A New Version of the Stream Cipher SNOW

Patrik Ekdahl and Thomas Johansson

Dept. of Information Technology
Lund University, P.O. Box 118, 221 00 Lund, Sweden
{patrik,thomas}@it.lth.se

Abstract. In 2000, the stream cipher SNOW was proposed. A few attacks followed, indicating certain weaknesses in the design. In this paper we propose a new version of SNOW, called SNOW 2.0. The new version of the cipher does not only appear to be more secure, but its implementation is also a bit faster in software.

Keywords. SNOW, Stream ciphers, summation combiner, correlation attacks.

1 Introduction

A stream cipher is a cryptographic primitive used to ensure privacy on a communication channel. A common way to build a stream cipher is to use a pseudo-random length-increasing function (or keystream generator) and mask the plaintext using the output from the keystream generator. Typically, the masking operation is the XOR operation, and the keystream output is thus used as a one-time-pad to produce the ciphertext.

A number of stream ciphers have been proposed during the history of cryptology. Most of them have been bit-oriented stream ciphers based on linear feedback shift registers (LFSRs). These range from the simple and very insecure Geffe generator, nonlinear combination generators, filter generators, to the more interesting clock-controlled generators like the (self-) shrinking generator and the alternating step generator [13].

Apart from security, the main characteristic of a stream cipher is its performance. Performance can be the speed of an implemented cipher on different platforms, but also chip area, power consumption etc. for hardware implementations.

A general research topic for any cryptographic primitive is to try to optimize the trade-off between security and performance. Bit-oriented stream ciphers do not perform very well in software implementations. This is the reason why we have recently seen word-oriented stream ciphers. A word-oriented stream cipher outputs a sequence of words of a certain word size (like 32 bits). Such a cipher can provide a very good performance, typically 5-10 times faster than a block cipher in a software implementation.

Several word-oriented stream ciphers have recently been proposed, e.g., RC4 [14], SEAL [15], different versions of SOBER [9,10], SNOW [5], SSC2 [17],

K. Nyberg and H. Heys (Eds.): SAC 2002, LNCS 2595, pp. 47–61, 2003.

SCREAM [2], MUGI [16]. It can be noted that essentially all of the proposed stream ciphers have documented weaknesses of varying strength (this does not include SCREAM and MUGI that were proposed in 2002).

The purpose of this paper is to propose a new version of the SNOW cipher. The original version, now denoted SNOW 1.0, was submitted to the NESSIE project. It has excellent performance, several times faster than AES. However, a few attacks have been reported. One attack is a key recovery attack requiring a known output sequence of length 2^{95} having expected complexity 2^{224} [7]. Another attack is a distinguishing attack [1] also requiring a known output sequence of length 2^{95} and about the same complexity. Although one might argue about the relevance of such a distinguishing attacks, the attacks do demonstrate some weaknesses in the design.

In this paper we propose a new version of SNOW, called SNOW 2.0, which appears to be more secure. Moreover, SNOW 2.0 can be implemented even faster than SNOW 1.0 in software. Our optimized C implementation reports a speed of about 5-6 clock cycles per byte.

The paper is organized as follows. In Section 2 we describe the original design, in the sequel referred to as SNOW 1.0. In Section 3 we describe the weaknesses found in SNOW 1.0. In Section 4 we present the new version SNOW 2.0, and in Section 5 we discuss the design differences between the two versions. In Section 6 we then focus on implementation aspects.

2 First Version of SNOW

In this section we give a short description of the original SNOW design. SNOW 1.0 is a word oriented stream cipher with a word size of 32 bits.

The cipher is described with two possible key sizes, 128 and 256 bits. As usual, the encryption starts with a key initialization, giving the components of the cipher their initial key values. In this description we will only concentrate on the cipher in operation. The details of the key initialization can be found in [5].

The generator is depicted in Figure 1. It consists of a length 16 linear feedback shift register over $\mathbb{F}_{2^{32}}$, feeding a finite state machine. The FSM consists of two 32 bit registers, called R1 and R2, as well as a some operations to calculate the output and the next state (the next value of R1 and R2).

The operation of the cipher is as follows. First, key initialization is done. This procedure provides initial values for the LFSR as well as for the R1,R2 registers in the finite state machine. Next, the first 32 bits of the keystream is calculated by bitwise adding the output of the FSM and the last entry of the LFSR. After that the whole cipher is clocked once, and the next 32 bits of the keystream is calculated by again bitwise adding the output of the finite state machine and the last entry of the LFSR. We clock again and continue in this fashion.

Returning to Figure 1, the LFSR has a primitive feedback polynomial over $\mathbb{F}_{2^{32}}$ which is

$$p(x) = x^{16} + x^{13} + x^7 + \alpha^{-1},$$

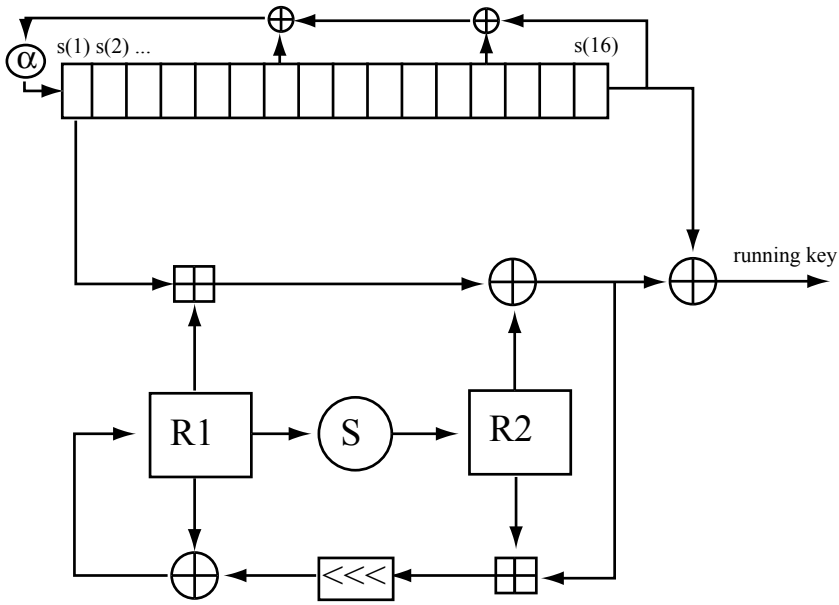

Fig. 1. A schematic picture of SNOW 1.0

where $\mathbb{F}_{2^{32}}$ is generated by the irreducible polynomial

$$\pi(x) = x^{32} + x^{29} + x^{20} + x^{15} + x^{10} + x + 1,$$

over $\mathbb{F}_2$, and $\pi(\alpha) = 0$. Furthermore let $s(1), s(2), \ldots s(16) \in \mathbb{F}_{2^{32}}$ be the state of the LFSR.

The output of the FSM, called FSM_{out}, is calculated as follows.

$$FSM_{\text{out}} = (s(1) \boxplus R1) \oplus R2.$$

The output of the FSM is XORed with $s(16)$ to form the keystream, i.e.,

$$\text{running key} = FSM_{\text{out}} \oplus s(16).$$

The keystream is finally XORed with the plaintext, producing the ciphertext.

Inside the FSM, the new values of R1 and R2 are given as follows,

$$\begin{aligned} \text{newR1} &= ((FSM_{\text{out}} \boxplus R2) \lll) \oplus R1, \\ R2 &= S(R1), \\ R1 &= \text{newR1}. \end{aligned}$$

By the notation $x \boxplus y$, we mean the integer addition of x and y mod2^{32}. The notation $x \lll$ is a cyclic shift of x 7 steps to the left, and the addition sign in $x \oplus y$ represents bitwise addition (XOR) of the words x and y.

Finally, the S-box, denoted $S(x)$, consists of four identical 8-to-8 bit S-boxes and a permutation of the resulting bits. The input x is split into 4 bytes, each byte enters a nonlinear mapping from 8 bits to 8 bits. After this mapping, the bits in the resulting word are permuted to form the final output of the S-box. A comprehensive description of the original design, including the key setup and modes of operation can be found in [5].

3 Weaknesses in SNOW 1.0

In this section we describe the weaknesses found in the original construction. In February 2002, Hawkes and Rose described a guess-and-determine attack on SNOW 1.0 [7,8]. The attack has data complexity of 2^{95} words and a process complexity of 2^{224} operations. Apart from some clever initial choices made by Hawkes and Rose, basically two properties in SNOW 1.0 are used to reduce the complexity of the attack below exhaustive key search. First, the fact that the FSM has only *one* input $s(1)$. This enables an attacker to invert the operations in the FSM and derive more unknowns from only a few guesses. The second property is an unfortunate choice of feedback polynomial in SNOW 1.0. The linear recurrence equation is given by

$$s_{t+16} = \alpha(s_{t+9} + s_{t+3} + s_t). \tag{1}$$

There is a distance of 3 words between s_t and s_{t+3} and a distance of $6 = 2 \cdot 3$ between s_{t+3} and s_{t+9}. Thus, by squaring (1)

$$s_{t+32} = \alpha^2(s_{t+18} + s_{t+6} + s_t) \tag{2}$$

we see that $(s_{t+i} \oplus s_{t+i+6})$ can be considered as a single input to either equation. Hence, the attacker does not need to determine both s_{t+i} and s_{t+i+6} explicitly, but only the XOR sum to use in (1) and (2).

A second weakness in the choice of the feedback polynomial emerges when considering *bitwise* linear approximations. Using the same technique as in [6], we can take the 2^{32}th power of the feedback polynomial $p(x) = x^{16}+x^{13}+x^7+\alpha^{-1} \in \mathbb{F}_{2^{32}}[x]$, giving us

$$p^{2^{32}}(x) = x^{16\cdot 2^{32}} + x^{13\cdot 2^{32}} + x^{7\cdot 2^{32}} + \alpha^{-1\cdot 2^{32}} \in \mathbb{F}_{2^{32}}[x]. \tag{3}$$

Since $\alpha \in \mathbb{F}_{2^{32}}$ we have $\alpha^{-1\cdot 2^{32}} = \alpha^{-1}$, and summation of $p(x) + p^{2^{32}}(x)$ yields

$$x^{16\cdot 2^{32}} + x^{13\cdot 2^{32}} + x^{7\cdot 2^{32}} + x^{16} + x^{13} + x^7. \tag{4}$$

Dividing (4) with x^7 gives us a linear recurrence equation satisfying

$$s_{t+16\cdot 2^{32}-7} + s_{t+13\cdot 2^{32}-7} + s_{t+7\cdot 2^{32}-7} + s_{t+9} + s_{t+6} + s_t = 0 \tag{5}$$

In (5) we have derived a linear recurrence equation that holds for *each single bit position*. Hence, any bitwise correlation found in the FSM can be turned into a

distinguishing attack. In a recent paper by Coppersmith, Halevi and Jutla [1], they find such a correlation and for the resulting distinguishing attack they need about 2^{95} words of output and the computational complexity is about 2^{100}. By computer search we have also found other smaller correlations, often involving similar bit positions. The strong correlations seem to be caused by an interaction between the permutation in the S-box and the cyclic shift by 7 in the FSM.

Even if this indeed is a security flaw unpredicted by the authors, one could argue about the relevance of such a distinguishing attack. Using e.g. AES (or any other block cipher with block length 128 bits) in counter mode, there is an almost trivial distinguishing attack after seeing about 2^{64} ciphertext blocks. However, as we regard security as the outmost important design criteria, we addressed all known weaknesses in SNOW 1.0 and propose a slightly modified design, SNOW 2.0.

4 SNOW 2.0

As we now turn to describe the new version SNOW 2.0, we want to emphasize that the notations from the previous sections are no longer valid and will be redefined in the following. The new version is schematically a small modification of the original construction, see Figure 2. The word size is unchanged (32 bits) and the LFSR length is again 16, but the feedback polynomial has been changed. The Finite State Machine (FSM) has two input words, taken from the LFSR, and the running key is formed as the XOR between the FSM output and the last element of the LFSR, as done in SNOW 1.0. The operation of the cipher is as follows. First, a key initialization is performed. This operation provides the LFSR with a starting state as well as giving the internal FSM registers $R1$ and $R2$ there initial values. Next the cipher is clocked once and the first keystream symbol is read out[1]. Then the cipher is clocked again and the second keystream symbol is read, etcetera.

Let us give a detailed description of the cipher, starting with the LFSR. The main reason for the specific feedback polynomial chosen in SNOW 1.0, was to have a fast realization in software. By choosing a multiplication with the same primitive element as the base is constructed from, we can realize the multiplication with just one left shift and a possible XOR with a known pattern. However, this choice opens up possible weaknesses, as discussed in Section 3. In SNOW 2.0, we have two different elements involved in the feedback loop, α and α^{-1}, where α now is a root of a primitive polynomial of degree 4 over $\mathbb{F}_{2^8}$. To be more precise, the feedback polynomial of SNOW 2.0 is given by

$$\pi(x) = \alpha x^{16} + x^{14} + \alpha^{-1} x^5 + 1 \in \mathbb{F}_{2^{32}}[x], \tag{6}$$

where α is a root of $x^4 + \beta^{23} x^3 + \beta^{245} x^2 + \beta^{48} x + \beta^{239} \in \mathbb{F}_{2^8}[x]$,
and β is a root of $x^8 + x^7 + x^5 + x^3 + 1 \in \mathbb{F}_2[x]$.

[1] Observe the change from the original version where the first symbol was read out *before* the cipher was clocked.

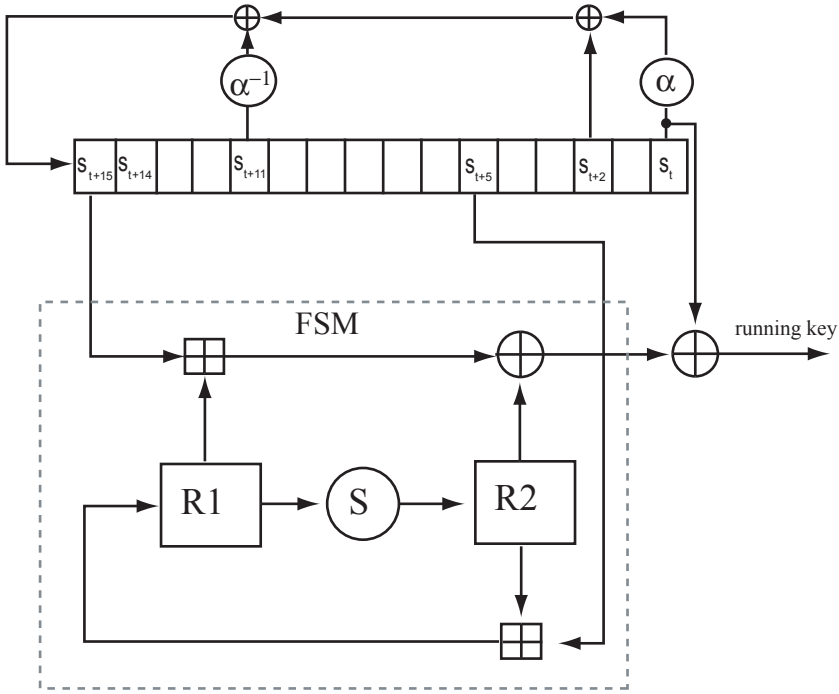

Fig. 2. A schematic picture of SNOW 2.0

Let the state of the LFSR at time $t \geq 0$ be denoted $(s_{t+15}, s_{t+14}, \ldots, s_t)$, $s_{t+i} \in \mathbb{F}_{2^{32}}$, $i \geq 0$. The element s_t is the rightmost element (or first element to exit) as indicated in Figure 2, and the sequence produced by the LFSR is $(s_0, s_1, s_2, \ldots)$. By time $t = 0$, we mean the time instance directly after the key initialization. Then the cipher is clocked once before producing the first keystream symbol, i.e., the first keystream symbol, denoted z_1, is produced at time $t = 1$. The produced keystream sequence is denoted $(z_1, z_2, z_3, \ldots)$.

The Finite State Machine (FSM) has two registers, denoted $R1$ and $R2$, each holding 32 bits. The value of the registers at time $t \geq 0$ is denoted $R1_t$ and $R2_t$ respectively. The input to the FSM is (s_{t+15}, s_{t+5}) and the output of the FSM, denoted F_t, is calculated as

$$F_t = (s_{t+15} \boxplus R1_t) \oplus R2_t, \quad t \geq 0 \tag{7}$$

and the keystream is given by

$$z_t = F_t \oplus s_t, \quad t \geq 1. \tag{8}$$

Here we use the notation $\boxplus$ for integer addition modulo 2^{32} and $\oplus$ for bitwise addition (XOR). The registers $R1$ and $R2$ are updated with new values according to

$$R1_{t+1} = s_{t+5} \boxplus R2_t \quad \text{and} \tag{9}$$
$$R2_{t+1} = S(R1_t) \quad t \geq 0. \tag{10}$$

4.1 The S-box

The S-box, denoted by $S(w)$, is a permutation on $\mathbb{Z}_{2^{32}}$ based on the round function of Rijndael [4]. Let $w = (w_3, w_2, w_1, w_0)$ be the input to the S-box, where $w_i, i = 0...3$ is the four bytes of w. Assume w_3 to be the most significant byte. Let

$$w = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} \tag{11}$$

be a vector representation of the input to the S-box. First we apply the Rijndael S-box, denoted S_R to each byte, giving us the vector

$$\begin{pmatrix} S_R[w_0] \\ S_R[w_1] \\ S_R[w_2] \\ S_R[w_3] \end{pmatrix}. \tag{12}$$

In the *MixColumn transformation* of Rijndael's round function, each 4 byte word is considered a polynomial in y over $\mathbb{F}_{2^8}$, defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]$. Each word can be represented by a polynomial of at most degree 3. Next we consider the vector in (12) as representing a polynomial over $\mathbb{F}_{2^8}$ and multiply with a fixed polynomial $c(y) = (x+1)y^3 + y^2 + y + x \in \mathbb{F}_{2^8}[y]$ modulo $y^4 + 1 \in \mathbb{F}_{2^8}[y]$. This polynomial multiplication can (as done in Rijndael) be computed as a matrix multiplication,

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \begin{pmatrix} S_R[w_0] \\ S_R[w_1] \\ S_R[w_2] \\ S_R[w_3] \end{pmatrix}, \tag{13}$$

where (r_3, r_2, r_1, r_0) are the output bytes from the S-box. These bytes are concatenated to form the word output from the S-box, $r = S(w)$.

4.2 Key Initialization

SNOW 2.0 takes two parameters as input values; a secret key of either 128 or 256 bits and a publicly known 128 bit *initialization variable* IV. The IV value is considered as a four word input $IV = (IV_3, IV_2, IV_1, IV_0)$, where IV_0 is the least significant word. The possible range for IV is thus $0 \ldots 2^{128} - 1$. This means that for a given secret key K, SNOW 2.0 implements a *pseudo-random length-increasing* function from the set of IV values to the set of possible output

sequences. The use of a IV value is optional and applications requiring a IV value typically reinitialize the cipher frequently with a fixed key but the IV value is changed. This could be the case if two parties agreed on a common secret key but wish to communicate multiple messages, e.g. in a frame based setting. Frequent reinitialization could also be desirable from a resynchronization perspective in e.g. a radio based environment.

The key initialization is done as follows. Denote the registers in the LFSR by $(s_{15}, s_{14}, \ldots, s_0)$ from left to right in Figure 2. Thus, s_{15} corresponds to the element holding s_{t+15} during normal operation of the cipher. Let the secret key be denoted by $K = (k_3, k_2, k_1, k_0)$ in the 128 bit case and by $K = (k_7, k_6, k_5, k_4, k_3, k_2, k_1, k_0)$ in the 256 bit case, where each k_i is a word and k_0 is the least significant word. First, the shift register is initialized with K and IV according to

$$\begin{array}{llll} s_{15} = k_3 \oplus IV_0, & s_{14} = k_2, & s_{13} = k_1, & s_{12} = k_0 \oplus IV_1, \\ s_{11} = k_3 \oplus \mathbf{1}, & s_{10} = k_2 \oplus \mathbf{1} \oplus IV_2, & s_9 = k_1 \oplus \mathbf{1} \oplus IV_3, & s_8 = k_0 \oplus \mathbf{1}, \end{array}$$

and for the second half,

$$\begin{array}{llll} s_7 = k_3, & s_6 = k_2, & s_5 = k_1, & s_4 = k_0, \\ s_3 = k_3 \oplus \mathbf{1}, & s_2 = k_2 \oplus \mathbf{1}, & s_1 = k_1 \oplus \mathbf{1}, & s_0 = k_0 \oplus \mathbf{1}, \end{array}$$

where $\mathbf{1}$ denotes the all one vector (32 bits).

In the 256 bit case, the LFSR initialization is correspondingly,

$$\begin{array}{llll} s_{15} = k_7 \oplus IV_0, & s_{14} = k_6, & s_{13} = k_5, & s_{12} = k_4 \oplus IV_1, \\ s_{11} = k_3, & s_{10} = k_2 \oplus IV_2, & s_9 = k_1 \oplus IV_3, & s_8 = k_0, \\ s_7 = k_7 \oplus \mathbf{1}, & s_6 = k_6 \oplus \mathbf{1} & \ldots\ , & s_0 = k_0 \oplus \mathbf{1}. \end{array}$$

After the LFSR has been initialized, R1 and R2 are both set to zero. Now, the cipher is clocked 32 times without producing any output symbols. Instead, the output of the FSM is incorporated in the feedback loop, see Figure 3. Thus, during the 32 clocks in the key initialization, the next element to be inserted into the LFSR is given by

$$s_{t+16} = \alpha^{-1} s_{t+11} \oplus s_{t+2} \oplus \alpha s_t \oplus F_t. \tag{14}$$

After the 32 clockings the cipher is shifted back to normal operation (Figure 2) and clocked once before the first keystream symbol is produced. The maximum number of keystream words allowed is set to 2^{50}, then the cipher must be rekeyed. This limit provides a bound for cryptanalysis and implies no practical limits to the operation of the cipher. The need for producing more than 2^{50} words using the same key, is quite unlikely.

5 Design Differences from SNOW 1.0

In this section we highlight the differences between SNOW 2.0 and SNOW 1.0 and their expected security improvements. We start with the choice of feedback

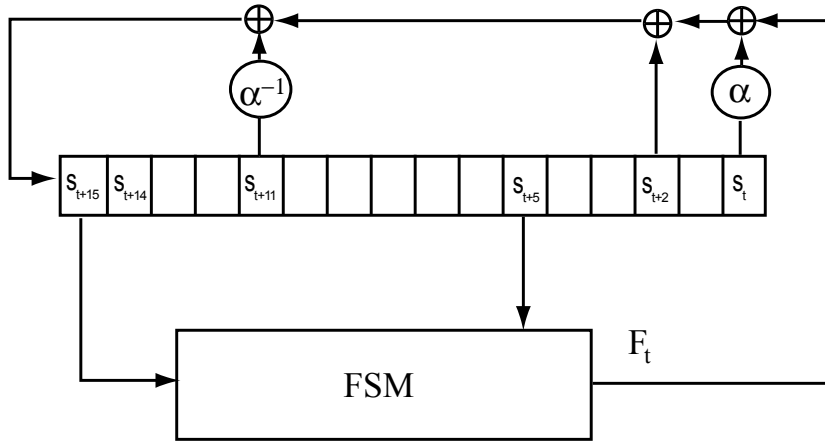

Fig. 3. Cipher operation during key initialization.

polynomial. In SNOW 1.0 the multiplication could be implemented by a single left shift of the word followed by a possible XOR with a known pattern of weight 6. This means that the resulting word was in many positions only a shift of the original word. In SNOW 2.0 we define $\mathbb{F}_{2^{32}}$ as an extension field over $\mathbb{F}_{2^8}$ and each of the two multiplications can be implemented as a *byte* shift together with a unconditional XOR with one of 256 possible patterns. This results in a better spreading of the bits in the feedback loop, and improves the resistance against certain correlation attack, as discussed in [6]. The use of *two* constants in the feedback loop also improves the resistance against bitwise linear approximation attacks, as discussed in Section 3. To the authors, there is no known method to manipulate the feedback polynomial such that the resulting linear recurrence hold for each bit position and have reasonably low weight. The unconditional XOR also seems to improve speed, by removing the possible branch prediction error in a pipelined processor.

The FSM in SNOW 2.0 now takes *two* inputs. This makes a guess-and-determine type of attack more difficult. Given the output of the FSM, together with $R1$ and $R2$ it is no longer possible to deduce the next FSM state directly. The update of $R1$ does not depend on the output of the FSM, but on a word taken from the LFSR. This also suggests that similar correlations as those found in [1] would be much weaker.

The S-box in SNOW 1.0 was also byte oriented but the final bit permutation did not diffuse as much as the new design. In SNOW 1.0, each input byte to the S-box affected only 8 bits of the output word. The choice of the new S-box, based on the round function of Rijndael, provides a much stronger diffusion. Each output bit now depends on each input bit.

6 Implementation Aspects

The design of SNOW 2.0 was done with a fast software implementation in mind. We have chosen a minimum number of different operations; XOR, integer addition, byte shift of a word, and table lookups, all available on modern processors. Even though there are many possible tradeoffs in a software implementation, we will discuss some of the design aspects which have high impact in software.

We start with the LFSR. The field $\mathbb{F}_{2^{32}}$ is defined as an extension field over $\mathbb{F}_{2^8}$, with $\alpha \in \mathbb{F}_{2^{32}}$ being the root of the degree 4 polynomial

$$x^4 + \beta^{23}x^3 + \beta^{245}x^2 + \beta^{48}x + \beta^{239} \in \mathbb{F}_{2^8}[x] \tag{15}$$

Hence, we have the degree reduction of α given by

$$\alpha^4 = \beta^{23}\alpha^3 + \beta^{245}\alpha^2 + \beta^{48}\alpha + \beta^{239} \tag{16}$$

In the feedback loop, multiplication with α and α^{-1} can be implemented as a simple byte shift plus an additional XOR with one of 256 possible patterns. This can be seen from the representation of a word as a polynomial in $\mathbb{F}_{2^8}[x]$ using $(\alpha^3, \alpha^2, \alpha, 1)$ as base. Thus, any element w in $\mathbb{F}_{2^{32}}$ can be written as

$$w = c_3\alpha^3 + c_2\alpha^2 + c_1\alpha + c_0, \tag{17}$$

where (c_3, c_2, c_1, c_0) are the bytes of w, c_0 being the least significant byte. Multiplying w with α, will yield a reduction according to (16) as follows

$$\alpha w = c_3\alpha^4 + c_2\alpha^3 + c_1\alpha^2 + c_0\alpha \tag{18}$$

$$= (c_3\beta^{23} + c_2)\alpha^3 + (c_3\beta^{245} + c_1)\alpha^2 + (c_3\beta^{48} + c_0)\alpha + c_3\beta^{239}. \tag{19}$$

Similar calculations can be done for the multiplication with α^{-1}. Thus, to get a fast implementation of the LFSR feedback, one can use precomputed tables

$$MUL_\alpha[c] = (c\beta^{23}, c\beta^{245}, c\beta^{48}, c\beta^{239}) \tag{20}$$

$$MUL_{\alpha^{-1}}[c] = (c\beta^{16}, c\beta^{39}, c\beta^{6}, c\beta^{64}), \tag{21}$$

where c runs through all elements in $\mathbb{F}_{2^8}$. The pseudo-code for the multiplication would be

```
// Multiplication w*alpha ("<<" is left shift,">>" is right shift)
result=(w<<8) XOR MUL_a[w>>24];
// Multiplication  w*alpha^-1
result=(w>>8) XOR MUL_ainverse[w and 0xff];
```

The S-box are implemented using the same techniques as done in Rijndael [4] and SCREAM [2]. Recall the expression for the S-box, $r = S(w)$

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \begin{pmatrix} S_R[w_0] \\ S_R[w_1] \\ S_R[w_2] \\ S_R[w_3] \end{pmatrix}. \tag{22}$$

The matrix multiplication can be split up into a linear combinations of the columns

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = S_R[w_0] \begin{pmatrix} x \\ 1 \\ 1 \\ x+1 \end{pmatrix} + S_R[w_1] \begin{pmatrix} x+1 \\ x \\ 1 \\ 1 \end{pmatrix} + S_R[w_2] \begin{pmatrix} 1 \\ x+1 \\ x \\ 1 \end{pmatrix} + S_R[w_3] \begin{pmatrix} 1 \\ 1 \\ x+1 \\ x \end{pmatrix}.$$

By using four tables of words, each of size 256, defined by

$$T_0[a] = \begin{pmatrix} xS_R[a] \\ S_R[a] \\ S_R[a] \\ (x+1)S_R[a] \end{pmatrix}, T_1[a] = \begin{pmatrix} (x+1)S_R[a] \\ xS_R[a] \\ S_R[a] \\ S_R[a] \end{pmatrix},$$

$$T_2[a] = \begin{pmatrix} S_R[a] \\ (x+1)S_R[a] \\ xS_R[a] \\ S_R[a] \end{pmatrix}, T_3[a] = \begin{pmatrix} S_R[a] \\ S_R[a] \\ (x+1)S_R[a] \\ xS_R[a] \end{pmatrix},$$

we can easily implement the S-box by addressing the tables with the bytes (w_3, w_2, w_1, w_0) of the input word w. In pseudo-code we can write

```
// Calculate r=S-box(w)
r=T0[byte0(w)] XOR T1[byte1(w)] XOR T2[byte2(w)] XOR T3[byte3(w)];
```

where `byte0(w)` means the least significant byte of w, etcetera.

We have two different C implementations, both using tables for feedback multiplication and S-box operations. The first version (*version 1*) implements the LFSR with an array using the *sliding window* technique, see e.g. [10]. This version is considered an "easy to read" standard reference version. The second version (*version 2*) implements the cipher with "hard coded" variables for the LFSR. This version produces $16 \cdot 32 = 512$ bits of keystream in each procedure call, corresponding to 16 consecutive clockings. Table 1 indicates the speed of the two implementations versions. For the key setup in SNOW 1.0, the IV mode is used as reference, since it also uses 32 clockings in the initialization phase. This accounts for a more reasonable comparison. The tests where run on an PC with Intel 4 processor running at 1.8GHz, 512 Mb of memory. Each program was compiled using gcc with optimization parameter "-O3" and *inline* directives in the code.

7 Conclusions

We have proposed a new stream cipher SNOW 2.0. The design is based on the NESSIE proposal SNOW 1.0 and addresses all weaknesses found in the original

Operation	SNOW 1.0		SNOW 2.0	
	version 1	version 2	version 1	version 2
Key setup	925	-	937	-
Keystream generation	47	34	38	18

Table 1. Number of cycles needed for key setup and cycles per word for keystream generation on a Pentium 4 @1.8GHz.

construction. The implementation is easier and encryption is faster than SNOW 1.0. Typical encryption speed is over 3Gbits/sec on a Intel Pentium 4 running at 1.8GHz.

A complete description of SNOW 2.0 was given and the design differences from SNOW 1.0 and how they apply to the known attacks were discussed. Some implementation aspects of the new design were discussed, in particular how to get a fast implementation of the LFSR and the S-box.

References

1. D. Coppersmith, S. Halevi, C. Jutla, "Cryptanalysis of stream ciphers with linear masking", To appear in *Advances in Cryptology - CRYPTO 2002*, Lecture Notes in Computer Science, Springer, 2002.
2. D. Coppersmith, S. Halevi, C. Jutla, "Scream: a software-efficient stream cipher", In *Fast Software Encryption (FSE) 2002*, Lecture Notes in Computer Science, vol. 2365, Springer 2002, 195-209.
3. D. Coppersmith, P. Rogaway, "Software-efficient pseudorandom function and the use thereof for encryption", US Patent 5,454,039, 1995.
4. J. Daemen, V. Rijmen, "The design of Rijndael", Springer Verlag Series on Information Security and Cryptography, Springer Verlag, 2002, ISBN 3-540-42580-2.
5. P. Ekdahl, T. Johansson, "SNOW - a new stream cipher", *Proceedings of first NESSIE Workshop*, Heverlee, Belgium, 2000.
6. P. Ekdahl, T. Johansson, "Distinguishing attacks on SOBER", In *Fast Software Encryption (FSE) 2002*, Lecture Notes in Computer Science, vol. 2365, Springer 2002, 210-224.
7. P. Hawkes, "Guess-and-determine attacks on SNOW", private correspondence, 2002.
8. P. Hawkes, G. Rose, "Guess-and-determine attacks on SNOW", Preproceedings of Selected Areas in Cryptography (SAC), August 2002, St John's, Newfoundland, Canada.
9. P. Hawkes, G. Rose "Primitive Specification and supportion documentation for SOBER-t16 submission to NESSIE", *Proceedings of first NESSIE Workshop*, Heverlee, Belgium, 2000.
10. P. Hawkes, G. Rose "Primitive Specification and supportion documentation for SOBER-t32 submission to NESSIE", *Proceedings of first NESSIE Workshop*, Heverlee, Belgium, 2000.
11. L. Knudsen, W. Meier, B. Preneel, V. Rijmen, S. Verdoolaege, "Analysis methods for (alleged) RC4", *Lecture Notes in Computer Science*, vol. 1514 , pp. 327–341., (Asiacrypt'98).

12. I. Mantin, A. Shamir, "A practical attack on RC4", In *Fast Software Encryption (FSE) 2001*, Lecture Notes in Computer SCience, vol. 2355, Springer 2002.
13. A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
14. R. Rivest, "The RC4 encryption algorithm", RSA Data Security, Inc. Mar. 1992.
15. P. Rogaway, D. Coppersmith, "A software optimized encryption algorithm". *Journal of Cryptology*, 11(4):273-287, 1998.
16. D. Watanabe, S. Furuya, H. Yoshida, B. Preneel, "A new keystream generator MUGI", In *Fast Software Encryption (FSE) 2002*, Lecture Notes in Computer Science, vol. 2365, Springer 2002, 179-194.
17. M. Zhang, C. Caroll, A. Chan, "The software-oriented stream cipher SSC2", In *Fast Software Encryption (FSE) 2000*, Lecture Notes in Computer Science, vol. 1978, Springer 2001, 31-48.

8 Appendix A. Test Vectors

```
Test vectors for SNOW 2.0, 128 bit key
Each key is given in bigendian format (MSB...LSB) in hexadecimal
==================

    (IV3,IV2,IV1,IV0)=(0,0,0,0)
    key=80000000000000000000000000000000
Keystream output 1...5:
      keystream=8D590AE9
      keystream=A74A7D05
      keystream=6DC9CA74
      keystream=B72D1A45
      keystream=99B0A083
==================

    (IV3,IV2,IV1,IV0)=(0,0,0,0)
    key=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Keystream output 1...5:
      keystream=E00982F5
      keystream=25F02054
      keystream=214992D8
      keystream=706F2B20
      keystream=DA585E5B
==================

    (IV3,IV2,IV1,IV0)=(4,3,2,1)
    key=80000000000000000000000000000000
Keystream output 1...5:
      keystream=D6403358
      keystream=E0354A69
      keystream=57F43FCE
      keystream=44B4B13F
      keystream=F78E24C2
==================

    (IV3,IV2,IV1,IV0)=(4,3,2,1)
    key=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Keystream output 1...5:
      keystream=C355385D
      keystream=B31D6CBD
      keystream=F774AF53
      keystream=66C2E877
      keystream=4DEADAC7
========== End of test vectors =========
```

```
Test vectors for SNOW 2.0, 256 bit key
Each key is given in bigendian format (MSB...LSB) in hexadecimal
==================
(IV3,IV2,IV1,IV0)=(0,0,0,0)
key=
8000000000000000000000000000000000000000000000000000000000000000
Keystream output 1...5:
      keystream=0B5BCCE2
      keystream=0323E28E
      keystream=0FC20380
      keystream=9C66AB73
      keystream=CA35A680
==================
(IV3,IV2,IV1,IV0)=(0,0,0,0)
key=
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Keystream output 1...5:
      keystream=D9CC22FD
      keystream=861492D0
      keystream=AE6F43FB
      keystream=0F072012
      keystream=078C5AEE
==================
(IV3,IV2,IV1,IV0)=(4,3,2,1)
key=
8000000000000000000000000000000000000000000000000000000000000000
Keystream output 1...5:
      keystream=7861080D
      keystream=5755E90B
      keystream=736F1091
      keystream=6ED519B1
      keystream=2C1A3A42
==================
(IV3,IV2,IV1,IV0)=(4,3,2,1)
key=
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Keystream output 1...5:
      keystream=29261FCE
      keystream=5ED03820
      keystream=1D6AFAF8
      keystream=B87E74FE
      keystream=D49ECB10
========== End of test vectors =========
```