Primitive Specification for SOBER-128

Philip Hawkes and Gregory G. Rose
{phawkes,ggr}@qualcomm.com
Qualcomm Australia
Level 3, 230 Victoria Rd
Gladesville NSW 2111
Australia
Tel: +61-2-9817-4188,
Fax: +61-2-9817-5199

Table of Contents

1 Justification

There is a need for a primitive stream cipher construction that is fast (faster than a block cipher in counter mode), easy to use correctly, well understood, freely available, and secure. SOBER-128 has been designed to meet these requirements, by being based entirely on a well-studied primitive in a manner that preserves the existing analyses.

Additionally, SOBER-128 introduces functionality to enable simultaneous calculation of a Message Authentication Code, and allows integrity checking of partially encrypted messages.

2 Description of SOBER-128

2.1 Introduction

SOBER-128 is a synchronous stream cipher designed for a secret key that is up to 128 bits in length. The cipher outputs the key stream in 32-bit blocks. SOBER-128 is a software-oriented cipher based on 32-bit operations (such as 32-bit XOR and addition modulo 2^{32}), and references to small fixed arrays. Consequently, SOBER-128 is at home in many computing environments, from smart cards to large computers. Source code for SOBER-128 is freely available and use of this source code, or independent implementations, is allowed free for any purpose.

2.1.1 History: The SOBER Family of Stream Ciphers

SOBER-128 was developed from SOBER [32], proposed by Rose in 1998. The algorithm for SOBER is based on 8-bit operations, versus the 32-bit operations used in SOBER-128. SOBER was superseded by SOBER-II [33] when various weaknesses were found in the original design. S16 was proposed as 16-bit extension of SOBER-II: S16 copies the structure of SOBER-II and uses 16-bit operations. However, there were opportunities for strengthening SOBER-II and S16 that could not be ignored. Consequently, replacements for SOBER-II, S16 and a 32-bit version were created. These replacements were called the t-class of SOBER ciphers [22]. The t-class contains three ciphers based on 8-bit, 16-bit and 32-bit operations. The ciphers SOBER-t16 and SOBER-t32 were submitted to the NESSIE program [31]; SOBER-t16 as a stream cipher for 128-bit key strength and SOBER-t32 as a stream cipher with 256-bit key strength. SOBER-t16 and SOBER-t32 proved to be among the strongest stream cipher submissions to NESSIE. However, both ciphers were found to fall short of the stringent NESSIE requirements.

SOBER-128 is an improved version of SOBER-t32. The modifications directly address the concerns arising in the analyses of the t-class ciphers. The 128-bit key strength proposed for SOBER-128 is reduced from the 256-bit key strength proposed for SOBER-t32 to ensure that SOBER-128 provides far in excess of the stated security level.

2.1.2 Transition between SOBER-t32 and SOBER-128

In designing SOBER-128 our goal was to ensure that as much as possible of the analyses of SOBER-t32 remained applicable to it. The changes made are easily identifiable, and fall into three categories:

1. Implementation changes that increase efficiency but do not affect cryptographic security:

    - Restructuring the S-Box so that the same transformation is done with less computer instructions

    - Changing the structure of the LFSR to increase the efficiency of updating it and reduce the size of the multiplication table; the new implementation is isomorphic to the old

    - Where possible, data is processed in 17-word units allowing significant compiler optimisation through instruction parallelism, loop unrolling and constant offsets

    - SOBER-128 only supports keys and IVs (Initialisation Vectors) that are multiples of 4 bytes; support for odd-length keys and IVs significantly complicated the code and has been deleted. Keys and IVs can always be padded by the user

    - SOBER-128 adopts "little-endian" byte ordering since Intel platforms seem to be of interest to more people these days

2. Changes that alter the cryptanalysis of the stream cipher:

    - "Stuttering", one of the fundamentals of the SOBER family, has been deleted. This mechanism added some security, but at high cost; it was computationally very expensive and exposed the cipher to side-channel attacks

    - The nonlinear transformation has been strengthened, in particular by adding a fixed rotation and a second s-box transformation

    - "Konst" has a non-zero value during key loading, and is derived in a manner that ensures that the most significant byte is non-zero after key loading

3. Addition of message authentication functionality.

2.1.3 Usage and threat model

SOBER-128 may be used to generate a single encryption keystream of arbitrary length. In this mode it would be possible to use SOBER-128 as a replacement for the commonly deployed RC4 cipher in, for example, SSL/TLS. In this mode, no IV is necessary.

In practice though, much communication is done in messages, where it is desirable to provide message integrity for the whole message, and privacy (encryption) for all or part of the message. The same

secret key should be usable for the entire (multi-message) communication, using an Initialisation Vector or Nonce to distinguish individual messages. SOBER-128 supports this model of use. Section 9 below describes the recommended interface.

SOBER-128 is intended to provide security under the condition that no IV is ever reused, that no more than 2^{80} words of data are processed with one key, and that no data failing authentication checks will be further processed (in particular, that the result of decrypting a message that fails authentication will not be revealed). There is no requirement that IVs be random, which makes guaranteeing uniqueness much easier. The sender and recipient of secure data must agree on which parts of the data are encrypted, and which parts are authenticated, for correct operation, as the encryption operation is plaintext-aware when message authentication is being performed. Section 9 describes the recommended interface calls.

2.2 Outline of this Document

Section 3 contains a description of SOBER-128. An analysis of the security characteristics of SOBER-128 is found in Section 4. Section 5 outlines the strengths and advantages of SOBER-128 while Section 6 states the design rationale. Computational efficiency is discussed in Section 7. Appendices provide a recommended C-language interface and the entries of the multiplication table and the substitution box used in the non-linear function.

3 Description

3.1 Overview

SOBER-128 is constructed from a *linear feedback shift register* (LFSR), a *non-linear filter* (NLF) and a nonlinear *plaintext feedback function* (PFF). Figure 1 contains a graphical representation of the keystream generator structure. Figure 2 contains a graphical representation of the Message Authentication Code accumulation structure. The primitive is based on 32-bit operations and 32-bit blocks: each 32-bit block is called a *word*. The LFSR produces a stream $\{s[t]\}$ of words using operations over the Galois field of order 2^{32}: this field is denoted by $GF(2^{32})$. The vector $\sigma_t = (s[t], \ldots, s[t+16])$ is known as the *state* of the LFSR at time t, and the state $\sigma_0 = (s[0], \ldots, s[16])$ is called the *initial state*. The *key state* and a 32-bit, key-dependent word called *Konst* are initialised from the secret key by the *key loading*. The key state can be used directly or can be further perturbed by the Initialisation Vector loading process to form the initial state.

In the absence of message authentication, successive states σ_t from the LFSR are then fed through the NLF to produce 32-bit *keystream words* denoted v_t. These combine to form the *keystream* $\{v_t\}$. Each keystream word v_t is obtained as

$$v_t = NLF(\sigma_t) = F(s[t], s[t+1], s[t+6], s[t+13], s[t+16], Konst) .$$

The function F is described in Sect. 3.3.

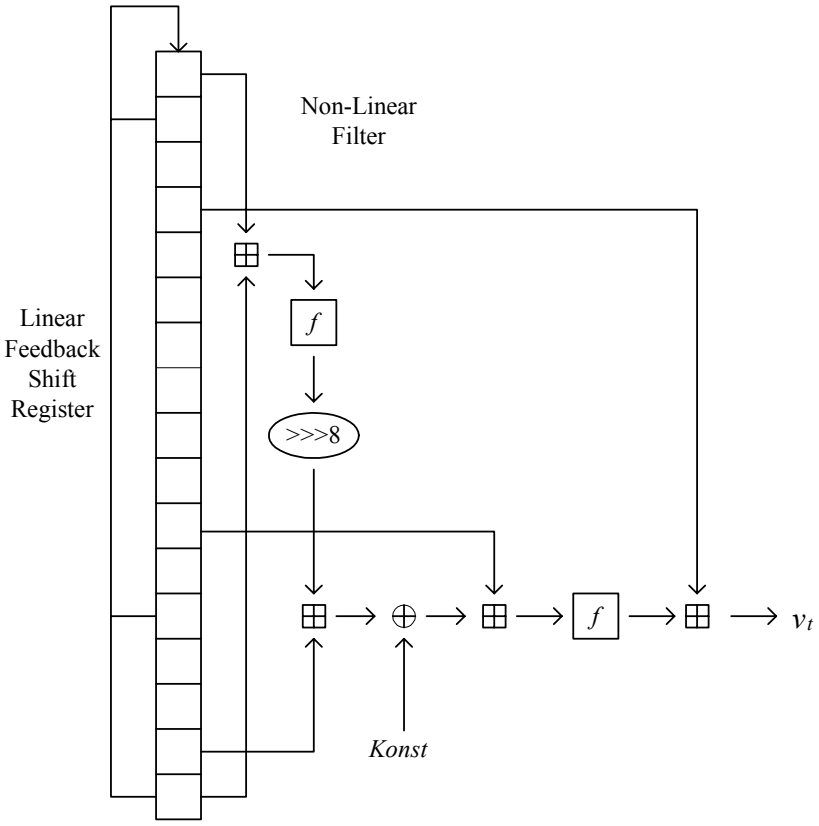

Figure 1. **The structure of SOBER-128 stream cipher *NLF***

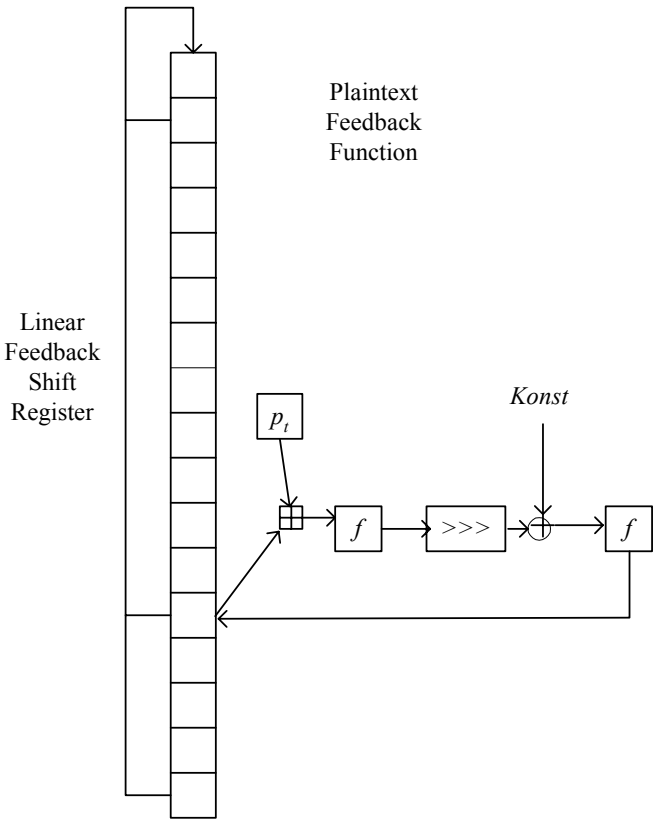

Figure 2. **The structure of SOBER-128 MAC accumulator *PFF***

When message authentication codes are being used, plaintext words p_t are incorporated into the LFSR in a nonlinear fashion. Strictly speaking, the LFSR can no longer be considered linear, but we will continue to refer to it as "the LFSR". One word $s[t+4]$ is replaced by

$$s[t+4] = PFF(s[t+4], p_t, Konst) .$$

3.1.1 Byte ordering

SOBER-128 is entirely based on 32-bit word operations internally, but the external interface is specified in terms of arrays of bytes. Conversion between 4-byte chunks and 32-bit words is done in "little-endian" fashion irrespective of the byte ordering of the underlying machine. This is a break with the tradition of previous members of the SOBER family.

3.2 The Linear Feedback Shift Register

Binary Linear Feedback Shift Registers can be extremely inefficient in software on general-purpose microprocessors. LFSRs can operate over any finite field, so an LFSR can be made more efficient in software by utilizing 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, in this case 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.

The standard representation of an element A in the field $GF(2^w)$ is a w-bit word with bits (a_{w-1}, a_{w-2},…,a_1,a_0), which represents the polynomial $a_{w-1}z^{w-1} + \ldots + a_1 z + a_0$. Elements can be added and multiplied: addition of elements in the field is equivalent to XOR. To multiply two elements of the field we multiply the corresponding polynomials modulo 2, and then reduce the resulting polynomial modulo a chosen irreducible polynomial of degree w.

It is also possible to represent $GF(2^w)$ using a subfield. For example, rather than representing elements of $GF(2^w)$ as degree-31 polynomials over $GF(2)$, SOBER-128 uses 8-bit bytes to represent elements of a subfield $GF(2^8)$, and 32-bit words to represent degree-3 polynomials over $GF(2^8)$. This is isomorphic to the standard representation, but not identical. The subfield $B = GF(2^8)$ of bytes is represented in SOBER-128 modulo the irreducible polynomial $z^8 + z^6 + z^3 + z^2 + 1$. Bytes represent degree-7 polynomials over $GF(2)$; the constant β_0 = 0x67 below represents the polynomial $z^6 + z^5 + z^2 + z + 1$ for example. The Galois finite field $W = B^4 = GF((2^8)^4)$ of words can now be represented using degree-3 polynomials where the coefficients are bytes (subfield elements of B). For example, the word 0xD02B4367 represents the polynomial $0xD0y^3 + 0x2By^2 + 0x43y + 0x67$. The field W can be represented modulo an irreducible polynomial $y^4 + \beta_3 y^3 + \beta_2 y^2 + \beta_1 y + \beta_0$. The specific coefficients β_i used in SOBER-128 are best given after describing the LFSR of SOBER-128.

An LFSR of order k over the field $GF(2^w)$ generates a stream of w-bit words $\{s[t]\}$ using a *register* of k memory elements `(R[0],R[1],...,R[n-1])`. The register stores the values of k successive LFSR words so after i *clocks* the register stores the values of $\sigma_i = (s[t], s[t+1], ..., s[t+n-1])$. At each clock, the LFSR computes the next word $s[t+k]$ in the sequence using a $GF(2^w)$ recurrence relation

$$s[t+k] = \alpha_0 s[t] + \alpha_1 s[t+1] + \ldots + \alpha_{k-1} s[t+n-1],$$

and updates the register (here new contains the value of $s[i+k]$):

```
R[0] = R[1]; R[1] = R[2]; ...; R[15] = R[16]; R[16] = new;
```

The register now contains $\sigma_{t+1} = (s[(t+1)], s[(t+1)+1], ... , s[(t+1)+n-1])$. The linear recurrence above is commonly represented by the *characteristic polynomial* $p(X) = X^n - \sum_{j=0..(n-1)} \alpha_j X^j$.

In the case of SOBER-128, the LFSR consists of n = 17 words of state information with w = 32-bit words. The LFSR was developed in three steps. First, the characteristic polynomial of the SOBER-128

LFSR was chosen to be of the form $p(X) = X^{17} + X^{15} + X^4 + \alpha$, over $GF(2^{32})$. The exponents {17, 15, 4, 0} were chosen because they provide good security; the use of these exponents dates back to the design of SOBER-t16 and SOBER-t32 [22]. Next, the coefficient α = 0x00000100 ≈ 0x00·y^3 + 0x00·y^2 + 0x01·y + 0x00 = y, was chosen because it allows an efficient software implementation: multiplication by α consists of retrieving a pre-computed constant from a table indexed by the most significant 8 bits, shifting the input word to the left by 8 bits, and then adding (XORing) the resulting words together. This is essentially the field multiplication used in SNOW[14] and is exactly that used in Turing [25].

The `Multab` for SOBER-128 is shown in Appendix B. In C code, the new word to be inserted in the LFSR is calculated:

```
new = R[15] ^ R[4] ^ (R[0] << 8) ^ Multab[(R[0]>>24) & 0xFF];
```

where `^` is the XOR operation; `<<` is the left shift operation; and `>>` is the right shift operation. Finally, the irreducible polynomial defining the representation of the Galois field W was chosen to be y^4 + 0xD0·y^3 + 0x2B·y^2 + 0x43·y + 0x67, for reasons explained in Section 6.

The LFSR is stepped once before generation of each keystream word and/or accumulation of each plaintext word for MAC calculation.

3.3 The S-Box Function *f*

Notation: The most significant 8 bits of 32-bit word a is denoted a_H.

The function f employs XOR, and an 8 × 32-bit substitution box (S-box) denoted SBox. For a 32-bit value a, the function is defined as $f(a)$ = SBox[a_H] ⊕ a.

The S-box is a combination of the Skipjack [17] S-box (called "F-table" in the definition of Skipjack) and an S-box tailor-designed by the Information Security Research Centre (ISRC) at the Queensland University of Technology [10]. The ISRC S-box was constructed as 24 mutually uncorrelated, balanced and highly non-linear single bit functions. Suppose that the S-box has the input a_H. The eight most significant bits (MSBs) of the output of the S-box, XORed with a_H, are equal to the output of the Skipjack S-box, given the input a_H. The 24 least significant bits (LSBs) of the output of the S-box are the output of the S-box constructed by the ISRC, given the input a_H. The entire S-box is given in Appendix A of this document. (Note: the SOBER-128 f yields the same output as the SOBER-t32 f, but does not require a masking operation. The SBox[] table differs only in the high byte of each word.)

Thus, the eight most significant bits of the output of f is the output of the Skipjack S-box, while the 24 LSBs are obtained by XORing the 24 bits of the output of the ISRC S-box with the 24 LSBs of the input (see Figure 2). The function f is defined this way to ensure that it is one-to-one and highly non-

linear, while using only a single, small S-box. The function *f* also serves to transfer the non-linearity from the high bits of its input to the low bits of its output.

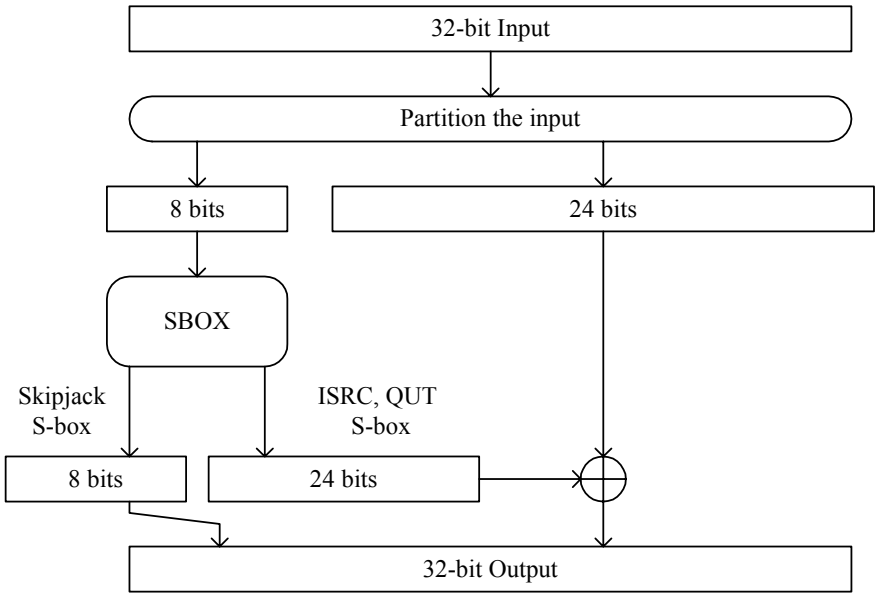

Figure 3. The structure of the function *f* used in SOBER-128

3.4 Keystream generation and The Non-Linear Filter

The NLF employs the operations of XOR, addition modulo 2^{32}, shifting, and the nonlinear S-box function above.

The NLF is defined by:

$$v_t = F(s[t], s[t+1], s[t+6], s[t+13], s[t+16], Konst)$$

$$= f((f(s[t] + s[t+16])>>>8 + s[t+1]) \oplus Konst) + s[t+6]) + s[t+13],$$

where the function $f(\cdot)$ using the S-box is defined above, ">>>" is a cyclic rotation and "+" denotes addition modulo 2^{32}. The generated keystream words are combined with plaintext words using XOR:

$c_t = p_t \oplus v_t$, or for decryption, $p_t = c_t \oplus v_t$

3.5 Message Authentication Codes

SOBER-128 calculates MACs in two phases, accumulation and finalisation.

3.5.1 MAC Accumulation

In the accumulation phase, input plaintext words (or the results of decryption) p_t are combined with the LFSR in a nonlinear fashion:

$$s[t+4] = \text{PFF}(s[t+4], p_t, Konst) = f((f(s[t+4] + p_t) >>> 8) \oplus Konst)$$

The Plaintext Feedback Function *PFF* employs the operations of XOR, addition modulo 2^{32}, shifting, and the nonlinear S-box function above.

MAC accumulation and generation of a keystream word depend on different register elements and can be calculated in either order. Whether encrypting or decrypting, though, it is always plaintext words that are accumulated.

3.5.2 MAC Finalisation

Generating an output MAC, once all accumulation and encryption for a given message has been completed, is done in the following manner. Reference is made to operations "Include" and "Diffuse", defined below.

1. Include(0x6996c53a) – (adds *INITKONST* to r_{15}; similar to keyloading with 4 byte key).
2. Apply Diffuse() 18 times – (one to process the above include; 17 for diffusion).
3. Generate the desired number of words of keystream, which constitute the MAC.

3.6 The Key and IV Loading

SOBER-128 is keyed and re-keyed by using operations that transform the values in the register under the influence of key material. Two principle operations are employed:

Include(*X*): this operation adds the word *X* to `r[15]` modulo 2^{32}.

Diffuse(): this operation clocks the register, obtains the output *v* of the NLF and replaces the value of `r[4]` with the value of (`r[4]` $\oplus$ *v*).

The main function used to load the key and IV 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 transform the current state of the register.

Algorithm for Loadkey(`k[], keylen`):

All keys must be a multiple of 4 bytes in length; `keylen` is the length of the key in bytes.

1. Convert `k[]` into `kwl` = `keylen`/4 words and store in an array `kw[]` of `kwl` "big-endian" words
2. For each *i*, $0 \le i \le$ (`kwl` –1), Include(`kw`[*i*]) and apply Diffuse().[1]
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 bit of the resulting register state in a nonlinear fashion, as discussed in Section 4.4.6. Including `keylen` ensures that keys and IVs of different lengths result in distinct initial states.

SOBER-128 is keyed using a secret, *t*-byte session key *K*[0], … , *K*[*t*-1] as follows:

1. The 17 words of state information are initialized to the first 17 Fibonacci numbers[2] by setting `R[0] = R[1] = 1`, and computing `R[`*i*`] = R[`*i*`-1] + R[`*i*`-2]`, for $2 \le i \le 16$. The value of *Konst* is set to the word 0x6996c53a (called INITKONST).

2. The cipher applies Loadkey(*K*[],*t*) which includes the key bytes and key length into the register, and diffuses the information throughout the register.

3. The LFSR is clocked and the NLF output is calculated and stored in a temporary variable. This process is iterated until the value calculated has a non-zero most significant byte. *Konst* is then set to the resulting value.

4. If the cipher is going to be used for multiple messages, then the 17 word state of the register, (`R[0],…,R[16]`), (which we call the *key state*) can be saved at this point for later use, and the key discarded. However, for shorter keys, the key could be saved and the keying procedure repeated as necessary, trading additional computation time for some extra memory.

5. If the cipher is not being used with IVs, then the cipher produces a key stream with the register starting in the key state. That is, the key state is used as the initial state. However, if the cipher uses IVs, then the cipher first resets the register state to the saved initial key state, and loads the *m*-byte IV *IV*[0], … , *IV*[*m*-1] using Loadkey(*IV*[],*m*). 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. Note that a zero-length IV is allowed, and is distinct from all other IVs and also distinct from the key state.

4 Security Analysis of SOBER-128

We are fortunate that almost all analyses of older versions of SOBER concentrated on "SOBER with the stuttering omitted", and then extended the attacks to SOBER with stuttering. This means that we are able to look at existing attacks and apply them to SOBER-128, below.

---

[1] Note that this means each Include() in Step 2 is followed by a Diffuse().
[2] There is no particular significance to these numbers being used, except for the ease of generating them.

4.1 Security Requirements

SOBER-128 is intended to provide 128-bit security. The base attack on SOBER-128 is an exhaustive key search, which has a computational complexity of 2^{128} (see Sect. 4.4.3)[3]. In all attacks, it is assumed that the attacker observes a certain amount of keystream produced by one or more secret keys, and the attacker is assumed to know the corresponding plaintext and IVs. SOBER-128 is considered to *resist* an attack if either the attack requires the owner of the secret key(s) to generate more than 2^{80} key stream words, or the computational complexity of the attack is equivalent to the attacker rekeying the cipher 2^{128} times and generating at least 5 words of output each time. We claim that SOBER-128 fulfils the following security requirements, when used subject to the condition that no key/IV pair is ever reused, and no more than 2^{80} words are processed with one key:

1. **Key/State Recovery Attacks**: SOBER-128 must resist attacks that either determine the secret key, or determine the values of *Konst* and the state at any specified time.

2. **Keystream Recovery Attacks**: SOBER-128 must resist attacks that accurately predict unknown values of the keystream without determining information about the LFSR state or the secret key.

3. **Distinguishing attacks**: SOBER-128 should resist attacks that distinguish a SOBER-128 keystream from random bit stream.

4. **Related-Key or related IV Attacks**: SOBER-128 should resist attacks of the above form that use keystream generated from one or more secret keys that are related in some manner known to the attacker.

5. **MAC forgery attacks**: SOBER-128 should resist attacks that allow a forger to create or modify a message that subsequently passes verification of the associated MAC.

SOBER-128 will be considered broken if an attacker can perform any of these attacks. Keystream recovery attacks seem unlikely, as the output sequence relies heavily on the state of the LFSR, so any likely keystream recovery attack will probably also allow the stronger key/state recovery attack. Most attacks concentrate on the first option of determining the values of *Konst* and the state. Related-key attacks are of less concern, since most security systems ensure that attackers cannot predict relationships between secret keys. However, it is still preferable that SOBER-128 resists such attacks.

MAC forgery attacks seem to require that the adversary be able to predict the LFSR state at the time the MAC is generated, and so are also thought to be subordinate to the key/state recovery attacks.

A comment on distinguishing attacks. There is currently some debate regarding the complexity of distinguishing attacks on stream ciphers. Some members of the cryptologic community claim that a stream ciphers cannot be secure when the data complexity and computational complexity for a successful distinguishing attack is less than the key space. For example, these people would say that

[3] Unless, of course, a shorter secret key is used. We assume use of a 128-bit secret key in this section.

SOBER-128 is not secure if there is a distinguishing attack requiring 2^{80} key stream words and 2^{100} computations. Other members of the cryptologic community claim that a stream ciphers can still be secure when the data complexity and computational complexity for a successful distinguishing attack is less than the limits imposed on other types of attacks. These parties would say that SOBER-128 is still secure even if there is a distinguishing attack requiring 2^{64} key stream words and 2^{80} computations. Although the designers hold the second view (that stream ciphers can still be secure even when the complexities of distinguishing attacks fall below the bounds of the keyspace), the intention of the design is to ensure that there are no distinguishing attacks on SOBER-128 requiring less than 2^{80} key stream words and less than 2^{128} computations. Since we do not allow SOBER-128 to be used to process more than 2^{80} words, such an attack would not break SOBER-128 used correctly. By comparison, AES-128 in counter mode has a distinguishing attack requiring 2^{66} keystream words, and AES-128 in CBC-MAC mode has a forgery attack of approximately the same complexity and data requirements.

4.2 Security Claims

We believe that any attack on SOBER-128 has a complexity exceeding that of an exhaustive key search. We do not claim any mathematical proof of security. Our analysis of SOBER-128 can be summarized thus:

- Guess-and-determine (GD) attacks appear to have a computational complexity in excess of 2^{250} (see [24,1]), although research in this area is ongoing.
- Algebraic attacks [8] appear to be infeasible.
- Correlation-based attacks [6] appear to be resisted by the LFSR and NLF.
- Timing attacks and power attacks can be mitigated in standard ways; there is no conditional execution after initial keying.
- We are unaware of any ways in which the key loading can be exploited.
- We are unaware of any weak keys or weak-key classes. It is theoretically possible for the initial state to be entirely zero, but we believe that this cannot occur when using 128-bit secret keys and 128-bit IVs. It is possible for the LFSR to enter the all-zero state during MAC computation, but it is extremely unlikely that it would remain in such a state.

4.3 A Summary of Previous Analyses of SOBER-t32

Guess and Determine (GD) attacks. The analysis by the designers [22] concluded that the best attack on un-stuttered SOBER-t32 would have a computational complexity of $O(2^{320})$. This analysis assumed that GD attacks required the attacker to guess all the bits of words, rather than a subset of the bits. De Canniere [5] showed that it was possible to improve the attack by guessing a subset of bits of words, resulting in an attack of complexity $O(2^{304})$. Babbage et al [1] further reduced the complexity to $O(2^{252})$. The analysis by Babbage et al is discussed in further detail below.

Analysis of the S-box: In an initial analysis of SOBER-t16 and SOBER-t32 for the NESSIE project, Schafheutle [34] examined linear and differential properties of the function S-boxes used in SOBER-t16 and SOBER-t32.

Analysis of the Key Loading: In determining the amount of mixing required for the key loading, the analysis by the designers [22] concluded that full diffusion was achieved after 17 rounds of the Diffuse() operation after the key material has been included into the register. This analysis was based on the NLF providing good diffusion across the entire output word (that is, every bit of the output word depends on every input bit). An initial statistical evaluation by Schafheutle [35] did not reveal any weakness in the key loading. A later analysis by Ditchl and Schafheutle [13] found an undesirable property of the key loading. They showed that with high probability (close to 100%) there are linear combinations of the initial state bits that do not change when single bits of the key are changed. Their analysis suggested that such linear properties will disappear when more rounds of the Diffuse() operation are applied (as is the case in SOBER-128). Our analysis suggests that the poor diffusion is also because *Konst* was set to zero during the key loading operation. We have addressed this by using a non-zero *Konst* during key loading, and a better *NLF* function.

Distinguishing Attacks: Ekdahl and Johansson [15] found distinguishing attacks on unstuttered SOBER-t32. The distinguishing attack on unstuttered SOBER-t32 required around 2^{87} output words from the NLF function. The attack stems from the low non-linearity of the NLF. The new NLF function in SOBER-128 increases the complexity of the corresponding attack on SOBER-128 to more than 2^{128} words. Babbage *et al*. [1] extended the attack of Ekdahl and Johansson on full SOBER-t16 to an attack on SOBER-t32. The attack on un-stuttered SOBER-t32 has a complexity of around 2^{128} outputs.

4.4 Analysis of SOBER-128

This analysis concentrates on vulnerability of SOBER-128 to known-plaintext attacks. An unknown-plaintext attack on a stream cipher uses statistical abnormalities of the output stream to recover plaintext, or to attack the cipher. The underlying LFSR has very good statistical properties (see Sect. 4.4.4.1), which are preserved and enhanced by the subsequent operations, so there is no significant risk of an unknown-plaintext attack.

This analysis is arranged as follows. Section 4.4.1 outlines a heuristic argument for the strength of SOBER-128. Section 4.4.2 describes the results of statistical analyses conducted on key streams generated by SOBER-128. Properties of the LFSR are analysed in Sect. 4.4.4. The security offered by the combination of the LFSR and NLF is considered in Sect. 4.4.5. The key loading is analysed in Sect. 4.4.6.

4.4.1 A Heuristic Argument for the Security of SOBER-128

SOBER-128 is constructed from components that combine to form a secure cipher: each component contributes different security characteristics.

- The LFSR provides a sequence of states with good statistical properties. However, an LFSR by itself is too weak because as few as 17 L-words can be used to reconstruct a state, due to the linear relationship between L-words.

- The NLF ensures that there is no linear relationship between the outputs and the L-stream. This is required so that an attacker cannot exploit the linearity inherent in the LFSR. Our analysis indicates that the combination of the LFSR and NLF appears to be sufficient to resist GD attacks. Heuristic arguments suggest that the LFSR and NLF also resist correlation-based attacks (see Sect. 4.4.5.2).

- The NLF has a very high algebraic order, and SOBER-128 does not appear to be vulnerable to algebraic attacks.

- The key loading is designed to ensure that every bit of the initial state is a complex function of every bit of the session key and IV. This would appear to be sufficient to resist attacks exploiting the key loading, such as related-key attacks.

- Finally, because SOBER-128 is based on a single shift register, general divide-and-conquer attacks do not seem to be applicable.

4.4.2 Statistical Analysis

Key streams generated by SOBER-128 have been analysed using the CRYPTX'98 statistical analysis program [9]. This program performs various statistical tests. Some tests measure if the pattern has properties of a random bit stream. CRYPTX'98 reported that the SOBER-128 key streams tested have the properties of random bit streams. Other CRYPTX'98 tests look for a class of statistical properties than can be exploited by various classes of attacks. The analysis reported that SOBER-128 key streams had none of these properties. This encourages us to believe that SOBER-128 has no glaring weaknesses.

4.4.3 Exhaustive State Search

For any $t \geq 0$, we define a *candidate word* u_t to be a guess for the value of the LFSR state word $s[t]$, and define a *candidate state* $\mu_t = (u_t, \ldots , u_{t+16})$ to be a guess for the values of the state σ_t. SOBER-128 can be considered broken once the initial state of the LFSR has been determined as then it is possible for an attacker to reconstruct the entire key stream. One method by which a stream cipher can be attacked is to search through every candidate state μ_t until the value of σ_t is found. This process of searching through every candidate for a certain value is commonly known as *guessing*. In this case the attack guesses candidates for the value of the state. A candidate state μ_t is *tested* (to see if it is correct) by constructing a key stream using this value μ_t, and comparing the resulting key stream with the observed key stream. If the two streams match, then the candidate is correct.

This attack (guessing every possible candidate state and testing each candidate state) is an *exhaustive state search*. In SOBER-128, the large size of the register (544 bits) and the corresponding large number of possible candidate states (there is total of $(2^{544}-1)$ states) makes any such attack prohibitive.

4.4.4 Analysis of the LFSR

Consider the linear recurrence for SOBER-128: $s[t+17] = s[t+15] \oplus s[t+4] \oplus \alpha\, s[t]$. The *characteristic polynomial* is the polynomial $p(x) = x^{17} + x^{15} + x^4 + \alpha$, where multiplication and addition is performed over $GF(2^{32})$. Each polynomial uniquely defines a linear recurrence, so the characteristic polynomial, rather than the linear recurrence, is often used to define an LFSR. Using $GF(2^{32})$ instead of $GF(2)$ in the shift register has very little effect on the properties of the register itself. The linear recursion over $GF(2^{32})$ can be shown to be equivalent to implementing 32 parallel bit-wise LFSRs, each of length $17 \cdot 32 = 544$ (see [26]). These linear recurrences are all the same, represented by a polynomial $p_2(x)$ over $GF(2)$ (shown in binary, with the first bit being the constant term and increasing exponent):

1000000000000000010101010101011101110111011000110110001111100011011011101011101110011001111011011111100111110001111011011010010010110111100010111111011010111100000000000010111111111101001100011010010101100101010111100000001001110100111010001110011111100101001010111011111011101000101010001001110100000011000110101011010000101011001100011010011100111101000011001001100001011010110000100100011000101111111001010010001111110011001100001011111100100110000001110101111010110100111001111011100001010000000100001110000100010100001000000111010111010110100111001111011100001010000000100001110000100010100001000000100010000001000000001

That is, $p_2(x) = 1 + x^{17} + x^{19} + x^{21} + x^{23} + x^{25} + x^{27} + x^{29} + x^{30} + \ldots + x^{520} + x^{528} + x^{536} + x^{544}$. This polynomial has 267 nonzero terms.

Note: The original version of this paper gave an incorrect binary expansion of the chosen polynomial, and claimed that this polynomial had 273 nonzero terms. This error was pointed out to us by Georny Lou; many thanks.

The period of the LFSR, hence the period of SOBER-128, is 2^{544}-1 output words (when the effect of MAC accumulation is ignored).

4.4.4.1 Properties of the L-Stream

Properties of the word-oriented LFSR over $GF(2^{32})$ can be deduced from the bit-wise LFSR over $GF(2)$. The connection polynomial for the bit-wise LFSR is *primitive*: that is, it is irreducible (has no factors) and it divides $(x^d + 1)$ for $d = 2^{544}$-1, but not for any d that divides $(2^{544}$-1). The statistical properties of the sequence output by such an LFSR (M-sequence) are well known.

- The period of the output bit-sequence (also known as the *sequence length*) of $p_2(x)$ is known to be $(2^{544}$-1): this is the maximum sequence length for a 544-bit LFSR. Consequently, the sequence

output by the SOBER-128 LFSR must have a period of at least (2^{544}-1) words. It is impossible for the sequence to have a length of more than (2^{544}-1) states as there are only (2^{544}-1) possible states. Therefore, the sequence output by the SOBER-128 LFSR must have a sequence length of exactly (2^{544}-1) words.

- The bit stream generated by the bit-wise LFSR has many statistical properties expected for a random bit stream. See Definition 5.28 (p180) and Fact 6.14 (p197) of [30] for further details. The linear recurrence used to generate the L-stream results in many linear relationships between L-words. These linear relationships correspond to multiples of the connection polynomial. That is, for every such linear relationship $\Sigma\ a_i\ s[t+i] = 0$, the polynomial $r(x) = \Sigma\ a_i\ x^i$ is a multiple of the connection polynomial $p(x)$.

4.4.5 Analysis of the combination of the LFSR and NLF

The LFSR provides the statistical properties essential for a good stream cipher. However, the LFSR itself is not cryptographically strong. Every output bit is a linear function of the initial state. Thus recovering any linearly independent set of enough L-stream bits, or linear functions thereof, from any known positions in the output sequence yields the state of the register. The NLF "hides" the linearity so output bits are not linear functions of the initial state, while also maintaining the good statistical properties of the L-stream.

The combination of the LFSR and NLF is very strong. To date, the most successful attacks on this combination appears to be the GD attacks. The combination of the LFSR and NLF appears to provide sufficient resistance to GD attacks, as the best such attack has a complexity of 2^{252} (see Sect. 4.4.5.4), significantly larger than the complexity of 2^{128} for an exhaustive key search. However, research in this area is ongoing.

The LFSR and NLF might be susceptible to correlation-based or distinguishing attacks, but it would seem unlikely (see Sect. 4.4.5.2). We are aware of an attack that exploited the near-linearity of the least significant bits (LSBs) in the original SOBER design to recover the initial state from a few more than 136 consecutive octets. However, this attack relied on NLF output bits that exhibit high correlation to the input (the output bits of SOBER-128 do not exhibit any high correlation).

4.4.5.1 Some Comments on the NLF

Statistical Properties. Note that the NLF is constructed from group operations (modular addition and XOR), bit rotation and the fixed one-to-one mapping *f*. This construction of the NLF ensures that the keystream exhibits the good statistical properties of the L-stream. In particular, the sequence length of the keystream is (2^{544}-1).

Non-linearity. The carry bits in the word addition, and the nonlinear S-box account for most of the nonlinear behavior in the NLF. 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 LSB of the sum. The LSB of the output would be the XOR of the LSBs of the inputs: an entirely linear function of the L-stream. 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-box also contributes non-linearity into other bits. XORing the *Konst* destroys commutation between the middle two modular additions in the NLF. The second pass through the S-box, after rotation, significantly reduces overall biases. None of the NLF operations commute, since *Konst* cannot be zero. The high byte of *Konst* is nonzero, ensuring that at least the second S-box operation introduces some degree of unpredictability to the output.

Algebraic Degree: Each bit in a keystream word can be expressed as a multivariate polynomial function of bits of the corresponding LFSR state: this multivariate polynomial is the algebraic normal form (ANF). A typical definition of the algebraic degree is the minimum degree of the 32 ANFs. An alternate definition is the minimum degree of the $(2^{32}-1)$ non-trivial linear combinations of the 32 ANFs. To the best of our knowledge, the algebraic degree (according to either definition) is large. We have not done a detailed study, but as first approximation, the input to the S-box in the last instance of the *f* function depends on least 48 bits of the state: the most significant bytes of $s[t]$, $s[t+16]$, $s[t+1]$, and $s[t+6]$; and the least significant bytes of $s[t]$ and $s[t+16]$. As a conservative estimate, we believe the algebraic degree will exceed 48/2 = 24. Research in this area is ongoing.

Linear Span: The linear span is the length of the smallest LFSR that can generate the key stream. Key [27] showed that if the underlying LFSR of the cipher is of length n the algebraic degree is d then the linear span is likely to be $N = \sum_{1 \le i \le d} {}^{n}C_i$, where ${}^{n}C_i$ is the value of "n choose i". In the case of SOBER-128, $n = 544$ and we estimate $d > 24$. These figures result in a linear span N of $4.55 \times 10^{41} = 2^{138.4}$ that far exceeds our bound on the data complexity of 2^{80}.

Analysis of the Function f and the S-box: The function f serves three purposes.

1. The function *f* ensures that the addition of s_t and s_{t+16} does not commute with the addition of s_{t+1} and s_{t+6}, and s_{t+13}.

2. The function *f* ensures that every bit of the output of the NLF depends algebraically on every bit of s_t and s_{t+16}.

3. The function *f* "carries" nonlinearity from high order bits of sums to low order bits.

The S-box used in SOBER-128 was designed to have good non-linearity properties. The S-box is restricted to 256 entries to reduce memory requirements. The S-box uses the eight most significant bits of $(s_t + s_{t+16})$ as input to ensure that even the least significant bits of s_t and s_{t+16} can affect the input to the S-box (through carry propagation). In this way, every bit of s_t and s_{t+16} can affect every bit of the output of *f*.

4.4.5.2 Correlation Attacks

Two conditions are required for a cipher to be susceptible to a correlation-based attack. First, there must be a significant correlation between the NLF outputs and the L-stream. Second, there must be a multiple of the bit-wise connection polynomial of low weight (low number of terms) and low degree. Neither of these conditions appears to be satisfied in SOBER-128.

- Due to the word-oriented nature of the cipher, it would appear very difficult to determine a high correlation between the bits output by the NLF and bits in the L-stream. It would seem that any such relationship has low correlation, due largely to the highly non-linear S-box.

- The word-oriented connection polynomial over $GF(2^{32})$ may have low weight, but the corresponding bit-wise connection polynomial has very high weight: $p_2(x)$ has 272 non-zero coefficients. This leads us to assume that there are no low-degree, low-weight multiples (we have not yet confirmed that this is true). This, in combination with low correlation, leads us to believe that a correlation attack will have a very large complexity.

4.4.5.3 Inversion Attacks

Inversion attacks [19] are so named because they "invert" the operations of the NLF: rather than using the inputs to determine the NLF output, an input is determined from the NLF output and the remaining inputs. The attacks were initially conceived as attacks on bit-wise LFSR-based ciphers with an NLF. The NLF had to be a linear function in the first or last input; for example, $v_t = \text{NLF}(\sigma_t) = g(s[t], \ldots, s[t+x-1]) + s[t+x] \pmod 2$. In this case, the L-words are single bits, with $s[t+x]$ denoting the last input to the NLF.

Suppose that the values $u_t, \ldots, u_{t+x-1}$, are candidates for the L-words/bits $s[t], \ldots, s[t+x-1]$. If these candidates were correct then this would imply that:

$$u_{t+x} = v_t + g(u_t, \ldots, u_{t+x-1}) \pmod 2,$$

$$u_{t+x+1} = v_{t+1} + g(u_{t+1}, \ldots, u_{t+x}) \pmod 2,$$

$$u_{t+x+2} = v_{t+2} + g(u_{t+2}, \ldots, u_{t+x+1}) \pmod 2,$$

and so-forth. We say that the candidates u_{t+x}, u_{t+x+1} and so forth have been *determined*. In an inversion attack, the attacker guesses the values for the candidates $u_t, \ldots, u_{t+x-1}$, and "inverts" the NLF to determine further candidates until a full candidate state μ_t has been determined. This state is then tested. Provided x is less than the register length, such an approach can offer a significantly lower complexity than an exhaustive state search.

The inversion attack can be easily extended to word-oriented stream ciphers. The attacker guesses the values of $u_t, \ldots, u_{t+x-1}$, (these values are words) and inverts the NLF to determine u_{t+x}, u_{t+x+1} and so

forth. In the case of SOBER-128, the attacker would guess values for the candidates u_1, …, u_{16} and *Konst*. Then the attacker determines the candidate u_{17} from the keystream word v_1 by inverting the NLF:

$$u_{17} = f^1(((f^1 (v_t - u[t+13]) - u[t+6]) \oplus konst) - u[t+1]) <<< 8) - u[t],$$

where "+" denotes addition modulo 2^{32} and "-" denotes subtraction modulo 2^{32}. The attacker now has a full candidate state μ_1. This state is then tested.

Unfortunately for the attacker, this attack requires guessing 544 bits of information. There is no advantage to performing such an attack on SOBER-128: an exhaustive key search has a considerably lower complexity. Hence, SOBER-128 is resistant to the inversion attack.

4.4.5.4 Guess-and-Determine Attacks

The inversion attack is based on exploiting the NLF to determine candidate L-words from a smaller set of candidate words (although this proves to be of no advantage in the case of SOBER-128). *Guess-and-determine attacks* (GD-attacks) are "smart" inversion attacks that not only exploit the NLF: these attacks also exploit the LFSR. There are linear relationships between certain sets of L-words which are a result of producing the L-stream using a linear recurrence (recall that this linear recurrence is $s[t+17] = s[t+15] \oplus s[t+4] \oplus \alpha\ s[t]$). These relationships can be exploited to determine additional candidate L-words from guessed and determined candidate L-words. For example, suppose that the candidate L-words u_t, u_{t+4} and u_{t+17} have been guessed or determined. Assuming that these candidates are correct, this would imply that:

$$u_{t+15} = u_{t+17} \oplus u_{t+4} \oplus \alpha\ u_t.$$

In this case, the candidate u_{t+15} can be determined by exploiting the linear recurrence directly. An attacker can also exploit other linear relationships between L-words (corresponding to multiples of the connection polynomial, as noted in Sect. 4.4.5.2).

Given a suitable portion of the keystream, a guess-and-determine attack is based on: **guessing** candidates for a small set of L-words; exploiting the NLF, linear recurrence and other linear relationships to **determine** a full candidate state; and then **testing** the candidate state. Examples of GD-attacks can be found in [1,3,4,21,22,33].

Recall that SOBER-II, SOBER-t32 and SOBER-128 have the same overall structure. SOBER-128 can be attacked by extending the known attacks on SOBER-II and S16 (such as [4,21]). These attacks, when applied to SOBER-128, have a computational complexity of approximately 2^{320} and require a small number of keystream words. We developed an attack search program that looked for improvements on these attacks. The search examined every GD-attack exploiting linear relationships corresponding to polynomials of degree 34 or less, and with 10 or fewer terms. No attacks on SOBER-128 were found that improved on the previous attacks. More details are provided in [24].

Babbage *et al.* [1] found a GD attack on un-stuttered SOBER-t32 that exploited the fact that the most significant byte of a keystream word depended only on the most significant byte of the state L-words and some carries propagated by the addition operation. The computational complexity of the attack is approximately 2^{252}. Babbage *et al*. noted, "… a cyclic shift at the end of the S-box would have destroyed the attack". SOBER-128 incorporates this change, and is thought to resist this, and similar, attacks. Research regarding this type of analysis is ongoing.

4.4.5.5 Timing Attacks and Power Attacks

In this analysis, we consider only components of the algorithm where there is a variation in the "atomic" operations performed (such as XOR, addition and table lookups). All of the components in SOBER-128 require the same atomic operations, and there are no conditional executions other than in selection of *Konst* during key setup. We do not believe that SOBER-128 is vulnerable to timing attacks.

We have not examined the ramifications of Simple Power Analysis attacks, in which the power consumption of operations is related to the Hamming weight of the operands. Differential Power Analysis and fault analysis would seem to be precluded under the requirement that IVs (and hence initial states) are not repeated.

4.4.5.6 Algebraic Attacks

Courtois and Meier [8] introduced algebraic attacks on stream ciphers. The feasibility of the attack depends on the algebraic degree (see Section 4.4.5.1). The attack is based on expressing the key stream bits as a multivariate polynomial function of the bits in the initial state. If the algebraic degree is d then the number of possible monomials in these polynomials is $N = \sum_{1 \le i \le d} {}^{n}C_i$, where ${}^{n}C_i$ is the value of "n choose i". These monomials can be treated considered independent variables, so each output bit is a linear combination of N variables. If an attacker obtains N key stream bits, then they have a solvable system of equations. The solution reveals the initial state. This process of replacing the monomials with independent variables is called linearization.

Further extensions of the attack can be found in [7,8]. The attack can be extended by finding multivariate functions in which the monomials are a product of both bits of the initial state and bits of the keystream. If the monomials are of degree d in the initial state bits, then (after substitution of the keystream bits) the attacker has equations that are a linear combination of N possible monomials. The initial state can then be determined through linearization as above. This extension of the attack reduces the complexity significantly, since it was shown that if output bits depend on D bits of the LFSR state, then there exist such multivariate equations of degree $d = D/2$ in the initial state bits. Notice that there is a close relationship between the linear span (Section 4.4.5.1) and this attack.

The data complexity of the attack is lower bounded by N. The process complexity is dominated by the time required to solve the linear system in N variables. In practice, this complexity is around $7 \cdot N^{2.8}$.

In the case of SOBER-128, $n = 544$ and we predict that $D > 48$ and $d > 24$ (see Section 4.4.5.1). This corresponds to a data complexity of $4.56 \times 10^{41} = 2^{138.3}$ and process complexity of around 2^{390}. These complexities greatly exceed our bounds; consequently, we believe SOBER-128 resists algebraic attacks.

4.4.5.7 Distinguishing Attacks

Ekdahl and Johansson [15] found distinguishing attacks on both SOBER-t16 and unstuttered SOBER-t32. The distinguishing attack on unstuttered SOBER-t32 requires around 2^{87} output words from the NLF function. The attack exploits a correlation between (1) the XOR of adjacent bits of the NLF stream and (2) the XOR of corresponding bits of the state values. The stronger NLF in SOBER-128 significantly reduce the biases observed in [15], so that no bias (averaged over all possible *Konst* values) exceeds 0.000345, at which point the data complexity of the attack is greater than 2^{128} words.

We have used specialised tools to further examine biases in the NLF output. No single-bit bias, or bias with consecutive pairs of bits, is large enough to allow the attack in [1] with less than 2^{128} words. We have found biases in the function:

$$F(X, Y) = NLF(X) \oplus NLF(Y) \oplus NLF(X \oplus Y)$$

(where X and Y are pseudorandom sets of 5 input words). These biases are on the edge of statistical significance with 2^{44} inputs. This leads us to believe there might be a distinguishing attack requiring somewhat more than 2^{100} words, based on the 6-term recurrence relationship in [1].

Research in this area is ongoing.

Babbage *et al.* [1] extended the attack of Ekdahl and Johansson on unstuttered SOBER-t32 to an attack on full SOBER-t32. This attack does not concern SOBER-128. Babbage *et al*. also extended the attack of Ekdahl and Johansson on full SOBER-t16 to an attack on SOBER-t32. This attack exploits the non-uniform distribution of

$$w_t = v_t \oplus (s[t] \oplus s[t+1] \oplus s[t+6] \oplus s[t+13] \oplus s[t+16]).$$

Over $GF(2^{32})$ given random inputs $s[t]$, $s[t+1]$, $s[t+6]$, $s[t+13]$, and $s[t+16]$. The attacker combine v_t, v_{t+4}, v_{t+15}, v_{t+17}, to eliminate the L-words:

$$W_t = \alpha \cdot v_t \oplus v_{t+4} \oplus v_{t+15} \oplus v_{t+17} = \alpha \cdot w_t \oplus w_{t+4} \oplus w_{t+15} \oplus w_{t+17}.$$

If the sequence $\{v_t\}$ is random, then the distribution of W_t will also be random. However, Babbage *et al.* noted that when $\{v_t\}$ is produced by un-stuttered SOBER-t32, then the distribution for W_t can be detected using around 2^{128} outputs.

Since SOBER-128 includes a stronger NLF incorporating a cyclic rotation, the non-uniformity of W_t for SOBER-128 may differ from the non-uniformity of W_t above. If there is any change in non-uniformity, it is anticipated that the non-uniformity will be less rather than more, and so it is reasonable to predict that 2^{128} is a lower bound on the complexity of a similar distinguishing attack on SOBER-128.

4.4.6 Analysis of the MAC construction

The MAC construction, although it is modeled after that of Helix [16], is entirely new to SOBER-128, so we cannot say with certainty that it does not introduce weaknesses.

Clearly, with nonlinear plaintext feedback, the "LFSR" is no longer linear. Consequently the guarantee of a long period no longer applies; nevertheless, the state space is sufficiently large, and the linear diffusion good enough, that we believe this does not weaken the operation of the cipher.

We make the following observations to justify the security of the MAC output:

The MAC output is generated by the stream cipher operation after a process that is equivalent to loading an IV. Therefore any attack that uses the MAC to recover information about the LFSR state would be equivalent to a known-plaintext state recovery on the stream cipher component.

Differential attacks on the stream cipher output after MAC accumulation are forbidden by the threat model, that is, reuse of the cipher for the same key/IV pair is forbidden. This was already a security requirement for safe use of the stream cipher (equally for all modes of use of a block cipher that require an IV), so we feel this is reasonable.

The remaining avenue of attack would seem to be to exploit correlations between input plaintext and resulting ciphertext. We attempt to counter this by performing the feedback function in a manner that depends on unknown state and with very high nonlinearity, at the same time ensuring high diffusion of the result through the LFSR, by the placement of the feedback with respect to the characteristic polynomial.

4.4.7 Analysis of the Key Loading

The key loading was designed to ensure that (after all key material has been included), the following properties hold:

- Every bit of the initial state is a non-linear function of every bit of the key and IV.
- For any session key, the set of initial states corresponding to the IV cannot be restricted to a linear subspace of $(GF(2^{32}))^{17}$. 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.
- The key length is included to ensure that there are no equivalent secret keys or equivalent IVs.
- No two secret keys (up to 128-bits in length) can result in the same initial key state. Also, given a key state, no two IVs (up to 128-bits in length) can result in the same initial state.

We believe that these properties ensure that the key loading cannot be exploited. The key loading in previous SOBER proposals did have weaknesses. In [4], Bleichenbacher, Patel and Meier discuss weaknesses in IV setting in SOBER-II (assuming correctly that analogous problems appeared in S16). The results demonstrated that there is correlation between related messages generated from the same initial key material for SOBER-II. However, we note that no actual attack based on this correlation is proposed. Nevertheless, we agree with the conclusion that the correlation is undesirable, and this was fixed in the t-class SOBER ciphers.

4.4.7.1 Weak Keys

If the initial state is entirely zero then the algorithm will cycle forever producing the same NLF output: f(f(0) $\oplus$ *Konst*). The probability that a random state is entirely zero is 2^{-544}. There are a maximum of $2^{128+128} = 2^{256}$ possible initial states derived from the 2^{128} keys and 2^{128} IVs. Thus, the probability that any of these initial states is entirely zero is less than 2^{-288}. In addition, we believe that the all zero state cannot occur using the stated key and IV sizes.

5 Strengths and Advantages of SOBER-128

SOBER-128 has the following strengths and advantages.

- Speed: SOBER-128 is fast, and in particular, compared to RC4, key loading is quite fast.
- SOBER-128 allows simultaneous encryption and message integrity
- Requires a small amount of memory.
- Flexibility in the processor size and implementation.
- The design allows for the use of a secret key and non-secret IV.
- LFSR-based (the properties of LFSRs are well-known).
- Utilises an NLF in a manner that has been well studied.
- Appears to provide more than adequate security.

6 Design Rationale of SOBER-128

The design rationale fall under the following subheadings:

1. The choice of LFSR and NLF taps.
2. The choice of LFSR multiplication constants.
3. The design criteria for the NLF, PFF, the function *f* and the S-box.
4. The design criteria for the key loading.

6.1 The LFSR and NLF Taps

The set of exponents of the connection polynomial with non-zero coefficients is called the *LFSR tapset*, denoted $T = \{0,4,15,17\}$. Note that the LFSR tapset contains the indices for inputs to the linear recurrence, combined with the register length. The *NLF tapset* $\Gamma = \{0,1,6,13,16\}$, contains the inputs to the NLF. The values in a tapset are called *taps*. Given a tapset S, we define the positive difference set $\Delta(S)$ to be the set of positive differences between the taps in that set. Consequently, $\Delta(T)=\{2,4,11,13,15,17\}$, while $\Delta(\Gamma)=\{1,3,5,6,7,10,12,13,15,16\}$. Each of the tapsets is a *full positive difference set* (FPDS), meaning that no positive difference is repeated. Full positive difference sets are highly recommended for LFSR-based ciphers (see [18,28]).

The first step in designing SOBER-128 was to choose the LFSR and NLF tapsets. The existence of GD attacks relies on the tapsets, rather than the individual operations with the NLF and LFSR. So it was possible to choose the tapsets for the resistance to GD attacks, and then consider the details of the LFSR and NLF. The tapsets were chosen using the following criteria:

1. There must be four LFSR taps (including the feedback tap), and they form an FPDS.
2. There must be five NLF taps and they form an FPDS.
3. The sets $\Delta(T)$ and $\Delta(\Gamma)$ should share only a small number of differences.
4. The choice of tapsets should provide good resistance to all GD attacks.

It was decided to use the tapsets from SOBER-II and S16, also used in the t-class ciphers and Turing [25]. These tapsets have been well analysed and appear to provide good resistance against GD attacks.

6.2 The Multiplication Constants in the LFSR

Given the LFSR taps, it remains to determine what multiplication constants to use. There were three criteria used to select the multiplication constants: computation considerations, sequence period considerations and bit-wise polynomial weight considerations.

1. **Computation considerations.** The computation required to clock the LFSR is dominated by the computation required to perform the field multiplication operations. However, multiplication by one is performed for free as $1 \cdot x = x$ in $GF(2^{32})$. *The first design criterion for the choice of multiplication constants is that two multiplication constants should be 1.*

2. **Sequence length considerations.** If the connection polynomial $p(x)$ is primitive, then $p_2(x)$ is primitive and the L-stream has maximal sequence length. *The second design criterion for the choice of multiplication constants is that the connection polynomial must be primitive.*

3. **Bit-wise polynomial weight considerations.** Several papers have shown that low-weight polynomials over $GF(2)$ can be exploited in correlation-based attacks [6,18,29]. *The third design criterion for the choice of multiplication constants is that the bit-wise connection polynomial has approximately half the coefficients equal to one (excluding the leading coefficient)*.

4. **Implementation:** Following the design of SNOW[14], we realise $GF(2^{32})$ as $GF((2^8)^4)$ for implementation efficiency.

6.3 The Non-Linear Filter

The following criteria were applied when designing the NLF.

1. **Memory Constraints.** The processors for which SOBER-128 is designed are likely to have restrictions on the amount of ROM available. The design was restricted to a single S-box was containing only 256 entries where each entry is a 32-bit word.

2. **Balanced Output.** To preserve the statistical properties of the L-stream, the NLF should be balanced (that is, every output word occurs with equal probability assuming that the state is uniformly distributed). Furthermore, we place an additional requirement that if any five of the six inputs (including *Konst*) are fixed then every value for the remaining input corresponds to a unique output.

3. **The Function** *f*. The function *f* has a 32-bit input and a 32-bit output. The S-box in the function *f* is restricted to an 8-bit input (due to memory constraints on the NLF). We desired the output of *f* to be a non-linear function of all bits of s_t and s_{t+16}. This is achieved by using the eight MSBs of $(s_t + s_{t+16})$ as the input to the S-box. Thus, even the 24 LSBs of s_t and s_{t+16} can affect (through carry propagation) the input to the S-box, and thus affect every output bit. To ensure that the "balanced output" criterion is satisfied, the function *f* was also required to be a one-to-one mapping (or permutation). Given the structure of *f*, this requirement is satisfied when the 8 MSBs output from the S-box form an 8-bit one-to-one function of the 8-bit input to the S-box.

4. **The S-box.** The main criteria for the S-box are that it be non-linear, and that the 8 MSBs must form an 8-bit one-to-one function. The Skipjack S-box was used for the eight MSBs for three reasons. First, the Skipjack S-box is one-to-one; second, it is highly non-linear; and third, we

wished to avoid suspicion that we had designed a trapdoor in the S-box. The ISRC portion of the S-box was constructed as 24 mutually-uncorrelated, balanced and highly non-linear single bit functions of the 8-bit input. We specified the requirements for the S-box, and then selected the best conforming S-box from external sources. We believe that any S-box selected with the requirements would be equally secure, but are specifying the particular one given.

6.4 The Key Loading

The key loading was designed to be an efficient method of combining key material into the LFSR state without requiring much additional coding. The key loading utilises the LFSR and NLF, combining the output from the NLF back into a certain position in the register to provide non-linearity. The variables in the construction of the key loading were:

- The position in the register to which key material is added in the Include() operation (the *input position*).
- The position in the register to which the NLF output is XORed in the Diffuse() operation (the *feedback position*).
- The number of Diffuse() operations applied after all key material has been included (the *diffusion number*).

The input position and feedback position were chosen to minimise the number of Diffuse() operations required to ensure that (after all key material has been included), the following properties hold:

- Every bit of the initial state is a non-linear function of every bit of the key and IV.
- For any session key, the set of initial states corresponding to the IV cannot be restricted to a linear subspace of $(GF(2^{32}))^{17}$.
- No word of the initial state is algebraically related to any subset of other words.

Our analysis of SOBER-t32 revealed that the minimum diffusion number was 17, and this applied when the input position is r_{15} and the feedback position is r_4 . Further analysis by Ditchl and Schafheutle [13] revealed a weakness when using 17 Diffuse() operations and when *Konst* = 0, although this weakness disappeared when 21 Diffuse() operations are used. SOBER-128 uses a stronger *NLF* and non-zero *Konst* during key loading to avoid this weakness.

7 Performance

Table 1 and Table 2 contain performance figures for SOBER-128. The figures are for optimized C-language code and were obtained on a 2.8GHz Xeon running Linux and compiled with the command line `gcc -O3 -march=i586.` Assembler code can be expected to perform significantly better.

Keys and IVs are each 128 bits. MACs are 64 bits. These figures are for the "s128fast.c" source code available at our web site http://www.qualcomm.com.au .

Key Loading		IV Loading	
MKeys/s	cycles/key	M IVs/s	cycles/IV
2.05	1363.60	2.47	1134.00

Table 1. Computation required for key loading and IV loading of SOBER-128. These results were obtained by averaging the measurements for a large number of keys.

Length	Operation	Mbyte/second	cycles/byte
Continuous	Stream encryption	572.2	4.89
1600-byte blocks	Stream encryption	476.2	5.87
	MAC generation	363.6	7.71
	MAC generation and encryption	259.7	10.78
	Decryption and MAC generation	277.8	10.07

Table 2. Performance figures for encryption, decryption and MAC generation. The block figures include the time for IV loading.

The implementation requires 140 bytes of RAM. This accounts for:

- the register and *key state* (34 words or 136 bytes),
- *Konst* (four bytes).

Small memory implementation would omit the key state and just re-key the cipher for each IV; in this case 72 bytes of RAM are required.

ROM memory is required for the field multiplication table (used in the LFSR), and the non-linear S-box (used in the NLF); all these arrays are fixed. The multiplication table contains 256 words. The non-linear S-box also contains 256 words. Thus 2048 bytes of ROM are required.

8 References

1. S. Babbage, C. De Cannière, J. Lano, B. Preneel and J. Vandewalle, Cryptanalysis of SOBER-t32, Pre-proceedings of *Fast Software Encryption FSE2003,* February 1999, pp. 119-136.

2. 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.

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. C. De Cannière, Guess and Determine Attack on SOBER, *NESSIE Public Document NES/DOC/SAG/WP5/010/a*, November 2001. See [31].

6. V. Chepyzhov and B. Smeets. On a fast correlation attack on certain stream ciphers. *Advances in Cryptology, EUROCRYPT’91, Lecture Notes in Computer Science, vol. 547*, *D. W. Davies ed*., *Springer-Verlag*, pages 176-185, 1991.

7. N. Courtois, Fast Algebraic Attacks on Stream Ciphers with Linear Feedback, awaiting publication. See `http://www.minrank.org/~courtois/myresearch.html`.

8. N. Courtois and W. Meier, Algebraic attacks on Stream Ciphers with Linear Feedback, To be published in the proceedings of *EUROCRYPT 2003*, Warsaw, Poland, May 2003.

9. E. Dawson, A. Clark, H. Gustafson and L. May, “CRYPT-X’98, (Java Version) User Manual”, *Queensland University of Technology*, 1999.

10. 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.

11. M. Dichtl. Statistical Results for the NESSIE submission SOBER-t16, *NESSIE Public Document NES/DOC/SAG/WP3/015/2*, March, 2001. See [31].

12. M. Dichtl. Statistical Results for the NESSIE submission SOBER-t32, *NESSIE Public Document NES/DOC/SAG/WP3/016/2*, March, 2001. See [31].

13. M. Dichtl and M. Schafheutle, Linearity Properties of the SOBER-t32 Key Loading, *NESSIE Public Document NES/DOC/SAG/WP5/046/1*, November 2001. See [31].

14. P. Ekdahl, T. Johansson, SNOW - a new stream cipher, Proceedings of First Open NESSIE Workshop, KU-Leuven, 2000. See [31].

15. P. Ekdahl and T. Johansson, Distinguishing Attacks on SOBER-t16 and t32, *Fast Software Encryption Workshop (FSE) 2002, Lecture Notes in Computer Science, vol. 1976, J. Daemen, V. Rijmen (Eds.), Springer*, pp. 210-224, 2002.

16. N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks and T. Kohno, Helix Fast Encryption and Authentication in a Single Cryptographic Primitive, Pre-proceedings of *Fast Software Encryption FSE2003,* February 2003, pp. 345-362.

17. FIPS 185- Escrowed Encryption Standard. See the following web page: http://www.itl.nist.gov.fipspubs/fip185.htm.

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

19. J. Dj. Golic, A. Clark, and E. Dawson. Inversion attacks and branching. *Information Security and Privacy, Fourth Australasian Conference, ACISP'99, Lecture Notes in Computer Science, vol. 1587, J. Pieprzyk, R. Savavi-Naini, J. Seberry eds., Springer-Verlag*, pages 88-102, 1999.

20. J. Dj. Golic and M. J. Mihaljevic. A Generalized Correlation Attack on a Class of Stream Ciphers Based on the Levenshtein Distance. *Journal of Cryptology*, 3: 201-212, 1991.

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

22. P. Hawkes and G. Rose. The t-class of SOBER stream ciphers. Technical report, QUALCOMM Australia, Suite 410 Birkenhead Point, Drummoyne, NSW 2047 Australia, 1999. See http://www.qualcomm.com.au.

23. P. Hawkes and G. Rose. *Primitive Specification and Supporting Documentation for SOBER-t32 Submission to NESSIE*. Submitted 2000. See https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sober-t32.zip

24. P. Hawkes and G. Rose. Exploiting multiples of the connection polynomial in word-oriented stream ciphers. *Advances in Cryptology - ASIACRYPT 2000, Lecture Notes in Computer Science, vol. 1976, T. Okamoto (Ed.), Springer,* pp. 303-316, 2000.

25. P. Hawkes and G. Rose. Turing, a Fast Stream Cipher. Pre-proceedings of *Fast Software Encryption FSE2003,* February 1999, pp. 307-324.

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

27. E. Key, An Analysis of the Structure and Complexity of Nonlinear Binary Sequence Generators, *IEEE Transactions on Information Theory, Vol. IT-22, No. 6,* November 1976.

28. B. Löhlein. Analysis of modifications of the conditional correlation attack. 1999. Accepted at 3rd IEEE/ITG Conference on Source and Channel Coding, 17-19 Jan. 2000, Munich.

29. T. Johansson and F. Jönsson. Improved fast correlation attacks on stream ciphers via convolution codes. *Advances in Cryptology, EUROCRYPT'99*, *Lecture Notes in Computer Science*, *vol. 1592*, *J. Stern ed., Springer-Verlag,* pages 347-362, 1999.

30. A. Menezes, P. Van Oorschot, S. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1997.

31. NESSIE: New European Schemes for Signatures, Integrity, and Encryption. See `http://www.cryptonessie.org.`

32. 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.

33. 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.qualcomm.com.au`.

34. M. Schaftheutle, A First Report on the Stream Ciphers SOBER-t16 and SOBER-t32, *NESSIE Public Document NES/DOC/SAG/WP3/025/2*, July 2001. See [31].

35. M. Schaftheutle, The Statistical Evaluation of the SOBER-t Key Expansion , *NESSIE Public Document NES/DOC/SAG/WP3/040/2*, July 2001. See [31].

9 Appendix: Recommended C-language interface

```
typedef struct {
   WORD        R[N];          /* Working storage for the shift register */
   WORD        initR[N];      /* saved post-key register contents */
   WORD        konst;         /* key dependent constant */
} s128_ctx;


/* interface definitions */
void s128_key(s128_ctx *c, UCHAR key[], int keylen);          /* set key */
void s128_IV(s128_ctx *c, UCHAR iv[], int ivlen);             /* set Init Vector */
void s128_stream(s128_ctx *c, UCHAR *buf, int nbytes);        /* stream cipher */
void s128_maconly(s128_ctx *c, UCHAR *buf, int nbytes);       /* accumulate MAC */
void s128_encrypt(s128_ctx *c, UCHAR *buf, int nbytes);       /* encrypt + MAC */
void s128_decrypt(s128_ctx *c, UCHAR *buf, int nbytes);       /* decrypt + MAC */
```

void s128_finish(s128_ctx *c, UCHAR *buf, int nbytes); /* finalise MAC */

Calls to *stream*, *maconly*, *encrypt* and *decrypt* may be arbitrarily mixed (although we don't recommend only encrypting data when other parts of the message are being authenticated). For correct operation, the sender and recipient must agree on which parts of the message receive which treatment. However, the buffering within segments doesn't matter; a long message could be *encrypt*ed in a single buffer at the sender, and then the receiver could call *decrypt* many times, once for each word of data.

For completely synchronous operation as a basic stream cipher, it suffices to call *key*, then *stream* as required. For all other operations, the communication should be broken into messages, and *IV* should be called at the beginning of each message, *finish* at the end of the message, and the other four calls made in an appropriate application-defined sequence in between. IVs should never be reused, but IVs are otherwise opaque to the system, and could easily be based on counters, timestamps, or whatever.

Our reference implementation for these primitives requires that *nbytes* be a multiple of four for all of these calls. It is straightforward to take any odd bit or byte lengths and define a method to handle them correctly, though. We choose not to specify such an interface here, because it almost invariably ends up being byte- or bit- order specific.

We recommend the following pseudocode for encryption of individual plaintext words p that are to be partially authenticated and partially encrypted (to form ciphertext c) under control of bitmasks M_a and M_e respectively:

1. zero a 4-byte buffer z, and call *stream* to generate encryption keystream bytes z_i, where ($0 \le i \le 3$).

2. set $c_i = p_i \oplus (z_i \wedge (M_e)_i)$, which is the output ciphertext.

3. set $z_i = p_i \wedge (M_a)_i$, then call *maconly* to incorporate z into the MAC.

For decryption, just reverse the roles of c and p in step 2 above. This procedure is only slightly less efficient than processing the data word-at-a-time, because it steps the LFSR twice instead of once, but it allows arbitrary mixing of encryption and authentication, and can be used to handle any length of input data.

10 Appendix: The S-box Entries

The entries in the NLF S-box are given below in hexadecimal form.

```
unsigned long SBox[256] = {
0xa3aa1887, 0xd65e435c, 0x0b65c042, 0x800e6ef4,
0xfc57ee20, 0x4d84fed3, 0xf066c502, 0xf354e8ae,
0xbb2ee9d9, 0x281f38d4, 0x1f829b5d, 0x735cdf3c,
0x95864249, 0xbc2e3963, 0xa1f4429f, 0xf6432c35,
0xf7f40325, 0x3cc0dd70, 0x5f973ded, 0x9902dc5e,
0xda175b42, 0x590012bf, 0xdc94d78c, 0x39aab26b,
```

```
0x4ac11b9a, 0x8c168146, 0xc3ea8ec5, 0x058ac28f,
0x52ed5c0f, 0x25b4101c, 0x5a2db082, 0x370929e1,
0x2a1843de, 0xfe8299fc, 0x202fbc4b, 0x833915dd,
0x33a803fa, 0xd446b2de, 0x46233342, 0x4fcee7c3,
0x3ad607ef, 0x9e97ebab, 0x507f859b, 0xe81f2e2f,
0xc55b71da, 0xd7e2269a, 0x1339c3d1, 0x7ca56b36,
0xa6c9def2, 0xb5c9fc5f, 0x5927b3a3, 0x89a56ddf,
0xc625b510, 0x560f85a7, 0xace82e71, 0x2ecb8816,
0x44951e2a, 0x97f5f6af, 0xdfcbc2b3, 0xce4ff55d,
0xcb6b6214, 0x2b0b83e3, 0x549ea6f5, 0x9de041af,
0x792f1f17, 0xf73b99ee, 0x39a65ec0, 0x4c7016c6,
0x857709a4, 0xd6326e01, 0xc7b280d9, 0x5cfb1418,
0xa6aff227, 0xfd548203, 0x506b9d96, 0xa117a8c0,
0x9cd5bf6e, 0xdcee7888, 0x61fcfe64, 0xf7a193cd,
0x050d0184, 0xe8ae4930, 0x88014f36, 0xd6a87088,
0x6bad6c2a, 0x1422c678, 0xe9204de7, 0xb7c2e759,
0x0200248e, 0x013b446b, 0xda0d9fc2, 0x0414a895,
0x3a6cc3a1, 0x56fef170, 0x86c19155, 0xcf7b8a66,
0x551b5e69, 0xb4a8623e, 0xa2bdfa35, 0xc4f068cc,
0x573a6acd, 0x6355e936, 0x03602db9, 0x0edf13c1,
0x2d0bb16d, 0x6980b83c, 0xfeb23763, 0x3dd8a911,
0x01b6bc13, 0xf55579d7, 0xf55c2fa8, 0x19f4196e,
0xe7db5476, 0x8d64a866, 0xc06e16ad, 0xb17fc515,
0xc46feb3c, 0x8bc8a306, 0xad6799d9, 0x571a9133,
0x992466dd, 0x92eb5dcd, 0xac118f50, 0x9fafb226,
0xa1b9cef3, 0x3ab36189, 0x347a19b1, 0x62c73084,
0xc27ded5c, 0x6c8bc58f, 0x1cdde421, 0xed1e47fb,
0xcdcc715e, 0xb9c0ff99, 0x4b122f0f, 0xc4d25184,
0xaf7a5e6c, 0x5bbf18bc, 0x8dd7c6e0, 0x5fb7e420,
0x521f523f, 0x4ad9b8a2, 0xe9da1a6b, 0x97888c02,
0x19d1e354, 0x5aba7d79, 0xa2cc7753, 0x8c2d9655,
0x19829da1, 0x531590a7, 0x19c1c149, 0x3d537f1c,
0x50779b69, 0xed71f2b7, 0x463c58fa, 0x52dc4418,
0xc18c8c76, 0xc120d9f0, 0xafa80d4d, 0x3b74c473,
0xd09410e9, 0x290e4211, 0xc3c8082b, 0x8f6b334a,
0x3bf68ed2, 0xa843cc1b, 0x8d3c0ff3, 0x20e564a0,
0xf8f55a4f, 0x2b40f8e7, 0xfea7f15f, 0xcf00fe21,
0x8a6d37d6, 0xd0d506f1, 0xade00973, 0xefbbde36,
0x84670fa8, 0xfa31ab9e, 0xaedab618, 0xc01f52f5,
0x6558eb4f, 0x71b9e343, 0x4b8d77dd, 0x8cb93da6,
0x740fd52d, 0x425412f8, 0xc5a63360, 0x10e53ad0,
0x5a700f1c, 0x8324ed0b, 0xe53dc1ec, 0x1a366795,
0x6d549d15, 0xc5ce46d7, 0xe17abe76, 0x5f48e0a0,
0xd0f07c02, 0x941249b7, 0xe49ed6ba, 0x37a47f78,
0xe1cfffbd, 0xb007ca84, 0xbb65f4da, 0xb59f35da,
0x33d2aa44, 0x417452ac, 0xc0d674a7, 0x2d61a46a,
0xdc63152a, 0x3e12b7aa, 0x6e615927, 0xa14fb118,
0xa151758d, 0xba81687b, 0xe152f0b3, 0x764254ed,
0x34c77271, 0x0a31acab, 0x54f94aec, 0xb9e994cd,
0x574d9e81, 0x5b623730, 0xce8a21e8, 0x37917f0b,
0xe8a9b5d6, 0x9697adf8, 0xf3d30431, 0x5dcac921,
0x76b35d46, 0xaa430a36, 0xc2194022, 0x22bca65e,
0xdaec70ba, 0xdfaea8cc, 0x777bae8b, 0x242924d5,
0x1f098a5a, 0x4b396b81, 0x55de2522, 0x435c1cb8,
0xaeb8fe1d, 0x9db3c697, 0x5b164f83, 0xe0c16376,
0xa319224c, 0xd0203b35, 0x433ac0fe, 0x1466a19a,
0x45f0b24f, 0x51fda998, 0xc0d52d71, 0xfa0896a8,
0xf9e6053f, 0xa4b0d300, 0xd499cbcc, 0xb95e3d40,
};
```

11 Appendix: The Multiplication Table

The entries in the multiplication table `Multab[]` are given below in hexadecimal form.

```
unsigned long Multab[256] = {
0x00000000, 0xD02B4367, 0xED5686CE, 0x3D7DC5A9,
0x97AC41D1, 0x478702B6, 0x7AFAC71F, 0xAAD18478,
0x631582EF, 0xB33EC188, 0x8E430421, 0x5E684746,
0xF4B9C33E, 0x24928059, 0x19EF45F0, 0xC9C40697,
0xC62A4993, 0x16010AF4, 0x2B7CCF5D, 0xFB578C3A,
0x51860842, 0x81AD4B25, 0xBCD08E8C, 0x6CFBCDEB,
0xA53FCB7C, 0x7514881B, 0x48694DB2, 0x98420ED5,
0x32938AAD, 0xE2B8C9CA, 0xDFC50C63, 0x0FEE4F04,
0xC154926B, 0x117FD10C, 0x2C0214A5, 0xFC2957C2,
0x56F8D3BA, 0x86D390DD, 0xBBAE5574, 0x6B851613,
0xA2411084, 0x726A53E3, 0x4F17964A, 0x9F3CD52D,
0x35ED5155, 0xE5C61232, 0xD8BBD79B, 0x089094FC,
0x077EDBF8, 0xD755989F, 0xEA285D36, 0x3A031E51,
0x90D29A29, 0x40F9D94E, 0x7D841CE7, 0xADAF5F80,
0x646B5917, 0xB4401A70, 0x893DDFD9, 0x59169CBE,
0xF3C718C6, 0x23EC5BA1, 0x1E919E08, 0xCEBADD6F,
0xCFA869D6, 0x1F832AB1, 0x22FEEF18, 0xF2D5AC7F,
0x58042807, 0x882F6B60, 0xB552AEC9, 0x6579EDAE,
0xACBDEB39, 0x7C96A85E, 0x41EB6DF7, 0x91C02E90,
0x3B11AAE8, 0xEB3AE98F, 0xD6472C26, 0x066C6F41,
0x09822045, 0xD9A96322, 0xE4D4A68B, 0x34FFE5EC,
0x9E2E6194, 0x4E0522F3, 0x7378E75A, 0xA353A43D,
0x6A97A2AA, 0xBABCE1CD, 0x87C12464, 0x57EA6703,
0xFD3BE37B, 0x2D10A01C, 0x106D65B5, 0xC04626D2,
0x0EFCFBBD, 0xDED7B8DA, 0xE3AA7D73, 0x33813E14,
0x9950BA6C, 0x497BF90B, 0x74063CA2, 0xA42D7FC5,
0x6DE97952, 0xBDC23A35, 0x80BFFF9C, 0x5094BCFB,
0xFA453883, 0x2A6E7BE4, 0x1713BE4D, 0xC738FD2A,
0xC8D6B22E, 0x18FDF149, 0x258034E0, 0xF5AB7787,
0x5F7AF3FF, 0x8F51B098, 0xB22C7531, 0x62073656,
0xABC330C1, 0x7BE873A6, 0x4695B60F, 0x96BEF568,
0x3C6F7110, 0xEC443277, 0xD139F7DE, 0x0112B4B9,
0xD31DD2E1, 0x03369186, 0x3E4B542F, 0xEE601748,
0x44B19330, 0x949AD057, 0xA9E715FE, 0x79CC5699,
0xB008500E, 0x60231369, 0x5D5ED6C0, 0x8D7595A7,
0x27A411DF, 0xF78F52B8, 0xCAF29711, 0x1AD9D476,
0x15379B72, 0xC51CD815, 0xF8611DBC, 0x284A5EDB,
0x829BDAA3, 0x52B099C4, 0x6FCD5C6D, 0xBFE61F0A,
0x7622199D, 0xA6095AFA, 0x9B749F53, 0x4B5FDC34,
0xE18E584C, 0x31A51B2B, 0x0CD8DE82, 0xDCF39DE5,
0x1249408A, 0xC26203ED, 0xFF1FC644, 0x2F348523,
0x85E5015B, 0x55CE423C, 0x68B38795, 0xB898C4F2,
0x715CC265, 0xA1778102, 0x9C0A44AB, 0x4C2107CC,
0xE6F083B4, 0x36DBC0D3, 0x0BA6057A, 0xDB8D461D,
0xD4630919, 0x04484A7E, 0x39358FD7, 0xE91ECCB0,
0x43CF48C8, 0x93E40BAF, 0xAE99CE06, 0x7EB28D61,
0xB7768BF6, 0x675DC891, 0x5A200D38, 0x8A0B4E5F,
0x20DACA27, 0xF0F18940, 0xCD8C4CE9, 0x1DA70F8E,
0x1CB5BB37, 0xCC9EF850, 0xF1E33DF9, 0x21C87E9E,
0x8B19FAE6, 0x5B32B981, 0x664F7C28, 0xB6643F4F,
0x7FA039D8, 0xAF8B7ABF, 0x92F6BF16, 0x42DDFC71,
0xE80C7809, 0x38273B6E, 0x055AFEC7, 0xD571BDA0,
0xDA9FF2A4, 0x0AB4B1C3, 0x37C9746A, 0xE7E2370D,
0x4D33B375, 0x9D18F012, 0xA06535BB, 0x704E76DC,
```

```
    0xB98A704B, 0x69A1332C, 0x54DCF685, 0x84F7B5E2,
    0x2E26319A, 0xFE0D72FD, 0xC370B754, 0x135BF433,
    0xDDE1295C, 0x0DCA6A3B, 0x30B7AF92, 0xE09CECF5,
    0x4A4D688D, 0x9A662BEA, 0xA71BEE43, 0x7730AD24,
    0xBEF4ABB3, 0x6EDFE8D4, 0x53A22D7D, 0x83896E1A,
    0x2958EA62, 0xF973A905, 0xC40E6CAC, 0x14252FCB,
    0x1BCB60CF, 0xCBE023A8, 0xF69DE601, 0x26B6A566,
    0x8C67211E, 0x5C4C6279, 0x6131A7D0, 0xB11AE4B7,
    0x78DEE220, 0xA8F5A147, 0x958864EE, 0x45A32789,
    0xEF72A3F1, 0x3F59E096, 0x0224253F, 0xD20F6658,
};
```