SNOW - a new stream cipher[⋆]

Patrik Ekdahl, Thomas Johansson

Dept. of Information Technology
Lund University, P.O. Box 118, 221 00 Lund, Sweden
{patrik,thomas}@it.lth.se
November 22, 2001

Abstract. In this paper a new word-oriented stream cipher, called SNOW, is proposed. The design of the cipher is quite simple, consisting of a linear feedback shift register, feeding a finite state machine.
The design goals of producing a stream cipher significantly faster than AES, with significantly lower implementation costs in hardware, and a security level similar to AES is currently met. Our fastest C implementation requires under 1 clock cycle per running key bit. The best attacks are generic attacks like an exhaustive key search attack.

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

1 Introduction

Cipher systems are usually subdivided into block ciphers and stream ciphers. Block ciphers tend to simultaneously encrypt groups of characters, whereas stream ciphers operate on individual characters of a plaintext message one at a time.

A binary additive stream cipher is a synchronous stream cipher in which the keystream, the plaintext and the ciphertext are sequences of binary digits. The output of the keystream generator, called the *running key*, $z(1), z(2), \ldots$ is "added" symbolwise to the plaintext sequence $m(1), m(2), \ldots$, producing the ciphertext $c(1), c(2), \ldots$. Each secret key k as input to the keystream generator corresponds to an output sequence. Since the secret key k is shared between the transmitter and the receiver, the receiver can decrypt by subtracting the output of the keystream generator from the ciphertext, obtaining the message sequence.

The goal in stream cipher design is to efficiently produce random-looking sequences that in some sense are "indistinguishable" from truly random sequences. From a cryptanalysis point of view, a good stream cipher should be resistant against a *known-plaintext attack*. In a known-plaintext attack the cryptanalyst is

[⋆] A first version of this paper was published in *Proc. of the first open Nessie Workshop*, Heverlee, Belgium, November 13–14 2000. Some time ago, the NESSIE board invited the submitters of stream ciphers to specify a new mode of operation that accommodates an initialization variable (IV). In this second version of the paper we have added such a mode of operation (and corrected a few misprints).

given a plaintext and the corresponding ciphertext, and the task is to determine the key k. For a synchronous stream cipher, this is equivalent to the problem of finding the key k that produced a given keystream $z(1), z(2), \ldots, z(N)$.

Apart from the security aspects, we would like our cipher to be very fast on different platforms, when implemented in software. Also, it should admit a compact hardware implementation.

This paper contains a proposal for a new stream cipher, called SNOW. In stream cipher design, we usually use linear feedback shift registers, LFSRs, and the secret key is often used to provide a value for the initial state of the LFSRs. Also SNOW is built around a LFSR, but over the alphabet $\mathbb{F}_{2^{32}}$.

SNOW is a word-oriented stream cipher, based on ideas from the classical summation generator. Symbols from the LFSR enters a finite state machine, FSM, and the output word from the FSM is bitwise added to other symbols from the LFSR, producing the running key. It meets the security requirements as well as being fast and fairly simple to implement in hardware.

In the description of SNOW, we specify a mode of operation, called IV mode. This mode allows frequent rekeying through a 64 bit initialization variable (IV). Such a mode has applications in e.g. telecommunication protocols.

Several word-oriented stream ciphers have appeared before, e.g, SEAL [9], SOBER [2], SSC2 [23], and RC4 [20]. However, most of them do not provide full security.

The most important general attacks on LFSR-based stream ciphers are *correlation attacks*. If a correlation between the known output sequence and the output of one internal LFSR can be detected, this can be used in a "divide-and-conquer" attack on that LFSR [21, 18]. In the security analysis of SNOW, most focus is on correlation attacks.

In Section 2 we provide a full description of SNOW. Section 3 gives some aspect on implementations of SNOW, and Section 4 gives a short overview of different methods for cryptanalysis. In Section 5 we describe some design choices and design goals for the cipher, before concluding.

2 A description of SNOW

The proposed stream cipher is a word oriented additive stream cipher, where a word in the specification is chosen to be 32 bits. Furthermore, we assume in the specification a "big-endian" representation for the arithmetics used in the cipher.

The cipher has been developed from ideas around the summation generator [19]. Another cipher that has used such ideas is the stream cipher E_0 in the Bluetooth standard [3].

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. For the moment, we assume that the key initialization is done, and concentrate on the cipher in operation.

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 FSM is shown in Figure 2.

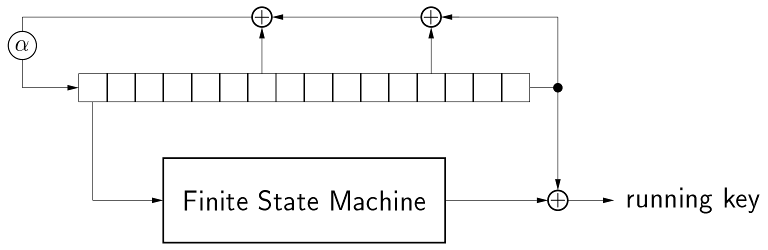

Fig. 1. The generator SNOW

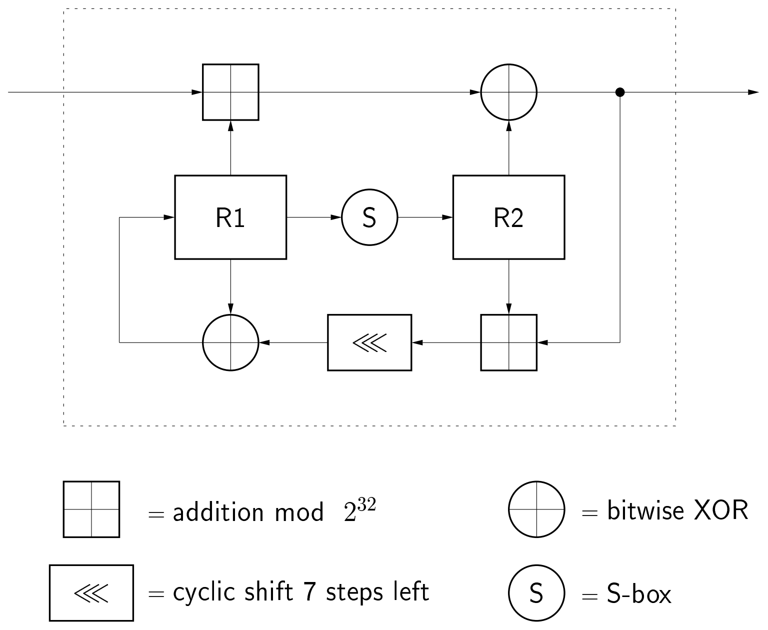

Fig. 2. The Finite State Machine

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 running key 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 running key 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.

Let us give a detailed description of the generator. Returning to Figure 1, the LFSR has a primitive feedback polynomial over $\mathbb{F}_{2^{32}}$ which is

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

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 $s(1), s(2), \dots s(16) \in \mathbb{F}_{2^{32}}$ be the state of the LFSR. Here $s(1)$ is associated with the leftmost memory element in Figure 1, and $s(16)$ with the rightmost. We consider a representation of elements in $\mathbb{F}_{2^{32}}$, using the base $\{\alpha^{31}, \dots, \alpha^2, \alpha, 1\}$, i.e., if $y \in \mathbb{F}_{2^{32}}$ then y is represented by $(y_{31}, y_{30}, \dots, y_1, y_0)$, where

$$y = y_{31}\alpha^{31} + y_{30}\alpha^{30} + \cdots + y_1\alpha + y_0.$$

We consider $y_{31}, y_{30}, \dots$ to be the most significant bits (MSB) and $\dots, y_1, y_0$ to be the least significant bits (LSB).

After the clocking, $s(1)$ is the input to the finite state machine. The output of the FSM, called FSM_{out}, is simply calculated in steps as follows.

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

The output of the FSM is xored with $s(16)$ to form the running key, i.e.,

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

The running key is finally xored with the plaintext, producing the ciphertext.

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

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

Let us explain the notation. By $x \boxplus y$ we mean the integer addition of x and y $\text{mod} 2^{32}$, where $(y_{31}, y_{30}, \dots, y_1, y_0)$ represents the element

$$y_{31} \cdot 2^{31} + y_{30} \cdot 2^{30} + \cdots + y_1 \cdot 2 + y_0 \in \mathbb{Z}_{2^{32}}.$$

The notation $x \lll$ is a cyclic shift of x 7 steps to the left, i.e., $x = (x_{31}, x_{30}, \dots, x_1, x_0)$ is mapped to the value

$$x = (x_{24}, x_{23}, \dots x_0, x_{31}, \dots, x_{26}, x_{25}).$$

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. It works as follows. The input x is split into 4 bytes, from most significant to least significant byte. Each of the bytes enters a nonlinear mapping from 8 bits to 8 bits.

Let the input to the nonlinear mapping be $w = (w_7, w_6, \ldots, w_0)$ and let the output be $r = (r_7, r_6, \ldots, r_0)$. Both vectors are considered as representing elements in $\mathbb{F}_{2^8}$ using the polynomial base $\{\beta^7, \ldots, \beta, 1\}$ generated by the irreducible polynomial $\pi(x) = x^8 + x^5 + x^3 + x + 1$ and $\pi(\beta) = 0$. The nonlinear mapping is defined to be

$$r = w^7 + \beta^2 + \beta + 1,$$

where the arithmetics are in $\mathbb{F}_{2^8}$.

After the mapping above has been applied to each byte, the bits in the resulting word are permuted. The permutation is described by

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
3	10	20	24	0	14	17	29	7	13	18	25	5	12	23	27

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	8	21	26	4	9	19	31	2	11	16	28	6	15	22	30

which should be interpreted as the 31:st bit position is mapped to the 3:d, the 30:th bit is mapped to the 10:th, etcetera. Using a vector notation, this can be written as

$$y = (y_{31}, y_{30}, y_{29}, \ldots, y_1, y_0) \rightarrow (y_8, y_0, y_{24}, \ldots, y_{15}, y_{27}).$$

The S-box is shown in Figure 3, where $y = S(x)$ and $\gamma = \beta^2 + \beta + 1$.

2.1 Modes of operation

Two different modes of operation are specified for SNOW. The two modes are referred to as standard mode and IV mode, respectively.

Standard mode: In standard mode SNOW implements a fast cryptographic pseudorandom number generator. This means that for each seed, which in this case is a secret key denoted by k, SNOW outputs a pseudorandom number sequence.

IV mode: In IV mode the generator is initialized using two variables, the secret key k and a known initialization variable (IV). This means that for a given secret key k, the generator now produces a set of pseudorandom number sequences, one for each IV value. Since the produced sequences are to be indistinguishable from truly random sequences in all aspects, the IV mode of SNOW can be said to implement a *length-increasing pseudorandom function* (from the set of IV values to the set of possible sequences). The length of the output sequences is usually larger than the IV length.

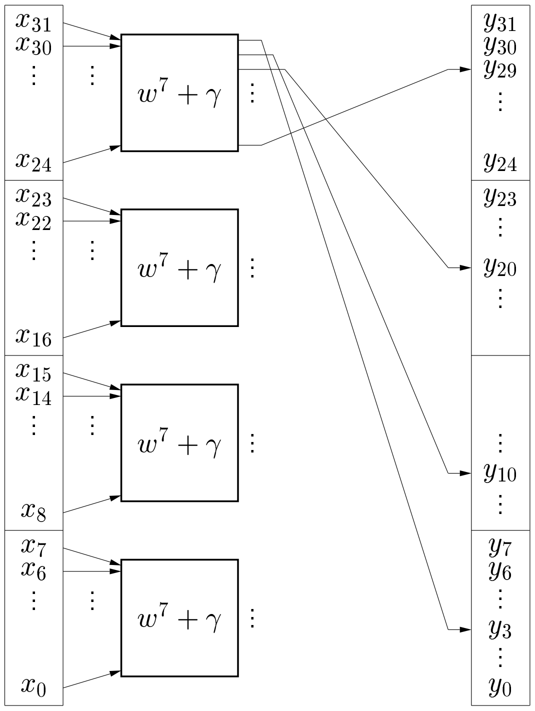

Fig. 3. The S-box $y = S(x)$.

In SNOW, the IV value is a 64 bit value, represented by the two 32 bit words (IV_2, IV_1). The IV value thus range from 0 to $2^{64} - 1$, where IV_2 is the most significant word and IV_1 is the least significant word.

Applications that use the IV mode typically do frequent reinitializations where the key k is fixed but the IV value is changed. We can think of several scenarios where IV mode is desirable.

- If the sender and the receiver have established a common key and wish to communicate multiple messages, IV mode might be suitable.
- In telecommunication scenarios, it is common to divide data into shorter frames (packets) which are sent consecutively. The frames include a frame number. In order to recover from loss of synchronization an IV mode of operation is necessary (or packets arriving out of order).
- In some applications it is desirable to have a stream cipher that enables it to efficiently seek to arbitrary locations in its keystream. This was the main motivation behind the design of the stream cipher LEVIATHAN [17]. Here we just want to point out that this comes for free in the IV mode. In order to be able to efficiently seek arbitrary positions we define the keystream as follows. We first fix a suitable length N. For a fixed k, we then produce N words of output using IV=0, then another N words using IV=1, another N

words using IV=2, etc. When we want to seek a position, we simply start at the corresponding IV value, initialize the generator, and possibly produce a few words before we come to the desired location. The length N should be chosen such that the frequent reinitializations only marginally decreases the performance, but still allows a fast seeking.

Since the IV mode will use frequent reinitializations, the performance of the key initialization will be an important performance parameter. Hence, the key initialization in the IV mode uses less SNOW clockings than in the standard mode (32 versus 64).

Finally, one could consider removing the definition of standard mode and only use IV mode. However, since the test vectors etc. use standard mode we choose to keep it in the description.

2.2 Key initialization

Let the secret key k be denoted by $k = (k(1), k(2), k(3), k(4))$ in the 128 bit case and $k = (k(1), k(2), k(3), k(4), k(5), k(6), k(7), k(8))$ in the 256 bit case.

The key initialization is done as follows. The LFSR is first initialized with the key. In the 128 bit case, the LFSR initialization is

$$\begin{array}{llll} s(1) = k(1) \oplus IV_1, & s(2) = k(2), & s(3) = k(3), & s(4) = k(4) \oplus IV_2, \\ s(5) = k(1) \oplus \mathbf{1}, & s(6) = k(2) \oplus \mathbf{1}, & s(7) = k(3) \oplus \mathbf{1}, & s(8) = k(4) \oplus \mathbf{1}, \end{array}$$

and for the second half,

$$\begin{array}{llll} s(9) = k(1), & s(10) = k(2), & s(11) = k(3), & s(12) = k(4), \\ s(13) = k(1) \oplus \mathbf{1}, & s(14) = k(2) \oplus \mathbf{1}, & s(15) = k(3) \oplus \mathbf{1}, & s(16) = k(4) \oplus \mathbf{1}, \end{array}$$

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

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

$$\begin{array}{llll} s(1) = k(1) \oplus IV_1, & s(2) = k(2), & s(3) = k(3), & s(4) = k(4) \oplus IV_2, \\ s(5) = k(5), & s(6) = k(6), & s(7) = k(7), & s(8) = k(8), \\ s(9) = k(1) \oplus \mathbf{1}, & \quad \ldots\ , & & s(16) = k(8) \oplus \mathbf{1}. \end{array}$$

In standard mode, we assume $IV_1 = IV_2 = 0$. After the LFSR has been initialized, R1 and R2 are both set to zero.

Then the cipher is clocked exactly v times without producing any running key. Instead, the output of the finite state machine is fed back into the feedback loop of the LFSR, as shown in Figure 4. In standard mode $v = 64$, and in IV mode $v = 32$. In one clock cycle, the next value of $s(1)$, here called newS(1), is given by

$$\text{newS}(1) = \alpha(s(7) \oplus s(13) \oplus s(16) \oplus FSM_{\text{out}}).$$

After v clockings, the LFSR and the two registers R1, R2 have received its values from the initialization phase. The first 32 bits of the running key are now available as $FSM_{\text{out}} \oplus s(16)$.

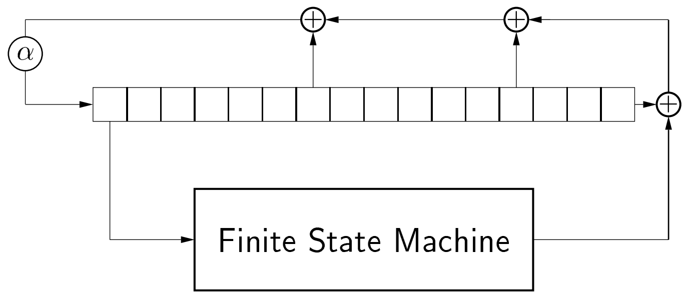

Fig. 4. The key initialization

The maximum allowed length of the running key (output sequence) is set to 2^{50} words. Then the cipher must be rekeyed. This limit is mainly set to provide a maximum length in cryptanalysis. Generating a running key of length greater than 2^{50} words in practice is quite unlikely to happen.

Finally, we note that with very small modifications, one could allow a variable key size in SNOW. The key size could be in the range 128-256 bits, but should not be significantly larger than 256 bits. This because the state space of the cipher has been chosen to match a maximum key size of 256 bits.

3 Implementation

The design of the cipher is chosen such that the implementation is simple and fast. The implementation of the finite state machine is more of a straight-forward kind, whereas the implementation of the LFSR may be done in several ways. Let us first focus on a software implementation.

3.1 Fast software implementations

We mention three different ways to implement the LFSR sequence generation.

1. "Adding and shifting"
2. "Pointer and circular buffer"
3. "Hardcoded LFSR in the program code"

In 1., we simply calculate the next value for $s(1)$, and shift all the other values in the array representing the LFSR states. This is not efficient in software and we do not recommend to implement in this way.

The approach in 2. is based on the idea of having a pointer pointing at the beginning of the LFSR in memory. When we clock the LFSR once we do not shift all the values one step in memory, but rather, we only move the pointer one position. This gives a compact code description of the LFSR sequence generation, faster than the approach in 1. The drawback is the overhead that comes from handling the pointer and the circular buffer.

The third approach is the one we suggest to use for fast implementations. The idea is essentially the same as in 2., but here we do not use pointer(s). Instead, we "hardcode" the LFSR sequence generation in the program code. This will increase the size of the code, but will be much faster. Let us provide an illustrating example of how this is coded. Assume that we have 16 words (variables) in memory representing the LFSR in a circular manner, declared as `s1,s2,...s16`. Let `FSM(s)` be a function producing the output of the finite state machine with input `s`, using two global variables `R1, R2`. The program code would be of the form

```
key1 = FSM(s1)⊕s16;
s16 = (s16⊕s13⊕s7)·α;
key2 = FSM(s16)⊕s15;
s15 = (s15⊕s12⊕s6)·α;
...
key16 = FSM(s2)⊕s1;
s1 = (s1⊕s14⊕s8)·α;
```

The code would produce 16 words of the running key, `key1,key2,...,key16`.

We further note that the multiplication by α above, described as `x`$\cdot\alpha$, is easily implemented in software. Let $x \in \mathbb{F}_{2^{32}}$, $x = (x_{31}, x_{30}, \ldots, x_0)$. Then

$$\alpha x = (x_{30}, x_{29}, \ldots, x_0, 0) \oplus x_{31} \cdot (00100000000100001000010000000011).$$

The finite state machine `FSM(s)` is implemented straight-forward. The operation that will be the most expensive is the S-box operation. We suggest to construct four 256 word tables $T1, T2, T3, T4$ which contain the result of each of the nonlinear mappings after the permutation. Then the result of the S-box can be obtained by addressing the four tables with the corresponding byte of the input, and then xoring the four resulting words.

```
temp = (s⊞R1)⊕R2;
tempR1 = ((R2⊞temp)«7)⊕R1;
R2 = T1[R1&377]⊕T2[(R1»8)&377]⊕T3[(R1»16)&377]⊕T4[(R1»24)&377];
R1 = tempR1;
return temp;
```

Here `x « y` means a left cyclic shift of `x` by `y` steps, and similar for the right shift `x » y`, not necessarily cyclic.

We note that the generation of the next symbol of the LFSR sequence and the generation of the FSM output are independent. Hence, these things can be done totally in parallel, if the processor allows it.

There are also other possibilities to improve the performance in various situations. We might for example use a larger memory size for the LFSR sequence generation, and produce a large number of LFSR output symbols stored in memory. Then we apply the FSM to them one by one and produce the running key.

In the S-box, we may generate two tables of size 2^{16} each instead of the four tables given before. This would increase performance if memory access to the tables is fast.

In Table 1 we give some performance values for different platforms. The C program giving these values is based on approach 3. in Section 3.1. Handoptimizing this program is expected to provide some additional improvements. Since the key initialization in IV mode consists of clocking the cipher 32 times, the time to do key initialization in IV mode is roughly the time this takes. Our current implementation makes key setup in 3μs on a Pentium III 500MHz, but a fast implementation is expected to take about 2μs.

	Processor/Speed		
Operating system/ Compiler	Pentium II 400MHz	Pentium III 500MHz	UltraSPARC 400Mhz
Linux/gcc	420	610	-
WinNT/Microsoft C++	430	520	-
Solaris 2.7/gcc	450	-	430

Table 1. Encryption speed (Mbits/s) on various platforms.

3.2 Hardware implementation

An evaluation of a hardware implementation has not been done, but we note that the cipher has very few components, allowing a compact hardware implementation. The implementation cost will be dominated by the implementation of the 512 bit LFSR, the integer additions, and the S-box. Furthermore, the critical path will be short, including mainly the two integer additions.

The nonlinear mapping in the S-box can be quite efficiently implemented in hardware. The mapping $x \to x^7$ over F_{2^8} can be split into $x \to x^4x^2x$, where x^4, x^2, x are all linear mappings. Hence, we first implement $y = x^4x^2$ and then in another level implementing yx, giving the desired result. Since each implementation level will have terms of degree at most two, the number of gates will be small.

The integer additions could be implemented in standard way, using carry-look-ahead.

4 Security of the cipher

The most important aspect for the cipher is its resistance against different attacks. The design goal has been to provide a security level similar to what is used in block cipher design. In particular, this means that there should not exist an attack significantly faster than an exhaustive key search.

We consider some general attacks on stream ciphers:

Exhaustive key search: This is the most efficient way of attacking SNOW. We exhaustively search the key space of the 128 bit key, or the 256 bit key, respectively.

Time-memory trade-off attacks: This kind of attacks can be applied if the state space of the cipher is too small. It has been a successful way of attacking for example A5 in the GSM standard [8, 1]. Basically, the procedure is as follows. Assume that the cipher is in a certain state, i.e., a certain value for all memory elements in the cipher. Calculate a number of output bits and put the pair (output,state) in a sorted list. Then we scan a received output sequence, hoping to find one of the stored output sequences in the received output sequence. If this occurs, the ciphers state just before producing the found output sequence can also be found in the list, hopefully leading to a successful recovery of the key.

We note that these kind of attacks do not seem to be applicable to SNOW. The state space is simply too large (2^{512+64}) compared to the key size.

Guess-and-Determine attacks: The basic idea is to guess the value of some unknown variables in the cipher, and from the guessed values deduce the value of other unknown variables. Such an attack was for example applied on (alleged) RC4 in [14]. Also here we have found no such attacks on the proposed cipher.

Time-memory trade-off with huge precomputation: Here we just want to point out that if we allow the precomputation complexity to be larger than the complexity of exhaustive key search, it is possible to have time-memory trade-off attacks faster than exhaustive key search, see for example [22]. This is valid for any cipher.

Correlation attacks:

Correlation attacks is a very powerful class of attacks, and probably the most serious threat against the security of the cipher. The basic idea is very simple. We try to determine a correlation between the running key and different linear combinations of the secret key bits. This correlation is then used in a decoding procedure, to recover the secret key. A lot of work has been done on these kind of attacks, see [21, 18, 6, 7, 10, 11, 5, 12, 4]. We should note that almost all this work has been directed towards practical attacks (attacks that could be simulated on a computer), and have mainly considered attacking LFSRs of length less than 100. In this analysis we will be concerned with complexities of up to 2^{256}, and there is no possibility to simulate the performance of different attacks. Hence, we will instead rely on the rather few theoretical results in [18, 13, 5] to calculate the performance.

Now let us examine the proposed cipher in a correlation attack. Our study will be focused on the finite state machine. Denote by

$$\mathbf{u} = u(1), u(2), u(3), \ldots$$

an input sequence to the FSM, and denote by

$$\mathbf{v} = v(1), v(2), v(3), \ldots$$

the corresponding output of the FSM. The goal for us is to find a correlation between $\mathbf{u}$ and $\mathbf{v}$. This correlation could be of any type. We first focus on a binary relation.

Let $u(i) = (u_{31}(i), u_{30}(i), \ldots, u_0(i))$, and the same for $v(i)$, $i = 1, 2, \ldots$. By a binary correlation we mean a linear combination of input and output

$$\sum_{i=1}^{n}\sum_{j=0}^{31} b_{ij}u_j(i) + \sum_{i=1}^{n}\sum_{j=0}^{31} c_{ij}v_j(i),$$

for some constants $b_{ij}, c_{ij} \in \mathbb{F}_2$, $i = 1, 2, \ldots, n$, $j = 0, 1, \ldots, 31$, which is nonuniformly distributed, i.e.,

$$P(\sum_{i=1}^{n}\sum_{j=0}^{31} b_{ij}u_j(i) + \sum_{i=1}^{n}\sum_{j=0}^{31} c_{ij}v_j(i) = 0) = \frac{1}{2} + \epsilon, \tag{1}$$

where $\epsilon > 0$ is called the correlation probability. If we can identify a large enough correlation, we can turn the key recovery problem into a decoding problem, and use known algorithms to solve it.

At a certain moment, let the unknown value in R1 be denoted by r and the unknown value in R2 be denoted by $\hat{r}$. In order to find correlations of the form (1), we must remove the influence of $r, \hat{r}$. Write up a first equation of the form

$$v(1) = (u(1) \boxplus r) \oplus \hat{r}. \tag{2}$$

Let us introduce the notation x' by which we mean x rotated 7 steps to the left. Furthermore, let $c(i)$ denote the carry bits in the integer addition in (2), allowing us to express $u(i) \boxplus r$ as $u(i) \boxplus r = u(i) \oplus r \oplus c(i)$. The equation (2) is rewritten as

$$v(1) = (u(1) \oplus r \oplus c(1)) \oplus \hat{r}. \tag{3}$$

Next, R1 is updated, given the value $(v(1) \boxplus \hat{r})' \oplus r$, and R2 gets the value $S(r)$. Again we introduce $\hat{c}(i)$ to denote the carry bits in the integer addition above. Then R1 will get the value

$$v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r.$$

Now a second equation can be written as

$$v(2) = (u(2) \oplus v(1)' \oplus \hat{r}' \oplus \hat{c}(1)') \oplus c(2) \oplus S(r). \tag{4}$$

Since we have two unknown variables $r, \hat{r}$, we need a third equation. Updating R1, R2 we get that R1 next gets the value

$$v(2)' \oplus S(r)' \oplus \hat{c}(2)' \oplus v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r,$$

and R2 the value

$$S(v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r).$$

Finally, the third equation is given by

$$v(3) = u(3) \oplus v(2)' \oplus S(r)' \oplus \hat{c}(2)' \oplus v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r \oplus c(3) \oplus S(v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r). \tag{5}$$

Summarizing, we have three equations in two unknowns,

$$v(1) = u(1) \oplus r \oplus c(1) \oplus \hat{r}. \tag{6}$$

$$v(2) = u(2) \oplus v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus c(2) \oplus S(r). \tag{7}$$

$$\begin{aligned} v(3) = {} & u(3) \oplus v(2)' \oplus S(r)' \oplus \hat{c}(2)' \oplus v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r \\ & \oplus c(3) \oplus S(v(1)' \oplus \hat{r}' \oplus \hat{c}(1)' \oplus r). \end{aligned} \tag{8}$$

The S-box can also be approximated by a linear expression, $S(x) = Ax + \psi$, where Ax denotes a linear transformation of x and ψ denotes the "noise" introduced by the linear approximation.

We have elaborated with the equations in (6)-(8), and removed the influence from $r, \hat{r}$ in order to obtain correlations. However, no correlations of significant size have been found. By significant we mean a binary linear combination of input and output bits from the FSM which equals zero with a probability, say, at least

$$1/2 + 2^{-20}.$$

Such a correlation would not break the cipher faster than exhaustive key search, but would be a correlation that is stronger than expected by the authors.

Let us now turn the question around, and give the correlation probability that is necessary for a correlation attack faster than exhaustive key search. Here we note that many of the proposed methods for fast correlation attacks are based only on simulations and guesses of performance. However, a few papers do present also asymptotic results, and we focus on the work in [5]. We consider only the key size 256 bits.

We refer to [5] for details on the attack. Here we just review the basic results. With a given LFSR length l, algorithm parameters k, t, the required length N of the running key $\mathbf{z}$ for the algorithm in [5] to succeed is

$$N \approx 1/4 \cdot (2kt! \ln 2)^{1/t} \cdot \epsilon^{-2} \cdot 2^{\frac{l-k}{t}}, \tag{9}$$

when the correlation probability is $1/2 + \epsilon$.

The algorithm collects a number of parity checks in a precomputation phase. These must be stored. The complexity of this precomputation phase is approximately $N^{\lceil (t-1)/2 \rceil}$ and requires memory $N^{\lfloor (t-1)/2 \rfloor}$. The memory can be reduced by increasing the computational complexity. Furthermore, the number of parity checks that need to be stored is roughly $N^t/t! \cdot 2^{-(l-k)}$, and the decoding complexity is 2^k times the number of parity checks.

In our case $l = 512$ and the maximum length is $N = 2^{50}$. Table 2 demonstrates the required correlation to perform an attack faster than exhaustive key

Value of t	Precomp. time	Precomp. memory	Number of parity checks to be stored	Required correlation
2	2^{50}	-	0	-
3	2^{100}	2^{50}	0	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
7	2^{200}	2^{150}	2^{40}	$1/2 + 0.115$ ($k = 215$)
8	2^{200}	2^{200}	2^{63}	$1/2 + 0.050$ ($k = 190$)
9	2^{250}	2^{200}	2^{88}	$1/2 + 0.025$ ($k = 168$)
10	2^{250}	2^{250}	2^{111}	$1/2 + 0.015$ ($k = 145$)
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
16	2^{400}	2^{400}	2^{244}	$1/2 + 0.0029$ ($k = 12$)

Table 2. The required correlation for an attack faster than exhaustive key search

search for different values of the parameter t in the algorithm. Also the precomputation time and memory is given. For $t > 16$, the number of stored parity checks will be larger than 2^{256}, which makes the attack void.

Hence we can conclude the following. In order to perform a binary correlation attack faster than exhaustive key search, one needs to find a correlation between input and output sequences of the FSM, larger than the value $1/2+0.0029$. Still, such an attack would only be academic, requiring around 2^{400} precomputation memory and complexity.

Other binary correlation attacks have appeared, but in an asymptotic behavior they do not perform differently. Another approach would be to consider correlations over a larger field, in our case $\mathbb{F}_{2^8}$ or $\mathbb{F}_{2^{32}}$ could be possible choices.

Distinguishing Attacks:

We mention the possibility of a "distinguishing attack". The attacker is successful if he can distinguish the generated pseudo-random sequences from truly random sequences. From the fact that we add an element of the LFSR sequence, before producing the running key, we believe that it is quite unlikely to find statistical weaknesses that can be exploited. A possibility would be to find a weakness in the key initialization, that would enable such an attack. No such weaknesses have been found.

An important type of distinguishing attacks that must be considered in IV mode is what is referred to as distinguishability in polynomial sampling [15]. Basically, we consider a large number of generated sequences, one for each IV value, and try to establish some kind of bias or dependence between them (anything that distinguishes them from a set of mutually independent random sequences). This type of attack is wonderfully demonstrated in [15], where a bias in the second output word of RC4 is found.

Returning our attention to SNOW, no distinguishability in polynomial sampling has been found.

5 Design choices and design goals

Here we give a summary of the design choices and design goals. Let us start with the latter.

The design goals have been to produce a stream cipher significantly faster than AES, with significantly lower implementation costs in hardware. Most importantly, the security must be maximum, i.e., on a level similar to AES.

Since any block cipher can be used in "stream cipher mode", a proposed stream cipher must be superior in speed and implementation compared to the block cipher. We choose to compare with the upcoming AES. In software, we have demonstrated a speed at least 2-10 times faster than AES (depending on which of the five remaining candidates we compare with). A hardware implementation has not been evaluated, but is expected to be significantly simpler than the AES candidates.

From security point of view, the security is set to maximum. This means in particular, that the best attacks are generic attacks like an exhaustive key search attack. Any proposed stream cipher will have a difficulty in obtaining the same public confidence as a block cipher like AES. However, we hope that the simplicity of the design will motivate research which in turn will increase the confidence.

Now let us give some comments on design choices. The general construction method, as given in Figure 1, was chosen due to the fact that adding the LFSR symbol with the FSM output before producing the running key provides nice statistical properties for the running key. Also, similar ideas have appeared before, but mainly for a binary alphabet. The alphabet size of (at least) 2^{32} was necessary to meet the design goals. The connection polynomial $p(x)$ is primitive and was chosen to support a fast implementation.

Inside the FSM, the use of integer addition and bitwise addition was influenced by the classical summation combiner. However, since we have a large alphabet size, a lot of dependence between consecutive bits in the words occur. This is effectively taken care of by the S-box, which not only introduce additional nonlinearity, but most importantly, permute the positions of the resulting word. The nonlinear mappings in the S-box was chosen as power functions over a finite field, similar to the selection of S-boxes in MISTY [16].

The S-box was chosen to support a simple implementation in both software and hardware, which is the motivation for choosing four 8-to-8 bit S-boxes. In software, using table lookup, each byte will address a table. This can sometimes be more efficiently implemented than 9-to-9 bit S-boxes, etc. The permutation was chosen to spread two bits of each input byte to each of the resulting bytes.

6 Conclusions

The paper has presented the new stream cipher SNOW. A complete description of SNOW, estimates of software performance, and an overview of possible attacks have been given. Several properties of SNOW have not yet been considered, for example hardware implementations, handoptimized code in software, and a more exhaustive treatment of the security. We hope to fill some of these gaps in the near future.

7 Acknowledgements

We acknowledge Ben Smeets and Fredrik Jönsson for all their help during the design phase.

References

1. A. Biryukov, A. Shamir, D. Wagner, "Real time cryptanalysis of A5/1 on a PC", *Preproceeding of Fast Software Encryption Workshop 2000*, (FSE'2000).
2. D. Bleichenbacher, S. Patel,"SOBER cryptanalysis", *Preproceeding of Fast Software Encryption Workshop 99*, (FSE'99), pp 303–314.
3. Bluetooth SIG, *Bluetooth specification*, version 1.0 A.
4. A. Canteaut, M. Trabbia, "Improved fast correlation attacks using parity-check equations of weight 4 and 5", *Advances in Cryptology–EUROCRYPT'2000*, Lecture Notes in Computer Science, vol. 1807, Springer-Verlag, 2000, pp. 573–588.
5. V. Chepyzhov, T. Johansson, and B. Smeets, "A simple algorithm for fast correlation attacks on stream ciphers", *Fast Software Encryption, FSE'2000*, to appear in Lecture Notes in Computer Science, Springer-Verlag, 2000.
6. V. Chepyzhov, and B. Smeets, "On a fast correlation attack on certain stream ciphers", In *Advances in Cryptology–EUROCRYPT'91*, Lecture Notes in Computer Science, vol. 547, Springer-Verlag, 1991, pp. 176–185.
7. A. Clark, J. Golic, E. Dawson, "A comparison of fast correlation attacks", *Fast Software Encryption, FSE'96*, Lecture Notes in Computer Science, Springer-Verlag, vol. 1039, 1996, pp. 145–158.
8. J. Golic, "Cryptanalysis of alleged A5 stream cipher", *Lecture Notes in Computer Science*, vol. 1233 , pp. 239–255., (Eurocrypt'97).
9. D. Coppersmith, P. Rogaway, "Software-efficient pseudorandom function and the use thereof for encryption", US Patent 5,454,039, 1995.
10. T. Johansson, F. Jönsson, "Improved fast correlation attacks on stream ciphers via convolutional codes", *Advances in Cryptology–EUROCRYPT'99*, Lecture Notes in Computer Science, vol. 1592, Springer-Verlag, 1999, pp. 347–362.
11. T. Johansson, F. Jönsson, "Fast correlation attacks based on turbo code techniques", *Advances in Cryptology–CRYPTO'99*, Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999, pp. 181–197.
12. T. Johansson, F. Jönsson, "Fast correlation attacks through reconstruction of linear polynomials", *Lecture Notes in Computer Science*, vol. 1880, pp. 300–315., (Crypto'2000).
13. F. Jönsson, T. Johansson, "Theoretical analysis of a correlation attack based on convolutional codes", *Proc. of 2000 IEEE International Symposium on Information Theory* , p. 212.
14. 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).

15. I. Mantin, A. Shamir, “A practical attack on RC4”, *Preproceeding of Fast Software Encryption Workshop 2001*, (FSE’2001).
16. M. Matsui, Block encryption algorithm MISTY, Technocal report of IEICE, ISEC96-11(1996-07).
17. D. McGrew, S. Fluhrer, “The stream cipher LEVIATHAN, Specification and supporting documentation”, *Proc. of First open Nessie workshop*, see `www.cryptonessie.org`.
18. W. Meier, and O. Staffelbach, “Fast correlation attacks on certain stream ciphers”, *Journal of Cryptology*, 1, pp. 159–176, 1989.
19. A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
20. R. Rivest, “The RC4 encryption algorithm”, RSA Data Security, Inc. Mar. 1992.
21. T. Siegenthaler, “Correlation-immunity of nonlinear combining functions for cryptographic applications”, *IEEE Trans. on Info. Theory*, 30 (1984), pp. 776–780.
22. D. Stinson, *Cryptography, Theory and Practice*, CRC Press.
23. M. Zhang, C. Caroll, A. Chan, “The software-oriented stream cipher SSC2”, *Preproceeding of Fast Software Encryption Workshop 2000*, (FSE’2000).