= p_w(x) \cdot (x^2 + \alpha)(x^4 + \beta) \\ &= x^{23} + (\alpha^2 + \beta)x^{19} + \alpha\beta x^{15} + x^{10} + x^8 + \beta^2 x^2 + \alpha\beta^2. \end{aligned}$$

Every possible EGD attack, using this EGD basis, has been tested. None of these EGD attacks has a complexity less than that of the best known GD attacks which have a complexity of 2^{10w} (when stuttering is ignored). This is not concrete proof of the resistance to EGD attacks, as it is still uncertain whether there are further linear relationships that are unresolvable with respect to this basis. We will continue to look for further linear relationships. We estimate that we have checked approximately half of the possible products of $p_w(x)$ that have degree less than or equal to 34; all have thus far proven to be resolvable with respect to the EGD basis above. We do not expect to find products of degree greater than 34 that are useful in an EGD attack.

We hope that in the near future we shall have found the complete EGD basis for the products of degree less than or equal to 34, which will then allow us to be more certain of the resistance of the t-class ciphers against every possible EGD attack. We would also like to determine if products of degree greater than 34 can serve any useful purpose in an EGD attack. However, at this stage, the best GD or EGD attacks have a complexity of 2^{10w} which is significantly large than the complexity of an exhaustive key search (2^{8w}).

4 Stuttering the Non-Linear Output

It is easily conceivable that the state of the LFSR could be used to efficiently reconstruct the state, particularly by a fast correlation attack. The task is made much more difficult if some of the states are not represented in the output, in a way that is difficult to predict (irregular decimation). This is the role of the stuttering in the t-class ciphers. While the stuttering formed the most ad-hoc part of the design of SOBER and SOBER-II, the stuttering has also been the source of least trouble in the security analyses done. In updating the SOBER family to the t-class, various other forms of decimation were considered. However, the original stuttering still appears to offer the best increase in security. Thus, the stuttering employed in the t-class is derived from the stuttering used in SOBER and SOBER-II.

Stuttering is based on occasional words of nonlinear output being used to determine the inclusion of other words in the output stream. When the generator is started, the first NLF output (NLF stream word) is taken to be used as a *stutter control word* (SCW). Each SCW is broken into pairs of bits called *dibits*, with the least significant dibit being used first. The dibits provide the cipher with instructions regarding how many times to cycle the LFSR, whether to output an NLF output, and how this value is included in the key stream. When the instruction from all the dibits have been performed, the LFSR is cycled and the NLF output from the register forms the next SCW.

There are four possible values for each dibit. The actions determined by the dibits are listed in Table 5. In this table, "C" denotes cycling the LFSR, while "O: *x*" denotes obtaining the NLF outputting, XORing this value with *x* and including this in the key stream. The values of e_w are defined as $e_8 = \text{0x69}$, $e_{16} = \text{0x6996}$, $e_{32} = \text{0xC5963A69}$, and e'_w is the bit-wise complement of e_w, $w \in \{8,16,32\}$.

Dibit	(0,0)	(0,1)	(1,0)	(1,1)
Action	C	C, O: e_w, C	C, C, O: 0	C, O: e'_w

Table 5. The actions defined by the dibits in the stutter control word of t*w*, $w \in \{8,16,32\}$

4.1 Cycle Length

The probabilistic nature of the stuttering means that we cannot accurately determine the cycle lengths for the key stream. The LFSR and NLF streams have cycle length $\left(2^{17w}-1\right)$. Soon after the LFSR outputs repeat, one of the SCWs will repeat, at which time the key stream will also begin repeating. The instructions for a random dibit result in a key stream being generated with probability $\frac{3}{4}$, so a random SCW outputs and average of $\frac{3w}{8}$ key stream words. Furthermore, a random dibit contains instructions for an average of $\frac{3}{2}$ LFSR cycles, so for each SCW there are an average of $\frac{3}{2} \cdot \frac{w}{2} + 1 = \frac{3}{4}w + 1$ LFSR cycles. Therefore, the number of key stream words generated before the key stream repeats is approximately

$$\frac{\frac{3}{8}w}{\frac{3}{4}w+1} \cdot \left(2^{17w}+1\right) = \frac{3w}{6w+8} \cdot \left(2^{17w}+1\right) \approx \tfrac{1}{2} \cdot \left(2^{17w}+1\right).$$

For example, the cycle length of t8 is around 2^{135}.

4.2 The Timing Distribution

The stuttering increases the security of the t-class ciphers at small cost by introducing uncertainty regarding the number of cycles of the LFSR between successive key stream words. However, the stuttering also introduces uncertainty in the amount of processing required to generate a given length of key stream.

The timing is directly proportional to the amount of processing required. The amount of processing required to cycle the LFSR has been found to be approximately equal to the amount of processing required for obtaining an NLF output. The amount of processing required for the stuttering is dominated by the processing required to cycle the LFSR and obtain the NLF output that becomes the SCW, from which the dibits are read. Therefore, amount of processing can be adequately described in terms of *processing units*, where a processing unit is equivalent to the processing required to either cycle the LFSR or obtain an NLF output. We make the following remarks regarding the number of processing units required to generate *n* key stream words (the analysis is found in the Appendix).

- The average number of processing units is approximately $\left(3+\frac{16}{w}\right)n$.
- For large *n*, the number of processing units exceeds the *probabilistic maximum* $\left(3+\frac{16}{w}\right)n+\left(2.65+\frac{6.20}{w}\right)\sqrt{n}$ with probability less than 10^{-4}.

Example 6. Table 6 lists the average value and probabilistic maximum for the number of process units of t16 required for generating *n* key stream words, for some large values of *n*. ■

Number of Processing Units	n : Number of Key Stream Words			
	256	512	1024	2500
Average	853.3	1706.7	3413.3	8333.3
Probabilistic Maximum	902.0	1775.5	3510.6	8485.3
% Diff Max. /Average	5.7	4.0	2.8	1.8

Table 6. The average value and probabilistic maximum for the number of process units of t16 required for generating *n* key stream words, for some large values of *n*. The final row lists the difference between the probabilistic maximum and the average, given as a percentage of the average.

4.3 Assume-Guess-and-Determine Attacks

GD attacks and EGD attacks rely on the attacker NLF words that occur in certain positions relative to each other. Hereafter we shall combine both attacks under the heading of GD attacks. A result of the stuttering is that the words from the key stream correspond to certain positions relative to each other only if the dibits in the stutter control word(s) assume certain values.

Example 7 The GD attack of Bleichenbacher, Patel and Meier [4] (described in Example 1) requires the attacker to know the NLF words v_{n+i}, $i \in \{-1,0,1,2,4,9,11,12,14\}$, for some $n \geq 0$. For the attack on SOBER-II (which also applies to t8) the authors of [4] chose to make the following assumptions:

- $v_{n-1} = 01011111_2$ is an SCW, which implies that the words v_{n+i}, $i \in \{0,1,2,4\}$ are put into the key stream and v_{n+6} is the next SCW.
- $v_{n+6} = 11110101_2$, which implies that the words v_{n+i}, $i \in \{7,9,11,12\}$ are put into the key stream and v_{n+13} is the next SCW.
- The least significant dibit of v_{n+13} is 01.

If the attacker has assumed the values of the SCWs correctly, then the attacker has all the NLF stream words required for performing the GD attack. Note that the assumptions regarding the SCWs give the attacker extra words in the NLF stream. ■

To perform a GD attack, the attacker must assume that the stutter control word(s) are such that the key stream outputs the maximum number of required NLF stream words. If the stutter control word(s) have the correct value, then the first key stream word outputted after first stutter control word is said to be *suitable*. After assuming that some key stream word z_j is suitable, the attacker performs the GD attack using corresponding values from the observed key stream. If the GD attack is able to find a register state that produces the correct stream, then the attack is successful. Otherwise, z_j must have been unsuitable, so the attacker then assumes that z_{j+1} is suitable and attempts a GD attack using the corresponding key stream words. The attacker continues until a correct register state is found. Such an attack is called an *assume-guess-and-determine* (*AGD*) attack.

Assume-Guess-and-Determine Attack

1. Set $j = 0$.
2. **Assume** that z_j is suitable, and determine the NLF stream words that would correspond to the following key stream words.
3. If a **guess-and-determine** or **extended-guess-and-determine** attack (based on these NLF stream words) determines a correct value for the state that reproduces the key stream, then the attack is complete. Otherwise, increment *j* and return to Stage 2, as z_j is not suitable.

The data complexity of an AGD attack is the product of the complexity of the GD attack, with the number of key stream words that are assumed to be suitable before a suitable key steam word is found. The latter factor is inversely proportional to the probability of a random key stream word being suitable. Let z_j be a random key stream word at some point in the key stream. As discussed above, an SCW outputs an average of $\frac{3}{8}w$ key stream words per SCW, so the probability that z_j is the first key stream word following an SCW is $\frac{8}{3w}$. Assuming that z_j is the first key stream word following an SCW, the probability that z_j is suitable is 2^{-2x} where *x* is the number of SCW dibits that have been specified in the AGD attack. Thus, the probability that z_j is suitable is $\frac{8}{3w} \cdot 2^{-2x}$.

Example 8. Consider the AGD attack on t8 given in Example 7. The probability that z_j is the first key stream word following an SCW is $\frac{8}{3 \cdot 8} = \frac{1}{3}$. The attack specifies 9 SCW dibits, so the probability that z_j is suitable is 2^{-18}, assuming that z_j is the first key stream word following an SCW. Thus, the probability that z_j is suitable is $\frac{1}{3} \cdot 2^{-18}$. The data complexity to be inversely proportional to this value: around

$3 \cdot 2^{18}$ key stream words or 786 kilobytes. For each key stream word z_j, a GD attack with complexity $2^{10w} = 2^{80}$ is performed for each key stream word, so the total complexity of the AGD attack on t8 is approximately $3 \cdot 2^{18} \cdot 2^{80} > 2^{99}$. ■

Similar AGD attacks can be defined for t16 and t32.

5 Keying the Stream Cipher

5.1 Byte Ordering Considerations

t16 and t32 utilise native processor operations on integer data items, but are expected to accept keys which are simply strings of bytes, and to produce a stream of bytes as output for encryption purposes. This means that a translation between native byte ordering and external byte ordering is necessary to ensure compatibility between implementations running on different processors. Since all Internet standards are defined using “big-endian” byte ordering, in which the most significant byte of a multi-byte quantity appears first in memory, this is what is chosen for t16 and t32. On “little-endian” machines, the words of the output stream must be byte reversed before being XORed into the buffer, respectively.

Note that it is simple to define ciphers that are exactly equivalent to t16 and t32 except that they are “little-endian”. These ciphers would share all the security aspects of the originals, but would execute a bit more efficiently on such CPUs.

Byte ordering considerations are ignored in the rest of the paper, but should be kept in mind.

5.2 Key Loading

The t-class ciphers are designed (primarily) 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 with a *frame key*, and the stream cipher re-keyed for each frame with the secret session key and the frame key. The frame key is public and consists of a 4-octet unsigned integer. SOBER, SOBER-II and S16 were designed to support such a two tier keying structure; S32 was not. All t-class ciphers have been designed to support the two-tiered keying structure in addition to the standard mode of operation. The cipher is keyed and re-keyed using two operations:

- Include(X); adds the word X to r_{15} modulo 2^w.
- Diffuse(); cycles the register and XORs the output of the NLF with r_4.

The main function used to load the session key and frame key is the Loadkey(k[],keylen) operation, where “k[]” is an array containing the “keylen” bytes of the key with one byte stored in each entry of “k[]”. The Loadkey() operation uses the values in “k[]” to alter the current values in the register as follows:

Loadkey(k[],keylen)

1. Define $kwl = \lceil \text{keylen} \div (w/8) \rceil$: the length of the key in words. Partition "k[]" into *kwl* *w*-bit words and store in an array "kw[]".
   - In t8, $\text{kw}[i] = \text{k}[i]$, $0 \le i \le \text{keylen-1}$.
   - In t16, this partitioning is performed as follows:
     $\text{keylen} \equiv 0 \bmod 16$: kw[0] = (k[0], k[1]), kw[1] = (k[2], k[3]), … ,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[2i], \text{k}[2i+1])$.
     $\text{keylen} \equiv 1 \bmod 16$: kw[0] = (0, k[0]), kw[1] = (k[1], k[2]), … ,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[2i-1], \text{k}[2i])$.
   - In t32, this partitioning is performed as follows:
     $\text{keylen} \equiv 0 \bmod 32$: kw[0] = (k[0], k[1], k[2], k[3]), …,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[4i], \text{k}[4i+1], \text{k}[4i+2], \text{k}[4i+3])$.
     $\text{keylen} \equiv 1 \bmod 32$: kw[0] = (0, 0, 0, k[0]), … ,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[4i-3], \text{k}[4i-2], \text{k}[4i-1], \text{k}[4i])$.
     $\text{keylen} \equiv 2 \bmod 32$: kw[0] = (0, 0, k[0], k[1]), …,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[4i-2], \text{k}[4i-1], \text{k}[4i], \text{k}[4i+1])$.
     $\text{keylen} \equiv 3 \bmod 32$: kw[0] = (0, k[0], k[1], k[2]), …,
     $1 \le i \le kwl - 1$, $\text{kw}[i] = (\text{k}[4i-1], \text{k}[4i], \text{k}[4i+1], \text{k}[4i+2])$.
2. For each i, $0 \le i \le kwl - 1$, Include(kw[i]) and apply Diffuse().[4]
3. Include(keylen).
4. Apply Diffuse() 17 more times. ■

The 17 applications of Diffuse() are designed to ensure that every bit of input affects every bits of the resulting register state in a nonlinear fashion, as discussed below. The key length is included to ensure that there are no equivalent session keys.

A t-class cipher is initially keyed using a secret, *t*-byte session key $K[0], \ldots, K[t-1]$ as follows:

1. The 17 words of state information are initialised to the first 17 Fibonacci numbers. There is no particular significance to these numbers being used, except for the ease of generating them. The value of *Konst* is set to the all zero word.
2. The cipher applies Loadkey(*K*[],*t*) which includes the session key bytes and session key length into the register, and diffuses the information throughout the register.
3. The LFSR is cycled and *Konst* is set to the value of the NLF output.

If the cipher is going to be re-keyed or the variable S-box constructed, then the 17 word state of the register, $r_0, \ldots, r_{16}$, (which we call the *initial key state*) can be saved at this point for later use, and the session key discarded. However, for shorter session keys, the session key could be saved and this procedure repeated as necessary, trading additional computation time for some extra memory.

[4] Note that each Include() in Step 2 is followed by a Diffuse().

If the cipher does not use the two-tiered keying structure, then the cipher produces a key stream with the register starting in the initial key state. That is, the initial key state is used as the initial state. However, if the cipher uses a frame key, the cipher first resets the register state to the saved initial key state, and then loads the 4-byte frame key $frame[0],\ldots,frame[3]$ using Loadkey(*frame*[],4). The state of the register following the re-keying is taken as the initial state, and the cipher produces a key stream with the register starting in this state.

5.3 Analysis of the t-class Key Loading

In [4], Bleichenbacher, Patel and Meier discuss weaknesses in frame keying in SOBER-II (assuming correctly that analogous problems appeared in S16). In this section we address some of the observations and demonstrate how changes made to arrive at the t-class eliminate these (and other) weaknesses.

Section 5 of [4] is devoted to "Analysis of the Frames". It demonstrates that there is correlation between related frames generated from the same initial key material. However we note that no actual attack based on this correlation is proposed; while *some* of the register elements display correlation, other elements do not, and the fact that the nonlinear function combines many elements, and that the linear feedback function continues mixing the correlated and uncorrelated elements suggests that such an attack might be difficult. Nevertheless, we agree with the conclusion that the correlation is undesirable.

A large contributor of the correlation was in fact due to a bug in our “sliding window” implementation of SOBER-II. The sliding window implementation effectively contains two full registers. In such an implementation it is essential to XOR the NLF output into TWO positions in the sliding window during the Diffuse() operation. The original implementation only XORed the NLF output into one position, which resulted in incorrect diffusion.

However, aside from this bug, there has proven to be insufficient Diffuse() operations after the session key or frame key had been included in the register in SOBER-II and S16. This resulted in a correlation between the session key or frame key and the initial state, as shown in the following example.

Example 9. For a set of 1000 random session keys, the initial states that result from loading $frame = 0\text{x}000000ab$ and $frame' = frame + 256$ were compared for XOR and modular addition difference in SOBER-II. For each word in the initial state a c^2 test was used to give a measure of how close the distribution of XOR (modular addition) differences are to a uniform distribution. If the distribution is uniform, then the expected value of the c^2 value is 255 (for more details see [4]). Table 7 lists our results, after the implementation bug had been corrected. An asterisk “*” indicates those values that demonstrate with 99% probability that the differences is not uniform. Notice that several register words have large c^2 values.

The same analysis was then applied to t8 with the new key loading (the analysis of t8 used 3000 random session keys); the results are also given in Table 7. These results indicate that the new key loading eliminated the correlation that exists in the key loading of SOBER-II and S16. ■

The t-class uses an updated key loading which is stronger than that used in SOBER-II and S16. The choice for the positions in the register to which information is included (r_{15}) and the NLF output fed back (r_4) optimises the key loading operations in terms of the speed at which key information diffuses throughout the register. In SOBER-II, the nonlinear output was folded back into the register at position 8. The reasoning was that this gave most rapid diffusion in the sense that every word of the register depended on every word of the key material. However that measure was only at the "first layer" – the dependence was (as can be seen) linear in nature in some cases, and differential analysis exposes the above weakness. In particular, because of the coincidences that the nonlinear tap at position 6 goes into the "more linear" part of the nonlinear function, and the linear tap is at position 4, there is a distinct odd/even behavior on the manner in which differences are distributed, as is exploited in [4].

Word of Register	χ^2 XOR difference		χ^2 arithmetic difference	
	SOBER-II	t8	SOBER-II	t8
0	859*	254	753*	254
1	339*	254	333*	254
2	258	255	259	255
3	254	255	255	254
4	255	255	255	254
5	256	255	255	255
6	255	254	256	254
7	255	254	255	254
8	254	255	253	254
9	21659*	255	4598*	255
10	1600*	253	481*	253
11	2815*	255	918*	255
12	282	254	262	254
13	344*	254	280	254
14	254	254	254	254
15	254	255	254	255
16	255	255	255	255

Table 7. The c^2 values for SOBER-II and t8.

If instead the nonlinear feedback is added at position 4 in the register, this changes dramatically. The next register cycle includes nonlinear material in the top of the register, which then goes through the less linear half of the nonlinear function, and so on. This eliminates both the odd/even behavior mentioned above, and better utilizes the "nonlinear half" of the nonlinear function. However, the 11 Diffuse() operations applied in SOBER-II do not provide sufficient diffusion of information, and there still remains a word r_5 which has a high c^2 value. The reason for this is that the word has not been modified other than by direct addition of key material and has had little influence from the NLF. Thus, there is a need to increase the number of Diffuse() layers following inclusion of the key into the register. This number has been increased to the minimum number of Diffuse() operations required to ensure that:

- Every bit of the initial state is a non-linear function of every bit of the key and frame number.
- For any session key, the set of initial states corresponding to the frame keys cannot be restricted to a linear subspace of $(\mathbf{Z}_2^w)^{17}$. If the initial states could be restricted to a linear subspace, then the linearity of the LFSR would ensure that every state of the register is always in some linear subspace, and this may yield an attack.
- No word of the initial state is algebraically related to any subset of other words.

Our analysis of the information propagation indicates that applying 17 Diffuse() operations will ensure that these properties hold. The analysis in Example 9 demonstrates just two types of information which no longer propagate through key loading into the initial state of the register.

6 Memory Requirements and Performance

The memory requirements of t-class ciphers are modest. Table 8 lists the memory requirements and performance data collated to date. It may be possible to improve on these figures.

Cipher	RAM (bytes)		ROM	Encryption Speed (Mbps)	
	Minimum	Typical	(bytes)	Unix 200	Pentium 233
t8	20	54	768	~25	~24
t16	38	105	2 560	~44	~25
t32	74	210	21 504	~76	~80

Table 8. Memory requirements and encryption speed for the t-class ciphers. The encryption speed was measured on a 200MHz Sun Ultra Sparc and a 233MHz Pentium using unoptimised code for the typical implementation. The speed was measure in megabits per second, by generating 10 key streams, each one megabyte in length.

The "minimum" RAM requirements account for an array holding the state of the register (17 words), a pointer to elements of the register (byte), a stutter control word (word), and a stutter control counter (byte) that determines when to draw another stutter control word from the NLF stream. The "typical" RAM requirements account for a "sliding window" implementation of the LFSR[5] (34 words), a pointer to

[5] See [17] for details.

elements of the register (byte), a stutter control word (word), a stutter control counter (byte) and an array for saving the initial key state (17 words). The ROM requirements allocate memory for the field multiplication tables (used in the LFSR) and the non-linear S-boxes (used in the NLF); all these arrays are fixed. The encryption speeds were measured on a 200MHz Sun Ultra Sparc and a 233MHz Pentium using the typical (sliding window) implementation. We note that the code used was not unoptimised, so it will be possible to achieve higher encryption speeds.

7 Security Analysis

We have already discussed many aspects of the security offered by the various components of the t-class ciphers. This section discusses how these components combine to provide the necessary security.

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 the t-class ciphers have 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 t-class ciphers to known-plaintext attacks.

Using $GF(2^w)$ instead of $GF(2)$ in the shift register has very little effect on the properties of the register itself. Herlestam [11] 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. The different bit positions in the words are merely offsets in the output sequence of that LFSR. Therefore, recovering any linearly independent set of enough 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 these ciphers rests entirely upon the interaction of the nonlinear function and the stuttering.

The carry bits in the word addition, and the nonlinear *S*-boxes account for most of the nonlinear behavior 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, the non-linear *S*-box brings in a bit with a high degree of non-linearity to disguise the linearity of the least significant bit of the remaining sum. The *S*-boxes also add significant non-linearity into other bits. XORing the *Konst* removes a significant amount of commutation in the NLF that might have been exploited. The only operations that commute are the additions of r_1 and r_6 with the output of f_w.

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, non-linear state (one of the major benefits of a software implementation).

Lastly, statistical tests have been used to bolster confidence that nothing entirely stupid has been done. Outputs have been examined using CRYPT-X'98 [6], with no anomalous results reported.

7.1 Regarding keys

For the t-class ciphers, there is a very small probability that a particular combination of key and frame number will yield an initial register state which is entirely 0, and the algorithm will cycle forever producing the same output. This could be fixed by checking for that state and replacing it with one that cannot appear as the result of a normal keying operation.

7.2 Attacks against Components of the t-class

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 the t-class ciphers, 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 are based on exhaustive key searches or exhaustive state searches, or using some other technique to predict past or future output from the cipher. In the case of the t-class ciphers, these are thought to be the same goal. The three major components (the LFSR, the NLF and the stuttering) all contribute to the strength of the cipher in different, complementary, ways.

The LFSR itself is not cryptographically strong; its contribution is primarily to provide the LFSR stream with good statistical properties. As mentioned above, the individual bit positions of the LFSR act as if they are bits of a longer binary LFSR.

The NLF by itself is reasonably strong. The combination of the LFSR and NLF provides sufficient resistance to GD and EGD attacks, as the best such attack has a complexity of 2^{10w}, significantly large than the complexity of 2^{8w} for an exhaustive key search. However, fast correlation attacks are conjectured to recover the initial state from a few thousand known octets of output from the nonlinear function. We are aware of an attack which exploited the near-linearity of the least significant bits in the original SOBER design [17] to recover the initial state from a few more than 136 consecutive octets. Both of these attacks are completely defeated by the stuttering, as attempting to guess the positions in the output stream very quickly exceeds the complexity of an exhaustive key search.

The stuttering by itself is not very strong. If it was applied to purely linear output, then 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 that proceeds by guessing enough stutter control words to form enough linear equations to be solved requires guessing only 17 dibit (34 bits). Both of these attacks are in turn frustrated by the non-linearity of the inputs, and are thought to be infeasible. The fact that the stutter control words are derived from the non-linear function contributes to the strength of the stuttering.

Finally, because there is a single, monolithic shift register in the t-class ciphers, general divide-and-conquer attacks do not seem to be applicable.

8 Conclusion

The t-class ciphers are conservatively designed stream ciphers with a very small software footprint, designed primarily for embedded applications in wireless telephony. Software implementations of LFSRs over GF(2^w) can be extremely efficient, allowing well-tried design principles to be brought to bear in software ciphers. The t-class improves on the original SOBER family by offering a stronger non-linear filter and a more secure key loading process.

Appendix: The Distribution of Processing Units

Dibit	(0,0)	(0,1)	(1,0)	(1,1)
Processing units	1	3	3	2

Table 9. The numbers of processing units that correspond to the possible dibits.

We examine the number of process units in two parts. The first part considers the process units used when cycling the LFSR for the (0,0) dibits, and the process units used when obtaining the SCW. The second part considers the number of process units used in those dibits when a key stream word is generated.

Process Units due to the (0,0) Dibit and Obtaining the SCW

For a fixed *n*, let *z* denote the number of (0,0) dibits encountered before *n* key stream words are generated. A total of $(z+n)$ dibits are read during stuttering, so the number of SCWs is $\lceil (z+n)/(w/2) \rceil = \lceil 2(z+n)/w \rceil$. Each SCW requires two processing units: one unit for cycling the LFSR and one unit for obtaining the NLF output. Also, each (0,0) dibit requires one process unit to cycle the LFSR. Thus, the total number of process units due to the number of (0,0) dibits is $z + 2 \cdot \lceil 2(z+n)/w \rceil$. The distribution of z follows a *negative binomial distribution*[6] with probabilities $p = \frac{3}{4}$ and $q = 1 - p = \frac{1}{4}$; that is, for $x \geq 0$,

[6] For more details, see any good textbook on statistics.

$$\Pr(\, z = x\,) = \binom{n+x-1}{x} \cdot p^n \cdot q^x = \binom{n+x-1}{x} \cdot \frac{3^n}{4^{n+x}}\,.$$

For small values of *n* this distribution can be determined explicitly to find suitable probabilistic bounds on *z*, however we have not evaluated such bounds here. For large values of *n* this distribution can be approximated by a normal distribution with mean $m_z = \frac{1}{3}n$, variance $s_z^2 = \frac{\frac{1}{4}}{\left(\frac{3}{4}\right)^2} n = \frac{4}{9}n$ and standard deviation $s_z = \frac{2}{3}\sqrt{n}$.

Therefore, the average number of process units due to the (0,0) dibits and obtaining the SCW is

$$m_z + 2 \cdot \left\lceil 2(m_z + n)/w \right\rceil = \tfrac{1}{3}n + 2 \cdot \left\lceil \tfrac{8}{3} n / w \right\rceil \square \tfrac{1}{3}n + \tfrac{16}{3w} n = \tfrac{1}{3}(1 + \tfrac{16}{w})n\,.$$

Furthermore, a property of the normal distribution is that $\Pr(\, z < m_z + 2.326 s_z\,) = 10^{-2}$, so the value of *z* can be upper bounded (with probability 10^{-2}) by $m_z + 2.326 s_z = \frac{1}{3}n + 1.55\sqrt{n}$. Consequently, the number of process units due to the number of (0,0) dibits and obtaining the SCW is greater than

$$\begin{aligned} &\tfrac{1}{3}n + 1.55\sqrt{n} + 2 \cdot \left\lceil 2\left(\tfrac{4}{3}n + 1.55\sqrt{n}\right)/w \right\rceil \\ &\quad \square\ \tfrac{1}{3}n + 1.55\sqrt{n} + 2 \cdot \left(\tfrac{8}{3w}n + \tfrac{3.10}{w}\sqrt{n}\right) = \tfrac{1}{3}\left(1 + \tfrac{16}{w}\right)n + \left(1.55 + \tfrac{6.20}{w}\right)\sqrt{n}, \end{aligned}$$

with a probability of only 10^{-2} .

Process Units Resulting When a Key Stream Word is Generated

For a fixed *n*, let *T* denote the number of (0,1) or (1,0) dibits encountered before *n* key stream words are generated. The number of process units resulting from these dibits is $3T + 2(n - T) = 2n + T$. The probability that a random non-zero dibit is (0,1) or (1,0) is $\frac{2}{3}$, and the probability that a random non-zero dibit is (1,1) is $\frac{1}{3}$. Thus, the distribution of *T* follows a *binomial distribution*[7] with probabilities $p = \frac{2}{3}$ and $q = 1 - p = \frac{1}{3}$. That is, for $x \ge 0$,

$$\Pr(\, T = x\,) = \binom{n}{x} \cdot p^x \cdot q^{n-x} = \binom{n}{x} \cdot \frac{2^x}{3^n}\,.$$

For small values of *n* this distribution can be determined explicitly to find a suitable bound on *T*; we have not evaluated such bounds here. For large values of *n* this distribution can be approximated by a normal distribution with mean $m_T = np = \frac{2}{3}n$,

[7] For more details see any good textbook on statistics.

variance $s_2^2 = npq = \frac{2}{9}n$ and standard deviation $s_z = \frac{\sqrt{2}}{3}\sqrt{n}$. Therefore, the average number of process units due to the dibits where a key stream word is generated is

$$2n + m_T + = 2n + \tfrac{2}{3}n = \tfrac{8}{3}n .$$

Furthermore, $\Pr(T < m_T + 2.326 s_T) = 10^{-2}$, so the values of T can be upper bounded by $m_T + 2.326 s_T = \frac{2}{3}n + 1.10\sqrt{n}$. Consequently the total number of process units due to the dibits where a key stream word is generated is greater than $2n + \frac{2}{3}n + 1.10\sqrt{n} = \frac{8}{3}n + 1.10\sqrt{n}$ with a probability of only 10^{-2}.

Combining the Distributions

Combining the results in the two subsections, the average number of process units is

$$\tfrac{1}{3}\left(1 + \tfrac{16}{w}\right)n + \tfrac{8}{3}n = \left(3 + \tfrac{16}{w}\right)n .$$

The *probabilistic maximum* on the number of process units is

$$\tfrac{1}{3}\left(1 + \tfrac{16}{w}\right)n + \left(1.55 + \tfrac{6.20}{w}\right)\sqrt{n} + \left(\tfrac{8}{3}n + 1.10\sqrt{n}\right) = \left(3 + \tfrac{16}{w}\right)n + \left(2.65 + \tfrac{6.20}{w}\right)\sqrt{n} .$$

This maximum is exceeded with probability $10^{-2} \times 10^{-2} = 10^{-4}$.

References

1. S. Blackburn, S. Murphy, F. Piper and P. Wild, "A SOBERing Remark". Unpublished report. Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U. K., 1998.

2. S. Blackburn, S. Murphy, F. Piper and P. Wild, "Brief Comments on SOBER-II". Unpublished report. Information Security Group, Royal Holloway University of London, Egham, Surrey TW20 0EX, U. K., 1998.

3. D. Bleichenbacher and S. Patel, "SOBER cryptanalysis", Pre-proceedings of *Fast Software Encryption '99*, 1999, pp. 303-314.

4. D. Bleichenbacher, S. Patel and W. Meier, "Analysis of the SOBER stream cipher". TIA contribution TR45.AHAG/99.08.30.12.

5. E. Dawson, B. Millan and L. Simpson, "Security Analysis of SOBER". Unpublished report by the Information Systems Research Centre, Queensland University of Technology, 1998.

6. E. Dawson, A. Clark, H. Gustafson and L. May, "CRYPT-X'98, (Java Version) User Manual", *Queensland University of Technology,* 1999.

7. E. Dawson, W. Millan, L. Burnett, G. Carter, "On the Design of 8*32 S-boxes". Unpublished report, by the Information Systems Research Centre, Queensland University of Technology, 1999.

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

9. C. Hall and B. Schneier, "An Analysis of SOBER", 1999.

10. P. Hawkes, "An Attack on SOBER-II". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1999. See http://www.home.aone.net.au/qualcomm.

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

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

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

14. 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.

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

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

17. G. Rose, "A Stream Cipher based on Linear Feedback over GF(2^8)", in C. Boyd, Editor, *Proc. Australian Conference on Information Security and Privacy*, Springer-Verlag 1998.

18. G. Rose, "SOBER: A Stream Cipher based on Linear Feedback over GF(2^8)". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1998. See http://www.home.aone.net.au/qualcomm.

19. G. Rose, "S32: A Fast Stream Cipher based on Linear Feedback over GF(2^{32})". Unpublished report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne NSW 2047 Australia, 1998. See http://www.home.aone.net.au/qualcomm.