

Primitive Specification for NLSv2

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 NLSv2.....	3
2.1	Introduction	3
2.1.1	History: The SOBER Family of Stream Ciphers	3
2.1.2	Usage and threat model.....	4
2.2	Formal declarations	4
2.3	Outline of this Document.....	5
2.4	Notation and conventions	5
3	Description	5
3.1	Keystream generation	5
3.2	The MAC accumulator	6
3.3	The S-Box Function f	9
3.4	Keystream generation	10
3.5	The Key and nonce Loading.....	11
4	Security Analysis of NLSv2.....	13
4.1	Security Requirements.....	13
4.2	Security Claims.....	15
4.3	Heuristic Analysis of NLSv2.....	16
4.3.1	Analysis of the Key Loading	16
4.3.2	Analysis of the stream cipher component.....	16
4.3.3	Analysis of the Mundja MAC function.....	17
5	Strengths and Advantages of NLSv2.....	17
6	Performance.....	18
7	References	19
8	Appendix: Recommended C-language interface	21
9	Appendix: The S-box Entries	22
10	Appendix: The Multiplication Table	24

1 Justification

NLSv2 is a synchronous stream cipher with message authentication functionality, submitted to the ECRYPT NoE call for stream cipher primitives, profile 1A. NLSv2 is an updated version of NLS [23]. The minor change between NLS and NLSv2 increases resistance to attacks utilizing large amounts of keystream.

NLS stands for Non-Linear SOBER, and the NLS ciphers are members of the SOBER family of stream ciphers [15, 19 27 & 28]. The message authentication functionality is an instantiation of the Mundja streaming MAC [22], which in turn is based on SHA-256 [13].

2 Description of NLSv2

2.1 Introduction

NLSv2 is a synchronous stream cipher designed for a secret key that may be up to 128 bits in length. The cipher outputs the key stream in 32-bit blocks. NLSv2 is a software-oriented cipher based on simple 32-bit operations (such as 32-bit XOR and addition modulo 2^{32}), and references to small fixed arrays. Consequently, NLSv2 is at home in many computing environments, from smart cards to large computers. Source code for NLSv2 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

NLSv2 was developed from SOBER [27], proposed by Rose in 1998. The algorithm for SOBER is based on 8-bit operations, versus the 32-bit operations used in NLSv2. SOBER was superseded by SOBER-II [28] 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 [15]. 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 [26]; SOBER-t16 as a stream cipher with 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 [20] was developed as a very conservative stream cipher, but with an innovative (and flawed) message integrity functionality. Mundja [22], a message integrity primitive based on SHA-256 designed to cooperate with a stream cipher, was developed to rectify the flawed message integrity of SOBER-128.

NLS [23] is an improved version of SOBER-128 that incorporates the Mundja primitive. The message integrity functionality is unchanged from Mundja. The stream cipher functionality is changed from SOBER-128 by being based on a nonlinear feedback shift register instead of an LFSR, and with a simplified and more efficient filter function.

NLSv2 is a tweaked version of NLS, in which the only change is the periodic updating of the *Konst* variable (in NLS, *Konst* is a key-dependent variable that remains constant for the duration of keystream generation). This change was motivated by research that found a distinguishing attack [5] on NLS.

2.1.2 Usage and threat model

NLSv2 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, NLSv2 also supports this model of use.

NLSv2 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, NLSv2 may be used to generate a single encryption keystream of arbitrary length. In this mode it would be possible to use NLSv2 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. NLSv2 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.

NLSv2 is intended to provide security under the condition that no nonce is ever reused with a single key, that no more than 2^{80} words of data are processed with one key, and that no more than 2^{48} words of data are processed with one key/nonce pair. 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 NLSv2. The designers undertake to inform the ECrypt Stream Cipher project of any changes regarding the intellectual property claims covering NLSv2.

2.3 Outline of this Document

Section 3 contains a description of NLSv2. An analysis of the security characteristics, and corresponding design rationale of NLSv2 is found in Section 4. Section 5 outlines the strengths and advantages of NLSv2. 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

$f16$ is the 16th Fermat number, $2^{16}+1 = 65537$.

$a \lll b$ (*resp.* \ggg) 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 addition modulo 2^{32} .

\wedge is bit-wise “and” of 32-bit words.

\sim is bit-wise complement of a 32-bit word.

NLSv2 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.

3 Description

3.1 Keystream generation

NLSv2’s stream generator is constructed from a *non-linear feedback shift register* (NLFSR) and a *non-linear filter* (NLF), with assistance from a *counter*. The primitive is based on 32-bit operations and 32-bit blocks: each 32-bit block is called a *word*. The vector $\sigma_t = (r_t[0], \dots, r_t[16])$ of words is known as the *state* of the register at time t , and the state $\sigma_0 = (r_0[0], \dots, r_0[16])$ 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*. If a nonce is being used, then the key state is further perturbed by the nonce loading process to form the initial state, otherwise the key state is used directly as the initial state. During nonce-loading, the stream generator performs a new calculation of *Konst*, in order to make *Konst* dependent on both the key and the *nonce*. Once initialized, the stream generator produces 32-bit *keystream words* denoted v_j that combine to form the *keystream* $\{v_j\}$.

The NLFSR (described below) transforms state σ_t into state σ_{t+1} . Successive states σ_t from the NLFSR are fed through the filter to produce 32-bit *output words* denoted out_t . Each output word out_t is obtained using the NLF as

$$out_t = NLF(\sigma_t) = (r_t[0] + r_t[16]) \oplus (r_t[1] + r_t[13]) \oplus (r_t[6] + Konst) .$$

When $t \equiv 0$ (modulo $f16$), then out_t is used as a new value for $Konst$ and the value of out_t is not output as keystream. Otherwise, out_t is used directly as keystream. The mapping to keystream words $\{v_j\}$ from output words $\{out_t\}$ is $v_j = out_{j - (j \text{ div } f16)}$ where $(j \text{ div } f16)$ denotes the integer part of $(j \div f16)$.

The NLFSR transforms state σ_t into state σ_{t+1} in the following manner:

1. $r_{t+1}[i] = r_t[i+1]$, for $i = 0..15$
2. $r_{t+1}[16] = f(r_t[0] \lll 19) + (r_t[15] \lll 9) + Konst \oplus r_t[4]$
3. $r_t[0]$ is abandoned
4. if $t \equiv 0$ (modulo $f16$), then three special actions are applied; $r_{t+1}[2]$ is modified by adding t (modulo 2^{32}); $Konst$ is changed to the resulting value of out_{t+1} ; and the state out_{t+2} is computed as in steps 1 to 3.

The nonlinear function f is defined in section 3.3 below.

NLSv2 allows for any portion of the plaintext to be encrypted 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 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.

NLSv2 allows for encryption/decryption and authentication of plaintext of any length, but most of the operations are designed to act on 32-bit blocks of plaintext or transmission message. Section 3.4 describes how NLSv2 operates when the remaining plaintext (or transmission message) does not form a full 32-bit word.

3.2 The MAC accumulator

The MAC accumulation in NLSv2 is an instance of the Mundja streaming MAC [22]. Mundja accumulates the message into two independent 8-word registers: the first is a strengthened version of the SHA-256 register [13] that uses nonlinear feedback; the second is a 256-bit Cyclic Redundancy Checksum (CRC) that uses linear feedback. Unlike the SHA-256 algorithm, there is no message expansion or separation into blocks. The security corresponding to the message expansion lies in finally combining these orthogonal registers. Information from the stream cipher state is used to

initialize both of these registers, and instead of constants, further unknown input from the NLSv2 stream cipher state is taken as input to the SHA-256 round function. During analysis of the Mundja round function, we noticed [21] that the SHA-256 round function is not strong enough for the purposes of Mundja, in the sense of it being difficult to input different texts that cancelled out to create a collision in the SHA register, so Mundja enhances the round function slightly by the introduction of a nonlinear S-Box to speed diffusion.

Padding: Suppose that the length of the message, M , is l octets. Append k zero octets $0x00$, where k is the smallest, non-negative solution to the equation $l+k \equiv 0 \pmod{4}$. The length of the padded message should now be a multiple of 4 octets (that is, a multiple of 32-bits). The padded message must be less than 2^{64} words in length.

Parsing: The padded message is parsed into a sequence of 32-bit words, $\{Mt\}$.

Initialization: All stream ciphers are initialized using a secret key. It is well known that the keystream generated by a stream cipher should not be used more than once. In effect, this means that a stream cipher should start in a unique secret state for every message. For example, some stream ciphers require initialization using a nonce: a number used only once. These mechanisms must also be applied when using Mundja. First, the stream cipher is initialized to obtain a unique secret state using the key and using whatever mechanisms are applicable to that stream cipher. Then, sufficient words of the stream cipher state are taken to initialize the two Mundja registers.

SHA Register Update: The SHA register has 8 words of state A, B, C, D, E, F, G, H . The round function uses addition modulo 2^{32} and four functions: $CH, MJ, \Sigma_0, \Sigma_1$ with 32-bit inputs and 32-bit outputs that are non-linear with respect to modular addition. These functions are defined as:

$$CH(X,Y,Z) = (X \wedge Y) \oplus (\sim X \wedge Z);$$

$$MJ(X,Y,Z) = (X \wedge Y) \oplus (Y \wedge Z) \oplus (Z \wedge X);$$

$$\Sigma_0(X) = (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22);$$

$$\Sigma_1(X) = (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25).$$

The value of register state at the beginning of t -th round is denoted using a subscript t . The t -th round of the enhanced SHA round function modifies the SHA register using plaintext input word W_t , also using input word $r_t[8]$ (from the NLSv2 current state) as input. The round function uses the same functions as SHA-256 above, and additionally a nonlinear 32-bit function f (defined in section xxx below). The SHA round function modifies the SHA register according to the following algorithm:

1. $T1 = f(H_t + r_t[8] + W_t) + \Sigma_1(E_t) + CH(E_t, F_t, G_t);$
2. $T2 = \Sigma_0(A_t) + MJ(A_t, B_t, C_t);$

$$3. \quad H_{t+1} = G_t; G_{t+1} = F_t; F_{t+1} = E_t; E_{t+1} = D_t + T1;$$

$$4. \quad D_{t+1} = C_t; C_{t+1} = B_t; B_{t+1} = A_t; A_{t+1} = T1 + T2.$$

CRC Register: Another component of Mundja, used to parallel the effect of the data expansion in SHA-256, is a 256-bit cyclic redundancy checksum of the message words, calculated over $GF(2^{256})$. This CRC is calculated word-at-a-time, using an eight word LFSR defined over $GF(2^{32})$. 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 $(a_{w-1}, a_{w-2}, \dots, a_1, a_0)$, which represents the polynomial $a_{w-1}z^{w-1} + \dots + a_1z + a_0$. 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 $GF(2^{32})$ as degree-31 polynomials over $GF(2)$, Mundja uses 8-bit octets 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 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 $\beta_0 = 0x67$ 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 octets (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 $y^4 + \beta_3y^3 + \beta_2y^2 + \beta_1y + \beta_0$.

The CRC polynomial defined over $GF(2^{32})$, then, is $x^8 + x^3 + \alpha$. The coefficient $\alpha = 0x00000100 = 0x00y^3 + 0x00y^2 + 0x01y + 0x00 = y$, 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 the field multiplication used in SNOW [10] and is exactly that used in Turing [19] and SOBER-128 [20], so the same multiplication table can be used.

The `MultiTab` for NLSv2 is shown in Appendix B. In C code, the new word to be inserted in the LFSR is calculated:

$$\text{new} = R[3] \wedge (R[0] \ll 8) \wedge \text{Multab}[(R[0] \gg 24) \& 0xFF];$$

where \wedge is the XOR operation; \ll is the left shift operation; and \gg is the right shift operation. Finally, the irreducible polynomial defining the representation of the Galois field W was chosen to be $y^4 + 0xD0 \cdot y^3 + 0x2B \cdot y^2 + 0x43 \cdot y + 0x67$, for reasons explained in the SOBER-128 specification [20].

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

1. $T = M_t \oplus (\alpha \otimes \text{CRC}_t[0]) \oplus \text{CRC}_t[5];$
2. for $j = 0..6 : \text{CRC}_{t+1}[j] = \text{CRC}_t[j + 1];$
3. $\text{CRC}_{t+1}[7] = T.$

Finalization and generation of MAC: When all the words of the input message have been processed, another word is mixed into the SHA register in the same manner as above. Note that whenever a word is input to the SHA register, the partner stream cipher state is updated as if a word of keystream has been generated. The input word is $0x6996c53a + 2^{24}k$, recalling that k is the number of padding octets added. This word is not added 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 SHA register, to emulate the effect of the data expansion in SHA-256. Eight times, the CRC register is cycled as above with $M_t = 0$, and the value in $\text{CRC}_t[7]$ is input to the SHA register as above, updating the stream cipher state as required to get W_i . This serves to further mix and propagate any differences generated in either register during the input phase.

Finally, to generate the MAC, this process (mixing values from the CRC register into the SHA register) is continued with the values of A_t (from the SHA register) after each mixing step used as a word of the output MAC, until the requested amount of output has been produced. Words are converted to bytes in little-endian fashion, as usual for Mundja.

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 $f(a) = \text{SBox}[a_H] \oplus a$.

The S-box is a combination of the Skipjack [14] 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 [8]. 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 NLSv2 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 1). 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.

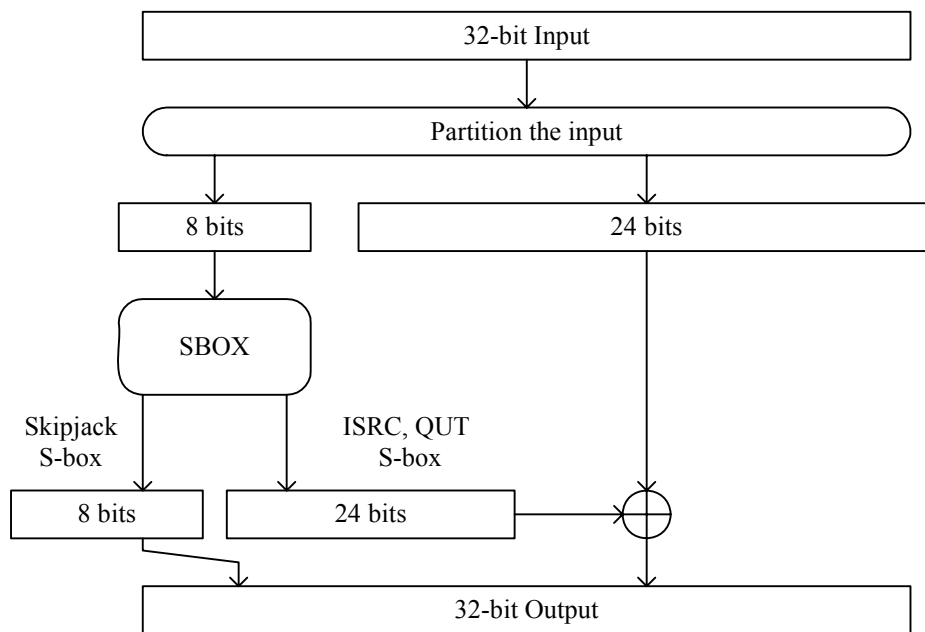


Figure 1. The structure of the function f used in NLSv2

3.4 Keystream generation

NLSv2 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 corresponding 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 or, at least, the nonlinear state is updated as if keystream was to be generated.

Keystream Mask: One method of implementing this is to form a *keystream mask* h_i for each word position, where KM_i 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 transmission message word c_i is formed from the corresponding plaintext word p_i , keystream word v_i and keystream mask KM_i s:

$$c_i = p_i \oplus (KM_i \wedge v_i),$$

where “ \wedge ” 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:

$$p_i = c_i \oplus (KM_i \wedge v_i).$$

Messages of unusual size: If the last portion of the plaintext (or the last portion of the transmission message bits) does not form a full 32-bit word, then the generated keystream word v_i is truncated to a keystream $v_{i(\text{short})}$ of suitable length.

3.5 The Key and nonce Loading

NLSv2 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 NLFSR, 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 nonce 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 $kw1 = keylen/4$ words and store in an array $kw[]$ of $kw1$ “little-endian” words
2. For each i , $0 \leq i \leq (kw1 - 1)$, Include($kw[i]$)
3. Apply Diffuse().
4. Include($keylen$).
5. 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 4.3.1. Including `keylen` ensures that keys and nonces of different lengths result in distinct initial states.

NLSv2 is keyed using a secret, t -byte session key $K[0], \dots, K[t-1]$ (and optional m -byte nonce $nonce[0], \dots, nonce[m-1]$) as follows:

1. The 17 words of state information are initialized to the first 17 Fibonacci numbers¹ by setting $R[0] = R[1] = 1$, and computing $R[i] = R[i-1] + R[i-2]$, for $2 \leq i \leq 16$. 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 NLFSR is clocked and the NLF output is calculated and *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, $(R[0], \dots, R[16])$, (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 nonces, 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 application uses nonces, then the cipher first resets the register state to the initial *key state* and resets *Konst* to INITKONST. The cipher then loads the m -byte nonce $nonce[0], \dots, nonce[m-1]$ using `Loadkey(nonce[],m)`. The NLFSR is clocked and the NLF output is calculated and *Konst* is then set to the resulting value. The resulting state of the register is taken as the initial state $r_0[i]$, $i = 0..16$, and the cipher produces a key stream with the register starting in this state. Note that a zero-length nonce is allowed, and is distinct from all other nonces and also distinct from the key state.
6. To initialize the Mundja MAC register words, the initial state words $r_0[i]$, $i = 0..15$ are simply copied into the SHA register words A,B, . . . ,H and the CRC register words $CRC[j]$, $j = 0..7$, in that order.

¹ There is no cryptographic significance to these numbers being used, except for the ease of generating them. Note that this provides an opportunity for “tweaking” NLSv2, as different initialization values at this point will result in completely distinct ciphers with identical security properties.

4 Security Analysis of NLSv2

Most of the components of NLSv2 have been subjected to scrutiny when they appeared in earlier members of the SOBER family of stream ciphers. 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 NLSv2, below.

4.1 Security Requirements

NLSv2 is intended to provide 128-bit security, although we believe it provides significantly more than that. The base attack on NLSv2 is an exhaustive key search, which has a computational complexity equivalent to generating 2^{128} keystream words². 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. NLSv2 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. With respect to the keystream functionality, we claim that NLSv2 fulfils the following security requirements, when used subject to the condition that no key/nonce pair is ever reused, and no more than 2^{80} words are processed with one key:

1. **Key/State Recovery Attacks:** NLSv2 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:** NLSv2 must resist attacks that accurately predict unknown values of the keystream without determining information about the NLFSR state or the secret key.
3. **Distinguishing attacks:** NLSv2 should resist attacks that distinguish a NLSv2 keystream from random bit stream.
4. **Related-Key or related nonce Attacks:** NLSv2 should resist attacks of the above form that use keystream generated from multiple key/nonce pairs 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).

² Unless, of course, a shorter secret key is used. We assume use of a 128-bit secret key in this section.

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. **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 ³. 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 $TAG(i) = MAC_K(M(i))$.

Mundja in NLSv2 is intended to be a MAC function with security parameters $n = 128$, and $d \leq 128$. That is, we claim that Mundja resists the above attacks when using 128-bit keys and outputting tags up to 128 bits in length.

NLSv2 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 register, 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

³ There are circumstances (albeit rare) where the users require a MAC function to resist a collision attack with known key. Mundja is not intended to prevent this type of attack. We note that other common MAC constructions, such as CBC-MAC [24], cannot prevent this type of attack either.

of less concern, since most security systems ensure that attackers cannot predict relationships between secret keys. However, it is still preferable that NLSv2 resists such 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 NLSv2 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 NLSv2 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 key space), the intention of the design is to ensure that there are no distinguishing attacks on NLSv2 requiring less than 2^{80} key stream words and less than 2^{128} computations. Since we do not allow NLSv2 to be used to process more than 2^{80} words, such an attack would not break NLSv2 if it is being used correctly. By comparison, AES-256 in counter mode has a distinguishing attack requiring 2^{66} keystream words.

4.2 Security Claims

We believe that any attack on NLSv2 has a complexity exceeding that of an exhaustive key search. We do not claim any mathematical proof of security. Our analysis of NLSv2 can be summarized thus:

- Guess-and-determine (GD) attacks [2] appear to have a computational complexity well in excess of 2^{128} (see [1, 18]).
- Algebraic attacks [6,7] appear to be infeasible.
- Correlation-based attacks [4] appear to be resisted by the register nonlinear feedback structure.
- Distinguisher attacks (such as [5]) are resisted by having *Konst* change periodically.
- 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 ways in which the key loading can be exploited.
- We are unaware of any weak keys or weak-key classes. Note that it is theoretically possible for the initial state to be entirely zero, but this is not relevant with the nonlinear feedback method.
- The output stream has a minimum cycle length of $C=2^{48}+2^{32}$, and any cycle must be a multiple of this value. We have no reason to believe that there are a significant number of cycles of length less

than 2^{80} . Our studies with the inclusion of the counter value disabled (thus removing any minimum cycle length) have been unable to demonstrate any cycle. Algebraic methods for constructing such a cycle have eluded us.

4.3 Heuristic Analysis of NLSv2

This analysis concentrates on vulnerability of NLSv2 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 NLSv2 have been taken from previous members of the SOBER family, or in the case of the MAC functionality, from SHA-256 and Mundja. Much 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. The S-box has been evaluated in detail.

4.3.1 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 nonce [9].
- 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 nonces.
- 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 nonces (up to 128-bits in length) can result in the same initial state.
- There is no initial state of the registers that is known to be weak in any sense, so it follows that there are no known weak keys.

We believe that these properties ensure that the key loading cannot be exploited.

4.3.2 Analysis of the stream cipher component.

The “shape” of the state register, in the sense of its feedback 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, which appear to have complexity greater than 2^{256} .

The feedback function of the stream cipher is highly nonlinear, through use of the Sbox. Rotations of the words used in the feedback function inputs ensure that all bits in the register have nonlinear effect

on the register contents quite rapidly (within 26 words of output). The nonlinear effects also compound quite rapidly. Since the register operates on nonlinear feedback, little can be proven about its cycle length, so a regular modification of the state is used to guarantee a minimum cycle length in excess of 2^{48} . This should be more than ample in practice. In the absence of any reason to believe that the feedback function behaves in a significantly non-random fashion, the average cycle length is approximately 2^{542} .

The nonlinearity of the feedback function, and the selection of the taps for the output filter function, should adequately disguise any short-distance correlations. The regular modification of the state should enhance this effect for long-distance correlations.

The output filter function is quite simple, and serves mostly to ensure that no exploitable combination of input words appears before many applications of the nonlinear Sbox have been applied.

The complexity of the Crossword Puzzle distinguisher attack [5] on the first version of NLS relies on detecting a *Konst*-dependent bias in a linear combination of keystream bits. For some values of *Konst* the bias tends towards the linear combination equaling zero, while for other values of *Konst* the bias tends towards the linear combination equaling one. When averaged over *Konst*, these biases cancel out and the average bias is zero (or very close to zero).

In the attack on the first version of NLS, the attacker can rely on getting a large amount of keystream generated from a single value of *Konst* for which the linear combinations of keystream bits will all have the same bias. However, by having *Konst* change periodically in NLSv2, the attacker is unable to find enough keystream with the identical values of *Konst*. The attacker must use keystream generated from multiple values of *Konst*. The resulting overall bias in the linear combination of keystream bits will (on average) tend to zero. There may be cases where a subset of the values of *Konst* result in similar biases, but these cases are not expected to be distinguishable without knowing (in advance) which subset of values to investigate.

4.3.3 Analysis of the Mundja MAC function

The MAC function's strength primarily relates to the strength of the SHA-256 hash function. Addition of a nonlinear Sbox to the round function significantly speeds nonlinear diffusion of differentials. Continuous input of key-dependent unknown input from the stream cipher register frustrates the design of specific differentials. A 256-bit CRC of the input data is included in the final "hash", to ensure that any differential introduced anywhere in the input stream is reintroduced at a later stage. For a more complete analysis of the Mundja MAC mechanism see [22].

5 Strengths and Advantages of NLSv2

NLSv2 has the following strengths and advantages.

- Speed: NLSv2 is fast, and in particular, compared to RC4, key loading is quite fast.
- 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

0 and Table 2 contain performance figures for NLSv2. The figures are for optimized C-language code and were obtained on a 1.6GHz Intel Pentium M running OpenBSD 3.8 and compiled with gcc 3.3.5. 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 “nlsfast.c” source code provided herewith.

MKeys/s	cycles/key	M IV/s	cycles/IV
1.879699	851.2	2.136752	758.8

Table 1. Computation required for key loading and nonce loading of NLSv2. These results were obtained by averaging the measurements for a large number of keys.

Length	Operation	Mbyte/second	cycles/byte
Continuous	Stream encryption	476.852419	3.355
1600-byte blocks	Stream encryption	425.531915	3.760
1600-byte blocks	Authentication only	133.333333	12.000
1600-byte blocks	Encryption + MAC generation	129.870130	12.310
1600-byte blocks	Decryption + MAC generation	119.760479	13.360

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

The implementation of the stream cipher requires 148 bytes of RAM. This accounts for:

- the register and *key state* (34 words or 136 bytes),
- A counter modulo $f16 \cdot 2^{32}$, which occupies 8 bytes. (Note: we choose to maintain the counter as two separate counters, modulo 2^{32} and modulo $f16$, in our reference source code.)

- *Konst* (four bytes).

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

The implementation of the MAC functionality requires a further 64 bytes of RAM, for the SHA register and the CRC register.

ROM memory is required for the field multiplication table (used in the NLFSR), and the non-linear S-box (used in the NLFv2); both 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.

In most environments that pass authenticated packets of data around, the counter that is used to prevent short cycles from occurring will never be used. Our reference code allows conditional compilation to remove support for this counter. This makes no measurable difference to the speed on our reference platform, but does result in smaller code.

NLSv2 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. 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. C. De Cannière, Guess and Determine Attack on SOBER, *NESSIE Public Document NES/DOC/SAG/WP5/010/a*, November 2001. See [26].
4. 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.
5. J. Cho and J. Pieprzyk, *Crossword Puzzle Attack on NLS*, 2006. See eprint.iacr.org/2006/049.pdf.
6. N. Courtois, Fast Algebraic Attacks on Stream Ciphers with Linear Feedback, awaiting publication. See <http://www.minrank.org/~courtois/myresearch.html>.

7. 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.
8. 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.
9. 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 [26].
10. P. Ekdahl, T. Johansson, SNOW - a new stream cipher, Proceedings of First Open NESSIE Workshop, KU-Leuven, 2000. See [26].
11. 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.
12. 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.
13. FIPS 180-2 Secure Hash Standard (SHS). See the following web page: csrc.nist.gov/CryptoToolkit/tkhash.html.
14. FIPS 185- Escrowed Encryption Standard. See the following web page: <http://www.itl.nist.gov/fipspubs/fip185.htm>.
15. P. Hawkes and G. Rose. The t-class of SOBER stream ciphers. Technical report, QUALCOMM Australia, 1999. See <http://www.qualcomm.com.au>.
16. P. Hawkes and G. Rose. *Primitive Specification and Supporting Documentation for SOBER-t16 Submission to NESSIE*. Submitted 2000. See <https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sober-t16.zip>
17. 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>
18. 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.
19. P. Hawkes and G. Rose. "Turing, a Fast Stream Cipher". In T. Johansson, Proceedings of *Fast Software Encryption FSE2003*, LNCS 2887, Springer-Verlag 2003.

20. P. Hawkes, G. Rose. *Primitive Specification for SOBER-128*, 2003. See eprint.iacr.org/2003/081.pdf.
21. P. Hawkes, M. Paddon and G. Rose. *On Corrective Patterns for the SHA-2 Family*, 2004. See eprint.iacr.org/2004/207.pdf.
22. P. Hawkes, M. Paddon and G. Rose. *The Mundja Streaming MAC*, 2004. See eprint.iacr.org/2004/271.pdf.
23. P. Hawkes, M. Paddon, G. Rose and M. Wiggers de Vries. *Primitive Specification for NLS*, 2005. See www.ecrypt.eu.org/stream/nls.html. Note: at time of writing, the specification for the original specification was available as www.ecrypt.eu.org/stream/ciphers/nls/nls.pdf.
24. International Organization for Standardization, Data Cryptographic Techniques-Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher Algorithm, ISO/IEC 9797, 1989.
25. A. Menezes, P. Van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
26. NESSIE: New European Schemes for Signatures, Integrity, and Encryption. See <http://www.cryptoneessie.org>.
27. 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.
28. 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>.
29. D. Watanabe, S. Furuya, *A MAC forgery attack on SOBER-128*, Proc. Fast Software Encryption 2004, Springer 2004.

8 Appendix: Recommended C-language interface

```
typedef struct {
    WORD    R[N];          /* Working storage for the shift register */
    WORD    M[NMAC];      /* Working storage for MAC accumulation */
    WORD    CRC[NMAC];    /* Working storage for CRC accumulation */
    WORD    initR[N];     /* saved register contents */
    WORD    konst;        /* key dependent constant */
}
```

```

WORD      sbuf;          /* partial word encryption buffer */

WORD      mbuf;          /* partial word MAC buffer */

int       nbuf;          /* number of part-word stream bits buffered */

WORD      CtrModF16;     /* Multiprecision counter, modulo f16 */

WORD      CtrMod232;     /* Multiprecision counter, LSW */

} nls_ctx;

/* interface definitions */

void nls_key(nls_ctx *c, UCHAR key[], int keylen);      /* set key */

void nls_nonce(nls_ctx *c, UCHAR nonce[], int nlen);    /* set nonce */

void nls_stream(nls_ctx *c, UCHAR *buf, int nbytes);    /* stream cipher */

void nls_maconly(nls_ctx *c, UCHAR *buf, int nbytes);   /* accumulate MAC */

void nls_encrypt(nls_ctx *c, UCHAR *buf, int nbytes);   /* encrypt + MAC */

void nls_decrypt(nls_ctx *c, UCHAR *buf, int nbytes);   /* decrypt + MAC */

void nls_finish(nls_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. 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 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, 0xaf4429f, 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, 0xdfcbbc2b3, 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, 0xe9dala6b, 0x97888c02,
0x19d1e354, 0x5aba7d79, 0xa2cc7753, 0x8c2d9655,
0x19829da1, 0x531590a7, 0x19c1c149, 0x3d537f1c,
0x50779b69, 0xed71f2b7, 0x463c58fa, 0x52dc4418,
0xc18c8c76, 0xc120d9f0, 0xaf80d4d, 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,
0xe1cffffbd, 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,
};

10 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, 0xBF61F0A,
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,  
};
```