Design and Primitive Specification for Shannon

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

{phawkes,cameronm,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

Shannon is a synchronous stream cipher with message authentication functionality, designed according to the ECrypt NoE call for stream cipher primitives, profile 1A (but well after the call).

Shannon is named in memory of Claude E. Shannon[20] of Bell Labs and MIT, founder of Information Theory. Shannon is an entirely new design, influenced by members of the SOBER family of stream ciphers[11][13][19], Helix/Phelix[7], Trivium[4], Scream[10], and SHA-256[8]. It consists of a single 32-bit wide, 16-element nonlinear feedback shift register and an extra word, which is supplemented for message authentication with 32 parallel CRC-16 registers.

Shannon is free to use for any purpose, and reference source code can be found at http://www.qualcomm.com.au/Shannon.html .

2 Introduction

Shannon is a synchronous stream cipher designed for a secret key that may be up to 256 bits in length. The cipher outputs the key stream in 32-bit blocks. Shannon is a software-oriented cipher based on simple 32-bit operations (operations on data are restricted to XOR, OR and fixed rotations. Consequently, Shannon is at home in many computing environments, from simple hardware implementations through smart cards to large computers. Source code for Shannon is freely available and use of this source code, or independent implementations, is allowed free for any purpose.

Shannon is a back-to-basics design incorporating lessons learned from a variety of sources. From members of the SOBER family of stream ciphers, it gets its basic shift register structure. Helix introduced the hybrid stream cipher directly incorporating message authentication. Trivium showed how a simple nonlinear feedback structure could compound rapidly to provide security, Scream first taught the value of keeping the nonlinearity in the cipher state. SHA-256, in its resistance to the attacks against earlier hash functions[15], demonstrates the importance of propagating differentials forward for message authentication codes. Finally, many aspects of the design have been influenced by the theory of Golomb Rulers[9] (also often known as Full Positive Difference Sets). The use of only extremely primitive operations and no tables follows work by Bernstein[1] on timing attacks related to table lookups.

2.1 Usage and threat model

Shannon 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, Shannon also supports this model of use.

Shannon 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, Shannon may be used to generate a single encryption keystream of arbitrary length. In this mode it would be possible to use Shannon 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. Shannon achieves this using a single secret key for the entire (multi-message) communication, with a nonce distinguishing individual messages. Section 8 below describes the recommended interface.

Shannon 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 Shannon. Should Shannon be formally submitted to any evaluation process the designers undertake to inform the project of any changes regarding the intellectual property claims covering Shannon.

2.3 Outline of this Document

Section 3 contains a description of Shannon. An analysis of the security characteristics, and corresponding design rationale of Shannon is found in Section 4. Section 5 outlines the strengths and advantages of Shannon. Computational efficiency is discussed in Section 6. Appendices provide a recommended C-language interface.

2.4 Notation and conventions

$a<<<b$ resp. $>>>$ means rotation of the word a left (respectively right) by b bits; note that Shannon uses only constants for the rotation amount.

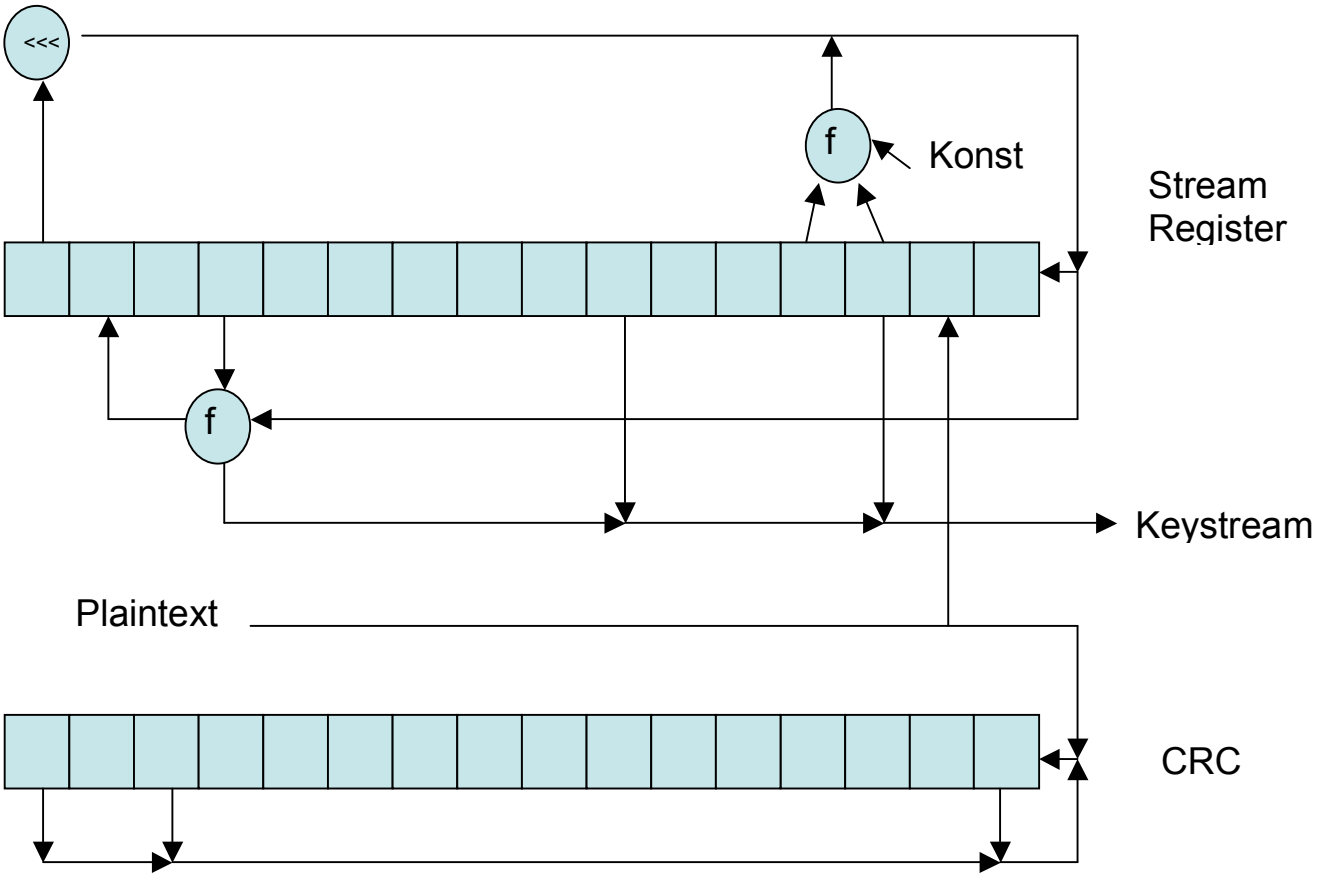
$\oplus$ is exclusive-or of words.

^ is bit-wise “and” of 32-bit words.

| is bit-wise “or” of 32-bit words.

Shannon 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 (as is native on Intel CPUs) irrespective of the byte ordering of the underlying machine.

3 Description



3.1 Keystream generation

Shannon’s stream generator is constructed from a *non-linear feedback shift register* and an *output filter*. The primitive is based on 32-bit operations and 32-bit blocks: each 32-bit block is called a *word*. The vector of words $\sigma_t = (r_t[0], \ldots, r_t[15])$ is known as the *state* of the register at time t, and the state $\sigma_0 = (r_0[0], \ldots, r_0[15])$ is called the *initial state*. The *key state* is initialised from the secret key by the *key loading*. The key state can be used directly as the initial state, or can be further perturbed by the nonce loading process to form the initial state. An additional state word called *Konst* (for historical reasons) is derived from the key loading or nonce loading, and is used in the nonlinear feedback.

The state transition function transforms state σ_t into state σ_{t+1}, and derives an output keystream word v_t, in the following manner:

1. $r_{t+1}[i] \leftarrow r_t[i+1]$, for $i = 1..14$ (that is, the middle words of the register are derived by shifting).

2. $r_{t+1}[15] \leftarrow f_1(r_t[12] \oplus r_t[13] \oplus Konst) \oplus (r_t[0] <<< 1)$

3. Let $t \leftarrow f_2(r_{t+1}[2] \oplus r_{t+1}[15])$

4. $r_{t+1}[0] \leftarrow r_t[1] \oplus t$ (that is, "feed forward" to the new lowest element)

5. $v_t \leftarrow t \oplus r_{t+1}[8] \oplus r_{t+1}[12]$

The nonlinear functions f_1 and f_2 are defined in section 3.3 below.

Shannon 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 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.

Shannon 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 Shannon 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 Shannon is a combination of nonlinear accumulation in the main shift register (as in Helix) and linear accumulation in 32 parallel CRC registers, which we implement as a word-wide 16-word register.

Padding: Suppose that the length of the message, M, is l bits. Append k zero bits, where k is the smallest, non-negative solution to the equation $l+k \equiv 0 \pmod{32}$. The length of the padded message should now be a multiple of 32 bits. The padded message must be less than 2^{48} words in length.

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

Initialization: Shannon requires initialisation using a nonce for messages to be authenticated. First, the stream cipher is initialized to obtain a unique secret state using the key and nonce. Then the CRC register CRC_0 is initialised as a copy of σ_0.

CRC Register: Another component of Shannon, used to parallel the effect of the data expansion in SHA-256, is a parallel 16-bit cyclic redundancy checksum of the message words. This CRC is calculated word-at-a-time using a standard CRC polynomial $x^{16} + x^{15} + x^2 + 1$.

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

1. $CRC_{t+1}[i] \leftarrow CRC_t[i+1]$, for $i = 1..15$.
2. $CRC_{t+1}[15] \leftarrow CRC_t[0] \oplus CRC_t[2] \oplus CRC_t[15] \oplus M_t$
3. $r_{t+1}[13] \leftarrow r_t[14] \oplus M_t$ (that is, a word of the stream cipher register is updated with the plaintext word)

Finalization and generation of MAC: When all the words of the input message have been processed, another word is mixed into the cipher register in the manner of step 3 above. The input word is 0x6996c53a⊕k, recalling that k is the number of padding bits 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 combined into the stream register, to emulate the effect of the data expansion in SHA-256.

1. $r_{t+1}[i] \leftarrow r_{t+1}[i] \oplus CRC_t[i]$, for $i = 1..15$.

The stream register is then cycled 16 times to thoroughly mix the CRC contents. Finally, to generate the MAC, the stream cipher is run until the requested amount of output has been produced.

3.3 The S-Box Functions f_1 and f_2

There are two related nonlinear functions used in the state updating (and output) function of Shannon, mapping an input word to an output word. Both functions have the same form given an input word w:

1. Let $t \leftarrow w \oplus ((w <<< A) \mid (w <<< B))$
2. Return $t \oplus ((t <<< C) \mid (t <<< D))$

The only difference between the functions is the ordering of the constants {A, B, C, D}. For f_1 they are {5, 7, 19, 22} respectively, while for f_2 they are {7, 22, 5, 19} respectively. Each bit of the output word is a fourth-degree function of 5 bits of the input. Thus, such functions have a bias of 2^{-3}.

3.4 Keystream generation

Shannon 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 ciphertext 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 were to be generated.

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_t is truncated to a keystream $v_{t(\text{short})}$ of suitable length.

3.5 The Key and Nonce Loading

Shannon is keyed and re-keyed (that is, incorporating the nonce) by using operations that transform the values in the register under the influence of key material. The method is almost identical to that used for accumulation of plaintext for the MAC: the difference is that the key material is XORed into $r[13]$ immediately *before* cycling the register. This allows slightly more efficient diffusion, and also ensures that the operation of key loading cannot be simulated by data encryption.

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`):

Keys may be an arbitrary number of bytes; when a key is not a multiple of four bytes it is effectively padded with zero bytes until it is. Note that the key length in bytes is incorporated in the process below, so that a padded key is not equivalent to any other longer key. Shannon is designed to be secure equivalent to brute-force attacks up to 256-bit keys, with appropriate use of nonces to avoid time-memory tradeoffs.

1. Pad the key with zeros to a multiple of four bytes.

2. Convert `k[]` into `kwl` = `keylen`/4 words and store in an array `kw[]` of `kwl` “little-endian” words

3. For each i, $0 \le i \le$ (`kwl` –1), set $r[13] = r[13] \oplus$ `kw`$[i]$ then cycle the register

4. set $r[13] \leftarrow r[13] \oplus$ `keylen` then cycle the register

5. Save a copy of the register: set $crc[i] \leftarrow r[i]$, i=0..15

6. Cycle the register 16 times according to the algorithm of section 3.1.

7. set $r[i] \leftarrow r[i] \oplus crc[i]$, i=0..15 ■

The 16 cycles of the register are designed to ensure that every bit of input affects every bit of the resulting register state in a nonlinear fashion, as discussed in Section 4.4.1. Including `keylen` ensures that keys and nonces of different lengths result in distinct initial states. The effect of saving the register and XORing it back later is to make the diffusion process non-reversible.

Initial keying. Shannon is keyed using a secret, t-byte session key $K[0], \ldots, K[t-1]$ (and optional m-byte nonce $nonce[0], \ldots, nonce[m-1]$) as follows:

1. The 16 words of state information are initialized to the first 16 Fibonacci numbers[1] by setting `R[0] = R[1] = 1`, and computing `R[`i`] = R[`i`-1] + R[`i`-2]`, for $2 \le i \le 15$. The value of *Konst* is initially 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. If the cipher is going to be used for multiple messages with distinct nonces, then the 16 word state of the register, (`R[0],…,R[15]`), (which we call the *key state*) can be saved at this point for later use to initialise the register before loading the nonce, and the real key can be 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.

4. 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, sets *Konst* to INITKONST, and loads the m-byte nonce $nonce[0], \ldots, nonce[m-1]$ using Loadkey($nonce$[],m). *Konst* is then set to the current value of $r[0]$. The state of the register following the Loadkey is taken as the initial state, and the cipher produces a key stream with the register starting in this state. Note that a zero-length nonce is allowed, and is distinct from all other nonces and also distinct from the key state.

5. To initialize the Shannon CRC register words are simply copied from the stream register words immediately before the diffusion process (see step 5 of Algorithm Loadkey above).

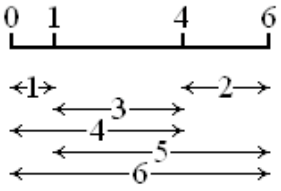
4 Design and Security Analysis of Shannon

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

[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" Shannon, as different initialization values at this point will result in completely distinct ciphers with identical security properties.

4.1 Design using Golomb Rulers

The design of Shannon as a nonlinear shift register required three applications of Golomb Rulers, or Full Positive Difference Sets. *In mathematics, a Golomb ruler, named for Solomon W. Golomb, is a set of marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between any two of its marks is its length. Translation and reflection of a Golomb ruler are considered trivial, so the smallest mark is customarily put at 0 and the next mark at the smaller of its two possible values.*



[Description and image courtesy of Wikipedia.]

Shannon requires three applications of Golomb Rulers. The first is for the basic nonlinear shift register. Given the selected 16-word length of the register, we require a ruler of length 17 with marks at 0 and 16. Simultaneously, though, we require a ruler for the output and "feed forward" function. This must be a ruler of length at most 16. These two rulers needed to be generated simultaneously and machine optimised for their resistance to "guess and determine" attacks (see, for example, [2], [3]). There are exactly three combinations that required the maximum 11 words of guessing before being able to determine the register. This is equivalent to an effort of 2^{384} (considering that Konst must also be guessed before mounting the attack). From these three possibilities, we chose marks of (0, 12, 13, 16) for the main feedback register and (2, 8, 12, 15) for the output function for no particular reason.

We wanted the nonlinearity in Shannon to come from a very simple function that transformed a single word using rotations and Boolean operations. It is important for this application to get maximum diffusion as the nonlinear function is "piled up", so we want to apply the concept of a Golomb Ruler to the shift amounts for the 32-bit words. However, rotations "wrap around" so we want to apply the concept "modulo 32". The chosen set of rotations is (0, 1, 5, 7, 19, 22). The constraints were:

1. Should have 0 (because no rotation is no work) and 1 (some embedded CPUs can execute rotations by a single bit more efficiently).

2. Maximum possible number of marks (6)

3. Of the possible even differences, we want the set that minimises the greatest power of two that divides any difference. (4)

4. As many odd marks as possible (5 is possible without the previous constraint, but only 4 with it).

We wanted to use the single-bit rotation in the main feedback loop, and the "no rotation" for the base of the nonlinear functions, leaving the other four marks for the rotations inside the function. There are

therefore six such functions, which were exhaustively tested for coverage (that is, the number of possible results given that the functions are not bijective). The function f_1 has the best coverage (84.74%) while f_2 had nearly as good coverage (84.34%) while combining the offsets in different pairs. It was a surprise to us that two of the possible combinations had particularly bad coverage (around 30%).

4.2 Security Requirements

Shannon is intended to provide 256-bit security. The base attack on Shannon is an exhaustive key search, which has a computational complexity equivalent to generating 2^{256} 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. Shannon 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^{256} times and generating at least 5 words of output each time. With respect to the keystream functionality, we claim that Shannon fulfils the following security requirements, when used subject to the condition that no key/nonce pair is ever reused, and that no more than 2^{48} words of data are processed with one key/nonce pair[3], and no more than 2^{80} words are processed with one key:

1. **Key/State Recovery Attacks**: Shannon 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**: Shannon 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**: Shannon should resist attacks that distinguish a Shannon keystream from random bit stream.

4. **Related-Key or related nonce Attacks**: Shannon should resist attacks of the above form that use keystreams 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).

Suppose the received message-tag pair is (RM,RTAG). The receiver decrypts the appropriate parts of the message and calculates an expected tag $XTAG = MAC_K(RM)$. If XTAG = RTAG, then the receiver

[2] Unless, of course, a shorter secret key is used. We assume use of a 256-bit or longer secret key in this section.
[3] Using no nonce at all is allowed, but this condition applies.

has some confidence that the message-tag pair was formed by a party that knows the key K. In most 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 $O(2^{\min(n,d/2)})$. Note that meaningful collision attacks against MAC functions are rare in practice.

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[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 $TAG(i) = MAC_K(M(i))$.

Shannon is intended to be a MAC function with security parameters n = 256, and $d \leq 128$. That is, we claim that Shannon resists the above attacks when using 256-bit keys and outputting tags up to 128 bits in length.

Shannon 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 of less concern, since most security systems ensure that attackers cannot predict relationships between secret keys. However, it is still preferable that Shannon resists such attacks.

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

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 cipher 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 Shannon 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 Shannon is still secure even if there is a distinguishing attack requiring 2^{64} key stream words and 2^{80} computations. Although the designers hold the second view (that stream ciphers can still be secure even when the complexities of distinguishing attacks fall below the bounds of the keyspace), the intention of the design is to ensure that there are no distinguishing attacks on Shannon requiring less than 2^{80} key stream words and less than 2^{128} computations. Since we do not allow Shannon to be used to process more than 2^{80} words, such an attack would not break Shannon if it is being used correctly. By comparison, AES-256 in counter mode has a distinguishing attack requiring 2^{66} keystream words.

A comment on nonce reuse. As with any stream cipher, an attacker who can force the reuse of a nonce can easily breach privacy. However the fact that the stream cipher register is plaintext aware introduces a much worse set of attacks if the attacker can both force reuse of the nonce and choose ciphertext and/or plaintext values to exploit differential cryptanalysis; the initial state of the register can then be determined. Since this is a fairly catastrophic failure, Shannon's key loading procedure has been made irreversible (at the very slight risk of equivalent key/nonce pairs), so that disclosure of the initial state doesn't reveal enough information to enable arbitrary forgery or disclose the key. Note that the attacker can presumably pass such messages easily to the recipient; this gives rise to the requirement that the recipient must verify the MAC before allowing access to the decrypted plaintext. At the sender's end the attacker would need to force the sender to both break the nonce protocol and accept chosen plaintext, which is a more difficult attack.

4.3 Security Claims

We believe that any attack on Shannon has a complexity exceeding that of generic attacks (e.g. exhaustive key search, time-memory tradeoffs, etc.) up to 256-bit keys. We do not claim any mathematical proof of security. Our analysis of Shannon can be summarized thus:

- Guess-and-determine (GD) attacks [2] appear to have a computational complexity well in excess of 2^{256} (see [2, 12]).

- Algebraic attacks [6] appear to be infeasible due to the rapid accumulation of nonlinearity in the shift register.

- Long-distance correlation-based attacks appear to be resisted by the register nonlinear feedback structure.

- "Crossword puzzle" attacks [5] are infeasible because the bias of the nonlinear functions, "piled up" over the length of the register, is less than 2^{-48}.

- Timing, power, cache timing and branch prediction 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.

- As an observation, we note that the entire construction is rotationally and reflectionally symmetric; if all the register words, Konst, and inputs are rotated a constant amount the outputs are similarly rotated. We know of no consequence to this except for a negligible shortening of the expected cycle length. Note, however, that Konst and its initial value cannot actually be rotated in normal operation of Shannon.

- We have no reason to believe that there are a significant number of cycles of length less than 2^{80}. Our studies have been unable to demonstrate any cycle. Algebraic methods for constructing such a cycle have eluded us. Assuming that the nonlinear functions behave well, the expected cycle length is 2^{505} words.

4.4 Heuristic Analysis of Shannon

This analysis concentrates on vulnerability of Shannon 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.

Some of the features of Shannon 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.

4.4.1 Analysis of the Key Loading

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

- The key length is included to ensure that there are no equivalent secret keys or equivalent nonces.

- No two secret keys (up to 256-bits in length) can result in the same initial key state. Also, given a key state, no two nonces (up to 256-bits in length) can result in the same initial state. This is because the nonlinear shift register is reversible even though the S-box functions are not bijective, in the same way that a Feistel block cipher can be reversed.

- 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.

4.4.2 Analysis of the stream cipher component.

The "shape" of the state register, in the sense of its feedback taps and nonlinear filter function taps, 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 somewhat nonlinear, through use of the functions f_1 and f_2. 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 8 words of output). The nonlinear effects also compound quite rapidly due to the tap and "feed forward" positions. 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 should be approximately 2^{505} (accounting for the rotational symmetry noted above, and referring to [17]).

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 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.

4.4.3 Analysis of the Shannon MAC function

The MAC function's strength primarily relates to the strength of the nonlinear stream cipher. A 512-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.

5 Strengths and Advantages of Shannon

Shannon has the following strengths and advantages.

- Speed: Shannon 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

Table 1 and 0 contain performance figures for Shannon. The figures are for optimized C-language code and were obtained on a 1.67GHz PowerPC G4 (Mac Powerbook) running Mac OS X 10.4.8 and compiled with gcc –O3 (gcc 4.0.1). (Note that this is a big-endian machine, so the times below include the required byte-swapping. Also cache effects dominate the figure for continuous stream encryption, as the buffer is far bigger than that cache. This shows that the simple integer operations are able to perform encryption at a speed comparable to that of simply moving the data in memory!) 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 "ShannonFast.c" source code provided herewith.

MKeys/s	cycles/key	Mnonces/s	cycles/nonce
2.03252	821.6401315	2.41546	691.379982
GHz	1.67		

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

Length	Operation	Mbyte/second	cycles/byte
Continuous	Stream encryption	120.65	13.84
1600-byte blocks	Stream encryption	163.93	10.19
1600-byte blocks	Authentication only	232.56	7.18
1600-byte blocks	Encryption + MAC generation	151.52	11.02
1600-byte blocks	Decryption + MAC generation	150.38	11.11

Table 2. Performance figures for encryption and message authentication. The block figures include the time for nonce loading and MAC finalization. This also includes byte-swapping as the PowerPC is a big-endian machine.

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

- the stream register and *key state* (32 words or 128 bytes),
- *Konst* (four bytes).

Small memory implementation would omit the key state and just re-key the cipher for each nonce; in this case 68 bytes of RAM are required.

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

No ROM memory is required for any tables.

Shannon uses only simple word-oriented instructions, such as OR, XOR, and rotation. 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. D. Bernstein, "Cache-timing attacks on AES", `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
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 [18].
4. Christophe De Cannière, Bart Preneel. "Trivium - A Stream Cipher Construction Inspired by Block Cipher Design Principles". `http://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf`.
5. J. Cho, "Crossword Puzzle attack on NLS", submitted to FSE 2007.
6. N. Courtois and W. Meier, "Algebraic attacks on Stream Ciphers with Linear Feedback", proceedings of *EUROCRYPT 2003*, Warsaw, Poland, May 2003.
7. 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.
8. FIPS 180-2 Secure Hash Standard (SHS). See the following web page: csrc.nist.gov/CryptoToolkit/tkhash.html.
9. Wikipedia, `http://en.wikipedia.org/wiki/Golomb_ruler`

10. S. Halevi, D. Coppersmith, C. Jutla, "SCREAM: a software efficient stream cipher", Eprint 2002/19.

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

12. 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.

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

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

15. P. Hawkes, M. Paddon and G. Rose. *On Corrective Patterns for the SHA-2 Family,* 2004. See eprint.iacr.org/2004/207.pdf.

16. P. Hawkes, M. Paddon and G. Rose. "*The Mundja Streaming MAC",* 2004. See eprint.iacr.org/2004/271.pdf.

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

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

19. 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.

20. http://en.wikipedia.org/wiki/Claude_Shannon

8 Appendix: Recommended C-language interface

```
#define N 16

typedef struct {
  uint32_t      R[N];          /* Working storage for the shift register */
  uint32_t      CRC[NMAC];     /* Working storage for CRC accumulation */
  uint32_t      initR[N];      /* saved register contents */
  uint32_t      konst;         /* key dependent constant */
  uint32_t      sbuf;          /* encryption buffer */
```

```
    uint32_t        mbuf;           /* partial word MAC buffer */
    int             nbuf;           /* number of part-word stream bits buffered */
} shn_ctx;
/* interface definitions */
void shn_key(shn_ctx *c, uint8_t key[], int keylen);    /* set key */
void shn_nonce(shn_ctx *c, uint8_t nonce[], int nlen);          /* set nonce */
void shn_stream(shn_ctx *c, uint8_t *buf, int nbytes);          /* stream cipher */
void shn_maconly(shn_ctx *c, uint8_t *buf, int nbytes);         /* accumulate MAC */
void shn_encrypt(shn_ctx *c, uint8_t *buf, int nbytes);         /* encrypt + MAC */
void shn_decrypt(shn_ctx *c, uint8_t *buf, int nbytes);         /* decrypt + MAC */
void shn_finish(shn_ctx *c, uint8_t *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.