Primitive Specification for SSS

Philip Hawkes, Michael Paddon, Gregory G. Rose, Miriam Wiggers de Vries

{phawkes,mwp,ggr,miriamw}@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 3
2 Description of SSS 3
2.1 Introduction 3
2.1.1 History: The SOBER Family of Stream Ciphers 3
2.1.2 Usage and threat model 3
2.2 Formal declarations 4
2.3 Outline of this Document 4
2.4 Notation and conventions 4
3 Description 5
3.1 Keystream generation 5
3.2 The MAC accumulator 6
3.3 The S-Box Function *f* 8
3.4 Keystream generation 10
3.5 The Nonce Loading 11
4 Security Analysis of SSS 11
4.1 Security Requirements 11
4.2 Security Claims 13
4.3 Heuristic Analysis of SSS 14
4.3.1 Analysis of the Key 14
4.3.2 Analysis of the stream cipher component. 14
4.3.3 Analysis of the MAC function 15
5 Strengths and Advantages of SSS 15
6 Performance 16
7 References 17
8 Appendix: Recommended C-language interface 18
9 Appendix: The Qbox and Skipjack F-table 19
10 Appendix: The Multiplication Table 21

1 Justification

SSS is a self-synchronizing stream cipher with message authentication functionality, submitted to the ECrypt NoE call for stream cipher primitives, profile 1A.

SSS stands for Self-Synchronizing SOBER, and is a member of the SOBER family of stream ciphers [6, 9, 10, 14, 15].

2 Description of SSS

2.1 Introduction

SSS is a self-synchronizing stream cipher designed for a secret key that may be up to 128 bits in length. The cipher outputs the key stream in 16-bit blocks. SSS is a software-oriented cipher based on simple 16-bit operations (such as 16-bit XOR and addition modulo 2^{16}), and references to small fixed arrays. Consequently, SSS is at home in many computing environments, from smart cards to large computers. Source code for SSS is freely available and use of this source code, or independent implementations, is allowed free for any purpose.

It is not obvious why a self-synchronizing cipher should have message integrity functionality, since the main purpose of it would seem to be recovery from data corruption, insertion, or deletion in continuous operation, but there may be some advantage to using the same cipher in multiple contexts. Accordingly, SSS also supports a message-based model, with nonces and integrity protection integrated.

2.1.1 History: The SOBER Family of Stream Ciphers

SSS was developed from SOBER [14, 15], proposed by Rose in 1998, and subsequent developments [6, 9, 10]. The algorithm for SSS is based on 16-bit operations, versus the 8-bit operations used in the first SOBER. SSS inherits the shape of the state register (in terms of nonlinear function taps and feedback taps), and some of the techniques used. SSS, being self-synchronizing, is a significant departure from the earlier members of the family which were all synchronous.

2.1.2 Usage and threat model

SSS offers message encryption or message integrity protection or both. In some applications, where it is desirable to provide message integrity for the whole message, and privacy (encryption) for all or part of the message, SSS also supports this model of use.

SSS includes a facility for simple re-synchronisation without the sender and receiver establishing new secret keys through the use of a nonce (a number used only once).

This facility does not always need to be used. For example, SSS may be used to generate a single encryption keystream of arbitrary length. In this mode it would be possible to use SSS as a replacement for the commonly deployed RC4 cipher in, for example, SSL/TLS. In this mode, no nonce is necessary.

In practice though, much communication is done in messages where multiple encryption keystreams are required and the integrity of individual messages needs to be ensured. NLS achieves this using a single secret key for the entire (multi-message) communication, with a nonce distinguishing individual messages. Section 9 below describes the recommended interface.

SSS is intended to provide security under the condition that no nonce is ever reused with a single key, and that no more than 2^{128} words of data are processed with one key. There is no requirement that nonces be random; this allows use of a counter, and makes guaranteeing uniqueness much easier.

2.2 Formal declarations

The designers state that we have not inserted any deliberate weaknesses, nor are we aware (at the time of writing) of any deficiencies of the primitive that would make it unsuitable for the ECrypt Call for Stream Cipher Primitives.

QUALCOMM Incorporated allows free and unrestricted use of any of its intellectual property required to exercise the primitive, including use of the provided source code. The designers are unaware of any intellectual property owned by other parties that would impact on the use of SSS.

2.3 Outline of this Document

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

2.4 Notation and conventions

a<<<b(resp. >>>) means rotation of the word a left (respectively right) by b bits

$\oplus$, $\otimes$ are addition and multiplication operations in the specified Galois field; as such, $\oplus$ is simply exclusive-or of words.

^ is bit-wise “and” of 16-bit words.

~ is bit-wise complement of a 16-bit word.

+ is addition modulo 2^{16}.

The most significant 8 bits of 16-bit word a is denoted a_H.

SSS is entirely based on 16-bit word operations internally, but the external interface is specified in terms of arrays of bytes. Conversion between 2-byte chunks and 16-bit words is done in "little-endian" fashion irrespective of the byte ordering of the underlying machine.

The terms *ciphertext* and *plaintext* are used with their normal meanings. For the purposes of message integrity where ciphertext and plaintext are mixed in a single message ("authenticated encryption with associated data" (*AEAD*) is a recent term for this), we will also refer to *messagetext*, which is whatever is transmitted and is hence assumed vulnerable to eavesdropping. Conversely, we will refer to *mactext*, which is the ciphertext that corresponds to transmitted plaintext, or the plaintext that corresponds to transmitted ciphertext. SSS calculates its message authentication code always with the mactext as input.

3 Description

3.1 Keystream generation

SSS' stream generator is constructed from a *simple shift register* and a *non-linear filter*. The primitive is based on 16-bit operations and 16-bit blocks: each 16-bit block is called a *word*. The vector of words $\sigma_t = (r_t[0], \ldots, r_t[16])$ is known as the *state* of the register at time t, and the state $\sigma_0 = (r_0[0], \ldots, r_0[16])$ is called the *initial state* after processing a nonce.

The key in SSS determines the action of a highly nonlinear transformation function f, described in more detail below. Depending on implementation, the key loading process may be used to precalculate this function as an 8×16 key-dependent S-box.

The state transition function transforms state σ_t and the current ciphertext word c_t into state σ_{t+1} in the following manner:

1. $r_t[0]$ is abandoned
2. $r_{t+1}[i] = r_t[i+1]$, for $i = 0..15$
3. $r_{t+1}[16] = c_t$
4. $r_{t+1}[14] = r_{t+1}[14] + f(c_t >>> 8)$
5. $r_{t+1}[12] = f(r_{t+1}[12])$
6. $r_{t+1}[1] = r_{t+1}[1] >>> 8$

The nonlinear function f is defined in section 3.3 below. Successive states σ_t from the register are fed through the filter to produce 16-bit *keystream words* denoted v_t. These combine to form the *keystream* $\{v_t\}$. Each keystream word v_t is obtained as

$$v_t = NLF(\sigma_t) = f(f(r[0] + r[16]\ ) + r[1] + r[6] + r[13]) >>> 8) \oplus r[0].$$

As usual, ciphertext $c_t = p_t \oplus v_t$ and plaintext $p_t = c_t \oplus v_t$.

SSS allows for any portion of the plaintext to be encrypted (or not) when forming the transmission message. When the sender forms the transmission message, the bits that contain encrypted plaintext are formed by XORing the corresponding plaintext bits with the corresponding keystream bits. Similarly, when the receiver extracts the plaintext from the transmission message, the bits that contain encrypted ciphertext are decrypted to the plaintext by XORing the corresponding transmission message bits with the corresponding keystream bits. Note that the keystream is generated, and ciphertext is fed back into the register, independently of whether what was transmitted was ciphertext or plaintext.

SSS allows for encryption/decryption and authentication of plaintext of any length, but most of the operations are designed to act on 16-bit blocks of plaintext or transmission message. Section 3.2 describes how SSS operates when the remaining plaintext (or transmission message) does not form a full 16-bit word.

3.2 The MAC accumulator

The MAC accumulator in SSS is simply a 17-word (272-bit) CRC, $\mu_t = (c_t[0], \ldots, c_t[16])$ of the *mactext* (see definition above), that is, the text that has not been seen by the attacker.

Padding: Suppose that the length of the message, M, is l bits. Append k zero bits of plaintext (that is, assume these bytes are transmitted as plaintext), where k is the smallest, non-negative solution to the equation $l+k \equiv 0 \pmod{16}$. The length of the padded message should now be a multiple of 16 bits.

Parsing: The padded message is parsed into a sequence of 16-bit mactext words, $\{M_t\}$.

Initialization: μ_0 is initialized from the nonce as described below.

CRC register update: The CRC is a 272-bit cyclic redundancy checksum of the message words, calculated over GF(2^{272}). This CRC is calculated word-at-a-time, using a seventeen word LFSR defined over GF(2^{16}). 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 16-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 ($a_{w-1}, a_{w-2}, \ldots, a_1, a_0$), which represents the polynomial $a_{w-1}z^{w-1}+\ldots+a_1z+a_0$. 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^{16}) as degree-15 polynomials over GF(2), Mundja uses 8-bit octets to represent elements of a subfield GF(2^8), and 16-bit words to represent degree-1 polynomials over GF(2^8). This is isomorphic to the standard representation, but not identical. The subfield B = GF(2^8) of octets is represented modulo the irreducible polynomial $z^8 + z^6 + z^3 + z^2 + 1$. Octets represent degree-7 polynomials over GF(2); the constant β_1 = 0x50 below represents the polynomial $z^6 + z^4$ for example. The Galois finite field $W = B^2$ = GF((2^8)^2) of words can now be represented using degree-1 polynomials where the coefficients are octets (subfield elements of B). For example, the word 0x500F represents the polynomial 0x50.y+0x0F. The field W can be represented modulo an irreducible polynomial $y^2 + \beta_1 y + \beta_0$. For SSS, β_1 = 0x50, and β_0 = 0x0F.

The CRC polynomial defined over GF(2^{16}), then, is $x^{17} + x^{15} + x^4 + \alpha$. The coefficient α = 0x0100 = 0x01y + 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 of the multiplicand, shifting the input word to the left by 8 bits, and then adding (XORing) the resulting words together. This is essentially a 16-bit version of the field multiplication used in SNOW [4], Turing [9] and SOBER-128 [10]. The `Multab` for SSS is shown in Appendix B. In C code, the new word to be inserted in the LFSR, based on the previous state of the LFSR and the input word `w` is calculated:

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

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^2 + 0x50·y + 0x0F; with these definitions, the LFSR without external input would generate an m-sequence.

CRC Register Update: Following initialization, the CRC update function is also applied to accumulate the content of the mactext sequence $\{M_t\}$. The t-th input word is also input to the CRC register according to the following algorithm:

1. for j = 0..15: $c_{t+1}[j] = c_t[j + 1]$;

2. $c_{t+1}[16] = M_t \oplus (\alpha \otimes c_t[0]) \oplus c_t[4] \oplus c_t[15]$.

Finalization and generation of MAC: When all the words of the input message have been processed (including a final padded word, if any), special finalization begins. The CRC register is not changed any more during this process. First, a word consisting of the value (k>>>3) (k is the number of bits of padding added) is encrypted and the ciphertext is fed back into the stream cipher register as above; however this word is not input to the CRC register. Adding this word to one register but not the other

desynchronizes them in a manner that cannot be emulated by normal inputs, preventing extension attacks.

Now data from the CRC register is transferred into the stream cipher register similarly, simply by treating words as plaintext to be encrypted, and allowing the normal ciphertext feedback mechanism to process them as above. The CRC words are processed in reverse order, so that the ones that have had least influence on the value of the CRC have the most influence on the state of the stream cipher. That is:

1. for j = 16 **downto** 0: (process as in 3.1 above)
    a. encrypt c[j]
    b. feed back ciphertext

Finally, to generate the MAC of length l_m, this process is continued, encrypting plaintext words of 0, and using the corresponding ciphertext words as the MAC:

2. for $j = 1 .. l_m$: (process as in 3.1 above)
    a. encrypt a word whose plaintext value is *0*
    b. feed back ciphertext
    c. output ciphertext as MAC

As usual, the MAC words are converted to bytes in little-endian fashion.

3.3 The S-Box Function *f*

The nonlinear function f is most efficiently realised as an 8×16-bit S-box. The function is defined as $f(a) = \text{SBox}[a_H] \oplus a$. The function is key-dependent, and presumed unknown to the attacker; in fact, this is the only key-dependent component of SSS, so in many ways this table could be considered to be the key. Note: there are many more than 2^{128} possible S-boxes that would be compatible with the properties required by SSS, but the way in which the S-box is calculated would allow the effect of some key bytes to cancel out if keys are longer than 128 bits. Except for this, we believe the strength of SSS to be somewhat more than 128-bit-equivalent; longer keys could be supported by a variant that uses a different method of computing this S-box. The S-box is constructed from two fixed tables in the manner previously used by Turing [9].

The S-box is a combination of the Skipjack [5] S-box (called "F-table" in the definition of Skipjack, abbreviated *ft* here) and an S-box designed at our request by the Information Security Research Centre (ISRC) at the Queensland University of Technology [3], called the *Qbox* here.

The Skipjack S-box is a highly nonlinear permutation of 8-bit inputs without any known algebraic structure.

The Qbox is a fixed nonlinear 8→16-bit table. It was developed by the Queensland University of Technology at our request [3]. It is best viewed as 16 independent Boolean functions of the 8 input bits. The criteria for its development were:

- the functions should be highly nonlinear (each has nonlinearity of 114)
- the functions should be balanced
- the functions should be pairwise uncorrelated

Let the key be K_i ($i = 0..N-1$) (where N is the key length in bytes). The input byte x is combined with a key byte and passed through the Sbox, the result is combined with another key byte, and so on, to form a temporary result.

$$t = t(x) = \mathrm{ft}[K_{N-1} \oplus \mathrm{ft}[K_{N-2} \oplus \ldots \mathrm{ft}[K_0 \oplus x]\ldots]]$$

Note that *t(x)* forms a permutation. This process can be visualised as the input byte "bouncing around" the Skipjack permutation table under the control of the key.

At each bounce, a rotated word from the Qbox is accumulated into another temporary word *w*; the rotation depends on the stage of progress, ensuring that no entries of the Qbox can cancel each other out. (Note: the 16-bit width of the Qbox used determines the 16-byte limit on the key length.) The accumulated word *w* is highly nonlinear with respect to the input, and highly dependent on the key material, however the bit positions in it are not likely to be balanced. The byte *t*, being a permutation, is by definition balanced. So replacing the most significant byte of *w* with *t* forms the final word for this input byte.

The following C code illustrates the result of applying the keyed transformation to an input word *w*:

```
WORD
Sbox(UCHAR *key, int keylen, WORD w)
{
    register int    i;
    WORD            t;
    UCHAR           b;

    t = 0;
    b = HIGHBYTE(w);
    for (i = 0; i < keylen; ++i) {
       b = ftable[b ^ key[i]];
       t ^= ROTL(Qbox[b], i);
    }
    return ((b << (WORDBITS-8)) | (t & LOWMASK)) ^ (w & LOWMASK);
}
```

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 non-linearity from the high bits of its input to the low bits of its output.

An S-box can easily be precomputed. When the S-box is precomputed, for efficiency, it is desirable to use a single XOR operation to effect the S-box operation, so the most significant byte of the stored S-box is XORed with the input byte.

```
for (i = 0; i < 256; ++i)
       S[i] = Sbox(key, keylen, (WORD)(i << 8)) ^ (i << 8));
```

Then the effect of the Sbox function is achieved with a single XOR operation and lookup in the table:

```
w ^= S[(w >> 8) & 0xFF];
```

3.4 Keystream generation

SSS supports Authenticated Encryption with Associated Data (AEAD, although we prefer the term Partial Encryption with Message Integrity). When the sender forms the transmission message, the bits that contain encrypted plaintext are formed by XORing the plaintext bits with the corresponding keystream bits. Similarly, when the receiver extracts the plaintext from the transmission message, the bits that contain encrypted plaintext are decrypted to the plaintext by XORing the corresponding transmission message bits with the corresponding keystream bits. "Associated data", that is data that it meant to be authenticated but not encrypted, is handled by simply failing to XOR the corresponding keystream into those bits. The keystream is generated anyway, since ciphertext is incorporated into the CRC register whenever plaintext is transmitted.

Keystream Mask: One method of implementing this is to form a *keystream mask* KM_t for each word position, where KM_t has ones in those bit positions that will be encrypted in the transmission message and zeroes in those bits positions that will not be encrypted in the transmission message. A messagetext word m_t is formed from the corresponding plaintext word p_t, keystream word v_t and keystream mask KM_t as:

$$m_t = p_t \oplus (KM_t \wedge v_t),$$

where "^" denotes the bit-wise AND operation. Likewise, a plaintext word is formed from the corresponding transmission message word, keystream word and keystream mask word as:

$$p_t = m_t \oplus (KM_t \wedge v_t).$$

Conversely, the mactext mac_t (input to the CRC register) is formed the same way at the transmission end, but with the complement of the mask:

$$mac_t = p_t \oplus (\sim KM_t \wedge v_t),$$

The receiver can form the mactext word mac_t either by first recovering the plaintext, or directly from the messagetext:

$mac_t = m_t \oplus v_t$.

Because of the self-synchronizing behaviour of SSS, the transmitter and receiver need to implicitly agree on the keystream mask; even with the same input bytes, the generated MAC will be different depending on exactly which parts of the message were encrypted and which parts were only "associated data".

3.5 The Nonce Loading

SSS is re-initialized with a nonce by setting the two registers to a known state, then using standard operations of the cipher to incorporate and diffuse the effect of the nonce. A nonce can be any sequence of bytes that is a multiple of the 16-bit word size (as always, bytes are interpreted in little-endian fashion to form words).

1. for i in 0..16: set $r[i] = c[i] = 0$.[1]

2. treat the nonce as received ciphertext, decrypting it and accumulating the plaintext into the CRC register.

3. process 17 words of plaintext zeros, as if they had been received as associated data. Since ciphertext is being calculated and fed back into both the cipher and CRC registers, this quite effectively diffuses the nonce.

4 Security Analysis of SSS

Some of the components of SSS have been subjected to scrutiny when they appeared in earlier members of the SOBER family of stream ciphers.

4.1 Security Requirements

SSS is intended to provide 128-bit security. The base attack on SSS is an exhaustive key search, which has a computational complexity equivalent to generating 2^{128} keystream words[2]. 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 nonces. SSS is considered to *resist* an attack if either the attack requires the owner of the secret key(s) to generate more than 2^{80} keystream 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 SSS fulfils the following security requirements, when used subject to the condition that no key/nonce pair is ever reused, no

[1] There is no cryptographic significance to the use of zero here. This provides an opportunity for "tweaking" the cipher, as any different initialization value for the stream cipher register will yield a completely different cipher with identical security properties.
[2] Unless, of course, a shorter secret key is used. We assume use of a 128-bit or longer secret key in this section.

more than 2^{80} words are processed with one key, and the result of decrypting altered ciphertext is not made available to the attacker[3]:

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

2. **Keystream Recovery Attacks**: SSS 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**: SSS should resist attacks that distinguish a SSS keystream from random bit stream.

4. **Related-Key or related nonce Attacks**: SSS 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.

A separate set of security requirements apply to the MAC functionality. First, we consider the security properties required of a MAC function. A MAC function is a cryptographic algorithm that generates a tag TAG = $MAC_K(M)$ of length *d* from a message M of arbitrary length and a secret key K of length *n*. The message-tag pair (M,TAG) is transmitted to the receiver (the message may be encrypted, in whole or in part, before transmission).

Suppose the received message-tag pair is (RM,RTAG). The receiver calculates an expected tag XTAG = $MAC_K(RM)$. If XTAG = RTAG, then the receiver has some confidence that the message-tag pair was formed by a party that knows the key K. In some cases, the message includes sequence data (such as a nonce) to prevent replay of message-tag pairs.

The length *n* of the key and the length *d* of the tag form the security parameters of a MAC algorithm, as these values dictate the degree to which the receiver can have confidence that the message-tag pair was formed by a party that knows the key K. A MAC function with security parameters (*n*, *d*) should provide resistance to four classes of attacks:

1. **Collision Attack**. In a collision attack, the attacker finds any two distinct messages M,M' such that $MAC_K(M) = MAC_K(M')$. A MAC function resists a collision attack if the complexity of the attack is $O(2^{\min(n,d/2)})$.

2. **First Pre-image Attack**. In a first pre-image attack, the attacker is specified a tag value TAG, and the attacker must find a message M for which $MAC_K(M) = TAG$. A MAC function resists a first pre-image attack if the complexity of the attack is $O(2^{\min(n,d)})$.

[3] This should be a standard requirement for any self-synchronizing stream cipher, since the attacker has complete control over the state of the cipher.

3. **Second pre-image attack**. In a second pre-image attack, the attacker is specified a message M, and the attacker generates a new message M' such that $MAC_K(M) = MAC_K(M')$. A MAC function resists a second pre-image attack if the complexity of the attack is $O(2^{\min(n,d)})$.

4. **MAC Forgery**. In MAC forgery, the attacker generates a new message-tag pair (M', y') such that $y' = MAC_K(M')$. A MAC function resists MAC forgery if the complexity of the attack is $O(2^{\min(n,d)})$.

In all these attacks, the attacker is presumed to be ignorant of the value of the key K[4]. However, we assume that (prior to the attack) the attacker can specify messages M(i) for which they will be provided with the corresponding tags $TAG(i) = MAC_K(M(i))$.

SSS is intended to be a MAC function with security parameters n = 128, and $d \leq 128$. That is, we claim that SSS resists the above attacks when using 128-bit keys and outputting tags up to 128 bits in length.

SSS will be considered broken if an attacker can perform any of these attacks.

A comment on distinguishing attacks. There is a trivial active distinguishing attack against any self-synchronizing cipher; after the attacker inserts or replaces as much ciphertext as forms the state of the cipher, the next unit of keystream output will be the same. Assuming known plaintext and revelation of the receiver's idea of the received data, distinguishing the cipher from random means simply verifying that the keystream was the same at this point. We assume that such distinguishing attacks are not part of the threat model being evaluated. If the receiver always discards any message that fails authentication, and the MAC tag is long enough, such an attack will be frustrated anyway.

4.2 Security Claims

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

- Algebraic attacks [1,2] appear to be infeasible, based on the fact that the S-box is both unknown and of high algebraic complexity.

- All output words, whether keystream or MAC, are formed from words that have been passed through the secret S-Box operation multiple times.

- The cipher register structure (transformation points and filter function taps) were chosen to ensure that it is difficult to isolate the effect of any particular word or combination of words. This is analogous to resistance against Guess-and-Determine attack resistance in the more traditional members of the SOBER family.

[4] There are circumstances (albeit rare) where the users require a MAC function to resist a collision attack with known key. SSS is not intended to prevent this type of attack. We note that other common MAC constructions, such as CBC-MAC [11], cannot prevent this type of attack either.

- Timing attacks and power attacks can be mitigated in standard ways; there is no data-dependent conditional execution after initial keying, nor are the operations used data-dependent in execution time on most CPUs.

- We are unaware of any weak keys or weak-key classes.

- Being self-synchronizing, it is not particularly meaningful to speak of a cycle length for SSS. However, if the plaintext is all zeros, for example, the ciphertext might exhibit cyclic structure. We have been unable to detect, or predict, any such cycles.

4.3 Heuristic Analysis of SSS

This analysis concentrates on vulnerability of SSS 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. Any unknown-plaintext attack would also be manifested as a serious distinguishing attack, so we don't consider this any further.

Almost all of the features of SSS have been taken from previous members of the SOBER family (including Turing). Some previous analysis can be directly applied. For example, Guess-and-Determine attacks have been evaluated in detail over several years, and are known to have complexity far greater than the key size; this has bearing on forming equations to attempt to recover the secret S-box. The tables from which the secret S-box is built have been evaluated in detail.

4.3.1 Analysis of the Key

The key determines the secret *f* function of SSS. The process was designed to ensure that the following properties of the function *f* described hold:

- The high order byte of the output is a permutation of the high order byte of the input.

- Every byte of the key influences all bits of the function.

- No two distinct secret keys (up to 128-bits in length) can result in the same *f* function.

4.3.2 Analysis of the stream cipher component.

The "shape" of the state register, in the sense of its modification taps and nonlinear filter function taps, is the same as has been used previously. The positions of these taps were mechanically optimized to give maximum resistance to Guess and Determine attacks in SOBER [8], which appear to have complexity greater than 2^{128}. In SSS, this property translates to making it difficult to form equations with a small number of inputs that would allow analysis of the *f* function.

There is no explicit feedback in SSS, because it is self-synchronizing. However, the fact that ciphertext is fed into the state register effectively gives such feedback, so much of the analysis applies. The

feedback function of the stream cipher is highly nonlinear, through use of the secret *f* function. The nonlinear effects compound quite rapidly. Since the register operates on nonlinear feedback, little can be proven about its cycle length. In the absence of any reason to believe that the feedback function behaves in a significantly non-random fashion, the average cycle length when encrypting constant data is assumed to be approximately 2^{270}.

The nonlinearity of the state update and filter functions, and the selection of the taps for the output filter function, should adequately disguise any short-distance correlations.

The output filter function is designed to ensure that all inputs that affect any particular output have been through the *f* function multiple times, and in particular, all bits have appeared in the most significant byte position at least once.

4.3.3 Analysis of the MAC function

The strength of SSS' MAC function relies upon the fact that the CRC register accumulates the mactext, which the attacker cannot predict. A single-bit change to the messagetext results in changes to nine words of subsequent mactext, of which six are effectively unpredictable (for a 96-bit "penalty"). To attempt to then cancel out such a change only introduces more change. The finalization processing both encrypts the contents of the CRC register, and uses it to alter the nonlinear state.

5 Strengths and Advantages of SSS

SSS has the following strengths and advantages.

- Speed: SSS is reasonably fast in software implementation.
- Tradeoffs between key agility and throughput are possible.
- 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 nonce.
- Appears to provide more than adequate security.
- Incorporates MAC functionality

6 Performance

The fundamental building block of SSS is the key-dependent *f* function. There are significant implementation choices to be made here, as we have identified at least four ways to perform this function:

1. Embedded applications might ignore the "key" entirely, and simply burn the S-box into ROM in the device.

2. Servers that require high key agility and a small memory usage per key might simply accept a performance penalty to evaluate the *f* function directly from the key every time.

3. The S-box can be built from the key at a key setup phase.

4. Midway between the second and third approaches above, a form of "lazy evaluation" can be used; when an S-box entry has not been filled in yet, the result of the *f* function can be evaluated and saved; this approach works well for very small amounts of data to be processed per key, but introduces the possibility of timing attacks.

Table 1 shows the cost of precomputing the S-box from a 128-bit key; there is effectively zero cost at key setup time to dynamically evaluating the *f* function. Table 2 contains performance figures for SSS, using a precomputed S-box. The figures are for optimized C-language code and were obtained on a 1.5GHz Intel Centrino running Cygwin over Windows 2000 and compiled with gcc 3.3.1 (with "-O3" optimization setting). Note that we believe these figures can be significantly improved. Assembler code can be expected to perform much better. Keys and nonces are each 128 bits. These figures are for the "sssfast.c" source code provided herewith.

MKeys/s	cycles/key	M Nonces/s	cycles/Nor
0.033009	45442	0.740741	2024.999

Table 1. Computation required for key loading and nonce loading of SSS. These results were obtained by averaging the measurements for a large number of keys.

Length	Operation	Mbyte/second	cycles/byte
Continuous	Stream encryption	87.72581	17.1
Continuous	Stream decryption	63.55006	23.6
1600-byte blocks	Stream encryption (nonce but no MAC)	86.58009	17.325
1600-byte blocks	Authentication only	64.51613	23.25
1600-byte blocks	Encryption + MAC generation	60.42296	24.825
1600-byte blocks	Decryption + MAC generation	68.96552	21.75

Table 2. Performance figures for encryption and message authentication. The block figures include the time for nonce loading and MAC finalization.

- A minimal memory implementation of the stream cipher alone requires 34 bytes of RAM, accounting for the cipher register. The S-box would occupy 512 bytes of ROM (precalculated). Message authentication functionality adds a further 34 bytes, for the CRC register, and another 512 bytes of ROM for the CRC multiplication table.

- An implementation that evaluates the *f* function on the fly from a key would require 256 bytes for the Skipjack F-table, and 512 bytes for the Qbox, all in ROM.

- An implementation that calculates an S-box from these tables, or uses lazy evaluation, would require a further 512 bytes of RAM for the S-box itself.

SSS uses only simple word-oriented instructions, such as addition, XOR, shift (by a constant number of bits) and table lookups. Instructions that commonly take a variable amount of time, such as data-dependent shifts, or which are difficult to implement in hardware or often not implemented on low-end microprocessors, such as integer multiplication, have been avoided.

7 References

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

2. 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.

3. 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.

4. P. Ekdahl, T. Johansson, SNOW - a new stream cipher, Proceedings of First Open NESSIE Workshop, KU-Leuven, 2000. See [13].

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

6. P. Hawkes and G. Rose. The t-class of SOBER stream ciphers. Technical report, QUALCOMM Australia, 1999. See `http://www.qualcomm.com.au.`

7. 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

8. 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.

9. P. Hawkes and G. Rose. "Turing, a Fast Stream Cipher". In T. Johansson, Proceedings of *Fast Software Encryption FSE2003,* LNCS 2887, Springer-Verlag 2003.

10. P. Hawkes, G. Rose. *Primitive Specification for SOBER-128,* 2003. See eprint.iacr.org/2003/081.pdf.

11. International Organization for Standardization, Data Cryptographic Techniques-Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher Algorithm, ISO/IEC 9797, 1989.

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

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

14. 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.

15. G. Rose, "SOBER: A Stream Cipher based on Linear Feedback over $GF(2^8)$". Unpublished report, QUALCOMM Australia, 1998. See `http://www.qualcomm.com.au`.

8 Appendix: Recommended C-language interface

The following definitions are used by our reference code, and implement a byte-oriented interface to SSS.

```
typedef struct {
    WORD     R[N];         /* Working storage for the shift register */
    WORD     CRC[N];       /* working storage for the CRC register */
    WORD     SBox[256];    /* key dependent SBox */
    UCHAR    key[MAXKEY];  /* copy of key */
```

```
    int      keylen;      /* length of key in bytes */
    WORD     sbuf;        /* partial word stream buffer */
    WORD     cbuf;        /* partial word ciphertext buffer */
    WORD     mbuf;        /* partial word MAC input buffer */
    int      nbuf;        /* number of part-word stream bits buffered */
} sss_ctx;
void sss_key(sss_ctx *c, UCHAR key[], int keylen);   /* set key */
void sss_nonce(sss_ctx *c, UCHAR nonce[], int nlen); /* set nonce */
void sss_enconly(sss_ctx *c, UCHAR *buf, int nbytes);/* stream encryption */
void sss_deconly(sss_ctx *c, UCHAR *buf, int nbytes);/* stream decryption */
void sss_maconly(sss_ctx *c, UCHAR *buf, int nbytes);/* accumulate MAC */
void sss_encrypt(sss_ctx *c, UCHAR *buf, int nbytes);/* encrypt + MAC */
void sss_decrypt(sss_ctx *c, UCHAR *buf, int nbytes);/* decrypt + MAC */
void sss_finish(sss_ctx *c, UCHAR *buf, int nbytes); /* finalise MAC */
```

For completely synchronous operation as a basic stream cipher, it suffices to call *key*, then *stream* as required (an implicit zero-length nonce is provided). For all other operations, the communication should be broken into messages, and *nonce* should be called at the beginning of each message. Nonces should never be reused, but nonces are otherwise opaque to the system, and could easily be based on counters, timestamps, or whatever.

Our reference implementation for these primitives provides a byte-wise interface.

9 Appendix: The Qbox and Skipjack F-table

```
WORD Qbox[256] = {
    0x1887,   0x435c,   0xc042,   0x6ef4,
    0xee20,   0xfed3,   0xc502,   0xe8ae,
    0xe9d9,   0x38d4,   0x9b5d,   0xdf3c,
    0x4249,   0x3963,   0x429f,   0x2c35,
    0x0325,   0xdd70,   0x3ded,   0xdc5e,
    0x5b42,   0x12bf,   0xd78c,   0xb26b,
    0x1b9a,   0x8146,   0x8ec5,   0xc28f,
    0x5c0f,   0x101c,   0xb082,   0x29e1,
    0x43de,   0x99fc,   0xbc4b,   0x15dd,
    0x03fa,   0xb2de,   0x3342,   0xe7c3,
    0x07ef,   0xebab,   0x859b,   0x2e2f,
    0x71da,   0x269a,   0xc3d1,   0x6b36,
    0xdef2,   0xfc5f,   0xb3a3,   0x6ddf,
    0xb510,   0x85a7,   0x2e71,   0x8816,
    0x1e2a,   0xf6af,   0xc2b3,   0xf55d,
    0x6214,   0x83e3,   0xa6f5,   0x41af,
    0x1f17,   0x99ee,   0x5ec0,   0x16c6,
    0x09a4,   0x6e01,   0x80d9,   0x1418,
    0xf227,   0x8203,   0x9d96,   0xa8c0,
    0xbf6e,   0x7888,   0xfe64,   0x93cd,
    0x0184,   0x4930,   0x4f36,   0x7088,
    0x6c2a,   0xc678,   0x4de7,   0xe759,
    0x248e,   0x446b,   0x9fc2,   0xa895,
    0xc3a1,   0xf170,   0x9155,   0x8a66,
    0x5e69,   0x623e,   0xfa35,   0x68cc,
    0x6acd,   0xe936,   0x2db9,   0x13c1,
    0xb16d,   0xb83c,   0x3763,   0xa911,
    0xbc13,   0x79d7,   0x2fa8,   0x196e,
    0x5476,   0xa866,   0x16ad,   0xc515,
    0xeb3c,   0xa306,   0x99d9,   0x9133,
    0x66dd,   0x5dcd,   0x8f50,   0xb226,
    0xcef3,   0x6189,   0x19b1,   0x3084,
    0xed5c,   0xc58f,   0xe421,   0x47fb,
    0x715e,   0xff99,   0x2f0f,   0x5184,
    0x5e6c,   0x18bc,   0xc6e0,   0xe420,
    0x523f,   0xb8a2,   0x1a6b,   0x8c02,
    0xe354,   0x7d79,   0x7753,   0x9655,
```

```
    0x9da1,   0x90a7,   0xc149,   0x7f1c,
    0x9b69,   0xf2b7,   0x58fa,   0x4418,
    0x8c76,   0xd9f0,   0x0d4d,   0xc473,
    0x10e9,   0x4211,   0x082b,   0x334a,
    0x8ed2,   0xcc1b,   0x0ff3,   0x64a0,
    0x5a4f,   0xf8e7,   0xf15f,   0xfe21,
    0x37d6,   0x06f1,   0x0973,   0xde36,
    0x0fa8,   0xab9e,   0xb618,   0x52f5,
    0xeb4f,   0xe343,   0x77dd,   0x3da6,
    0xd52d,   0x12f8,   0x3360,   0x3ad0,
    0x0f1c,   0xed0b,   0xc1ec,   0x6795,
    0x9d15,   0x46d7,   0xbe76,   0xe0a0,
    0x7c02,   0x49b7,   0xd6ba,   0x7f78,
    0xffbd,   0xca84,   0xf4da,   0x35da,
    0xaa44,   0x52ac,   0x74a7,   0xa46a,
    0x152a,   0xb7aa,   0x5927,   0xb118,
    0x758d,   0x687b,   0xf0b3,   0x54ed,
    0x7271,   0xacab,   0x4aec,   0x94cd,
    0x9e81,   0x3730,   0x21e8,   0x7f0b,
    0xb5d6,   0xadf8,   0x0431,   0xc921,
    0x5d46,   0x0a36,   0x4022,   0xa65e,
    0x70ba,   0xa8cc,   0xae8b,   0x24d5,
    0x8a5a,   0x6b81,   0x2522,   0x1cb8,
    0xfe1d,   0xc697,   0x4f83,   0x6376,
    0x224c,   0x3b35,   0xc0fe,   0xa19a,
    0xb24f,   0xa998,   0x2d71,   0x96a8,
    0x053f,   0xd300,   0xcbcc,   0x3d40,
};

/*
 * skipjack ftable
 */
const unsigned char ftable[256] = {
0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,
0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,
0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,
0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,
0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,
0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,
0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,
0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,
0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,
0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,
0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,
0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,
0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,
0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,
0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,
0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,
0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
};
```

10 Appendix: The Multiplication Table

The entries in the multiplication table are given below in hexadecimal form.

```
unsigned short tab500F[256] = {
0x0000, 0x500F, 0xA01E, 0xF011,
0x1033, 0x403C, 0xB02D, 0xE022,
0x2066, 0x7069, 0x8078, 0xD077,
0x3055, 0x605A, 0x904B, 0xC044,
0x40CC, 0x10C3, 0xE0D2, 0xB0DD,
0x50FF, 0x00F0, 0xF0E1, 0xA0EE,
0x60AA, 0x30A5, 0xC0B4, 0x90BB,
0x7099, 0x2096, 0xD087, 0x8088,
0x8198, 0xD197, 0x2186, 0x7189,
0x91AB, 0xC1A4, 0x31B5, 0x61BA,
0xA1FE, 0xF1F1, 0x01E0, 0x51EF,
0xB1CD, 0xE1C2, 0x11D3, 0x41DC,
0xC154, 0x915B, 0x614A, 0x3145,
0xD167, 0x8168, 0x7179, 0x2176,
0xE132, 0xB13D, 0x412C, 0x1123,
0xF101, 0xA10E, 0x511F, 0x0110,
0x533F, 0x0330, 0xF321, 0xA32E,
0x430C, 0x1303, 0xE312, 0xB31D,
0x7359, 0x2356, 0xD347, 0x8348,
0x636A, 0x3365, 0xC374, 0x937B,
0x13F3, 0x43FC, 0xB3ED, 0xE3E2,
0x03C0, 0x53CF, 0xA3DE, 0xF3D1,
0x3395, 0x639A, 0x938B, 0xC384,
0x23A6, 0x73A9, 0x83B8, 0xD3B7,
0xD2A7, 0x82A8, 0x72B9, 0x22B6,
0xC294, 0x929B, 0x628A, 0x3285,
0xF2C1, 0xA2CE, 0x52DF, 0x02D0,
0xE2F2, 0xB2FD, 0x42EC, 0x12E3,
0x926B, 0xC264, 0x3275, 0x627A,
0x8258, 0xD257, 0x2246, 0x7249,
0xB20D, 0xE202, 0x1213, 0x421C,
0xA23E, 0xF231, 0x0220, 0x522F,
0xA67E, 0xF671, 0x0660, 0x566F,
0xB64D, 0xE642, 0x1653, 0x465C,
0x8618, 0xD617, 0x2606, 0x7609,
0x962B, 0xC624, 0x3635, 0x663A,
0xE6B2, 0xB6BD, 0x46AC, 0x16A3,
0xF681, 0xA68E, 0x569F, 0x0690,
0xC6D4, 0x96DB, 0x66CA, 0x36C5,
0xD6E7, 0x86E8, 0x76F9, 0x26F6,
0x27E6, 0x77E9, 0x87F8, 0xD7F7,
0x37D5, 0x67DA, 0x97CB, 0xC7C4,
0x0780, 0x578F, 0xA79E, 0xF791,
0x17B3, 0x47BC, 0xB7AD, 0xE7A2,
0x672A, 0x3725, 0xC734, 0x973B,
0x7719, 0x2716, 0xD707, 0x8708,
0x474C, 0x1743, 0xE752, 0xB75D,
0x577F, 0x0770, 0xF761, 0xA76E,
0xF541, 0xA54E, 0x555F, 0x0550,
0xE572, 0xB57D, 0x456C, 0x1563,
0xD527, 0x8528, 0x7539, 0x2536,
0xC514, 0x951B, 0x650A, 0x3505,
0xB58D, 0xE582, 0x1593, 0x459C,
0xA5BE, 0xF5B1, 0x05A0, 0x55AF,
0x95EB, 0xC5E4, 0x35F5, 0x65FA,
0x85D8, 0xD5D7, 0x25C6, 0x75C9,
0x74D9, 0x24D6, 0xD4C7, 0x84C8,
0x64EA, 0x34E5, 0xC4F4, 0x94FB,
0x54BF, 0x04B0, 0xF4A1, 0xA4AE,
0x448C, 0x1483, 0xE492, 0xB49D,
```

```
0x3415, 0x641A, 0x940B, 0xC404,
0x2426, 0x7429, 0x8438, 0xD437,
0x1473, 0x447C, 0xB46D, 0xE462,
0x0440, 0x544F, 0xA45E, 0xF451,
};
```