A new SNOW stream cipher called SNOW-V

Patrik Ekdahl[1], Thomas Johansson[2], Alexander Maximov[1], and Jing Yang[2]

[1] Ericsson Research, Lund, Sweden
{patrik.ekdahl, alexander.maximov}@ericsson.com

[2] Dept. of Electrical and Information Technology, Lund University, Lund, Sweden
{thomas.johansson, jing.yang}@eit.lth.se

Abstract. In this paper we are proposing a new member in the SNOW family of stream ciphers, called SNOW-V. The motivation is to meet an industry demand of very high speed encryption in a virtualized environment, something that can be expected to be relevant in a future 5G mobile communication system. We are revising the SNOW 3G architecture to be competitive in such a pure software environment, making use of both existing acceleration instructions for the AES encryption round function as well as the ability of modern CPUs to handle large vectors of integers (e.g. the Advanced Vector Extensions AVX from Intel). We have kept the general design from SNOW 3G, in terms of linear feedback shift register (LFSR) and Finite State Machine (FSM), but both entities are updated to better align with vectorized implementations. The LFSR part is new and operates 8 times the speed of the FSM. We have furthermore increased the total state size by using 128-bit registers in the FSM, we use the full AES encryption round function in the FSM update, and, finally, the initialization phase includes a masking with key bits at its end. The result is an algorithm generally much faster than AES-256 and with expected security not worse than AES-256.

Keywords: SNOW · Stream Cipher · 5G Mobile System Security.

1 Introduction

Stream ciphers have always played an important part in securing the various generations of 3GPP mobile telephony systems, starting with the GSM system employing the A5 suit of ciphers, continuing with the use of SNOW 3G as the secondary algorithm in UMTS, and more recently as the primary algorithm in LTE, for both integrity and confidentiality. When we now turn to the next generation system, called 5G, we see some fundamental changes in system architecture and security level that in many cases invalidate the previous algorithms. We will focus on the LTE (or 4G, as it is commonly called) system when describing the current state in link protection for mobile systems.

The basis for the link security in all 3GPP generations of mobile telephony systems is a shared secret key between the device (commonly called the User Equipment, UE) and the home network, the Mobile Network Operator that the user has a service agreement with, and from whom the user receives the credentials in form of a UICC with a USIM application (often referred to as the SIM-card). The shared key is stored in the Home Subscriber Server (HSS) and in the Secure Element on the UICC. From this key, through a rather complicated set of key derivations, the home network and the UE both agree on new keys to be used for integrity and confidentiality protection of the control channel, and confidentiality protection of the user data channel. The 4G system defines three different possible algorithms for integrity (EIAx) and confidentiality (EEAx), based on three different primitives SNOW 3G [SAG06], AES [oST01], and ZUC [SAG11]. The algorithms used in UMTS and LTE are all using the 128-bit key size, and are depicted in Table 1.

	UMTS		LTE	
	Integrity	Encryption	Integrity	Encryption
Kasumi	UIA1	UEA1		
SNOW 3G	UIA2	UEA2	EIA1	EEA1
AES			EIA2	EEA2
ZUC			EIA3	EEA3

Table 1: Base algorithms used in UMTS and LTE for integrity and confidentiality.

The SNOW family of stream ciphers started with the SNOW [EJ01] proposal in the European project NESSIE, a call for new primitives. Two attacks [HR02,CHJ02] were soon discovered and the design was subsequently updated to the SNOW 2.0 [EJ02] design. Attacks on SNOW 2.0 will be more discussed in section 3. The ETSI Security Algorithm Group of Experts (SAGE) modified the SNOW 2.0 design and proposed the resulting cipher SNOW 3G as one of the algorithms protecting the air interface in 3GPP telecommunication networks.

Although sufficient for 4G system, these EIA and EEA algorithms face some challenges in the 5G environment. For the 5G system, the 3GPP standardization organization is looking towards increasing the security level to 256-bit key lengths [SA318]. For ExA1, and ExA2, this does not immediately appear to be a problem, since both the underlying primitives (AES and SNOW) are specified for 256-bit keys. ZUC is currently only specified and evaluated under 128-bit key strength, but another version, ZUC-256, supporting 256-bit keys has recently been presented [Bin]. However, since the design of the radio and core network will also fundamentally change in the 5G system, there are other challenges. Many of the network nodes will become virtualized [3GP] and thus the ability to use specialized hardware for the cryptographic primitives will be reduced. Many newer processors from both Intel and ARM now include instructions to accelerate AES, and it will be fairly easy to reach encryption speeds of 20-25 Gbps for EIA2 and EEA2, but for the stream ciphers SNOW and ZUC, we need to look for other solutions. Current benchmarks on SNOW 3G gives approximately 6-7 Gbps in a pure software implementation, which is far too low for the targeted speed of 10 Gbps in the 5G system (see, e.g., [ITU17]).

In this paper we revise the SNOW 2.0/ SNOW 3G architecture to be competitive in a pure software environment, relying on both the acceleration instructions for the AES round function as well as the ability of modern CPUs to handle large vectors of integers (e.g. the Advanced Vector Extensions AVX from Intel). We have kept most of the design from SNOW 3G, in terms of linear feedback shift register (LFSR) and Finite State Machine (FSM), but both entities are updated to better align with vectorized implementations. We have also increased the total state size by going from 32-bit registers to 128-bit registers in the FSM. Each clocking of SNOW-V (V for Virtualization) now produces 128 bits of keystream.

We also propose an AEAD (Authenticated Encryption with Associated Data) operational mode to provide both confidentiality and integrity protection. The keystream width of 128 bits makes the authentication framework of GMAC [Dwo07] very easy to adopt to SNOW-V.

This paper is organized as follows. In section 2, we present the new design, including pseudocode. In section 3, we give a brief security analysis, describing most of the common attack approaches and how they apply to SNOW-V. In section 4, hardware implementation aspects are given and in section 5 the corresponding treatment of software implementations is given. section 6 considers software performance results and implementation aspects using future SIMD instruction set. In section 7 we describe how authentication can be included, in an AEAD mode of operation, and the paper ends with conclusions in section 8.

2 The design

SNOW-V follows the design pattern of previous SNOW versions and consists of an LFSR part and an FSM part. The overall schematic is shown in Figure 1. The LFSR part is now a circular construction consisting of two shift registers, each feeding into the other. The FSM has three 128-bit registers and two instances of a single AES encryption round function.

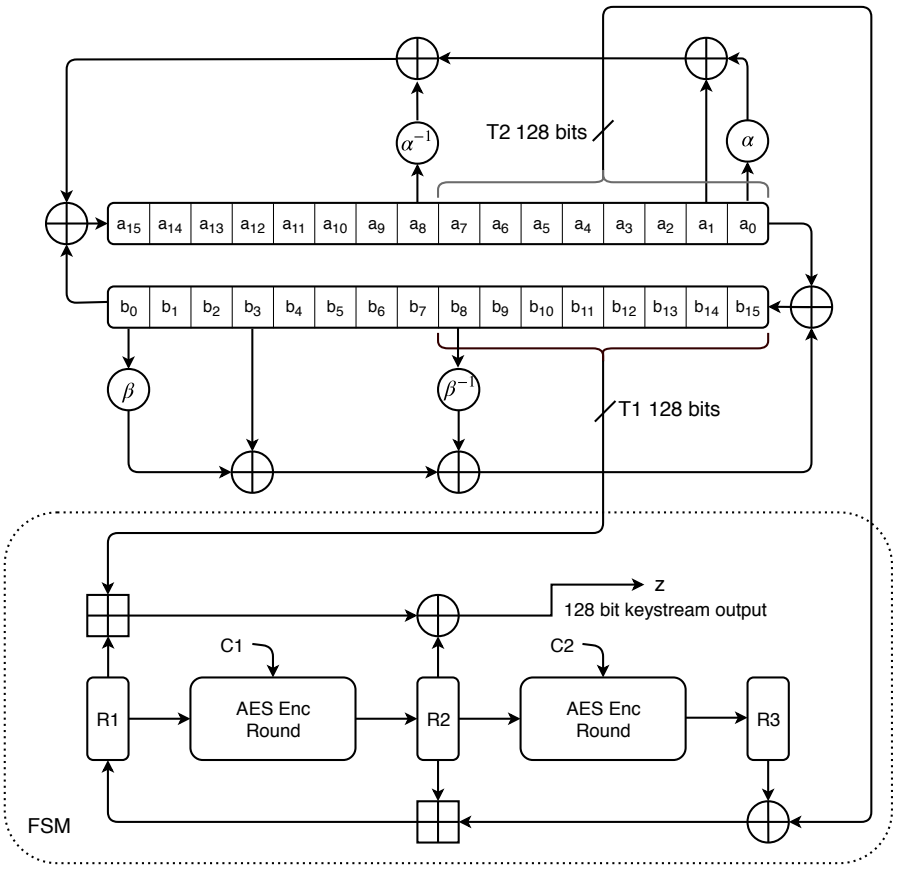

Fig. 1: Overall schematics of SNOW-V.

Starting with the LFSR part, we will now provide a detailed description of the design. The two LFSRs are named LFSR-A and LFSR-B, both of length 16 and with a cell size of 16 bits. The 32 cells are denoted $a_{15} \dots a_0$ and $b_{15} \dots b_0$ respectively.

Each cell represents an element in $\mathbb{F}_{2^{16}}$, but LFSR-A and LFSR-B have different generating polynomials. The elements of LFSR-A are generated by the polynomial

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x] \tag{1}$$

and the elements of LFSR-B are generated by

$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in \mathbb{F}_2[x]. \tag{2}$$

When we consider these elements of $\mathbb{F}_{2^{16}}$ as words, the x^0 position will be the least significant bit in the word. Let $\alpha \in \mathbb{F}^A_{2^{16}}$ be a root of $g^A(x)$ and $\beta \in \mathbb{F}^B_{2^{16}}$ be a root of $g^B(x)$. At time $t \geq 0$ we denote the states of the LFSRs as $(a^{(t)}_{15}, a^{(t)}_{14}, \ldots, a^{(t)}_1, a^{(t)}_0), a^{(t)}_i \in \mathbb{F}^A_{2^{16}}$ and $(b^{(t)}_{15}, b^{(t)}_{14}, \ldots, b^{(t)}_1, b^{(t)}_0), b^{(t)}_i \in \mathbb{F}^B_{2^{16}}$ respectively for LFSR-A and LFSR-B. Referring to Figure 1, the elements $a^{(t)}_0$ and $b^{(t)}_0$ are the elements to first exit the LFSRs. The LFSRs produce sequences $a^{(t)}$ and $b^{(t)}, t \geq 0$ which are given by the expressions

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \mod g^A(\alpha) \tag{3}$$

and

$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \mod g^B(\beta), \tag{4}$$

where the initial states of the LFSRs are given by $(a^{(15)}, a^{(14)}, \ldots, a^{(0)})$ and $(b^{(15)}, b^{(14)}, \ldots, b^{(0)})$. We would like to emphasize the notation here; $a^{(t)}$ means the symbol produced by the linear recursion in Equation 3 at time t, whereas $a^{(t)}_i, 0 \leq i \leq 15$ are the values of the cells in the LFSR-A at time t. In the case of α and β, the notation α^{-1} and β^{-1} are the inverses in the respective implemented fields.

As the reader might notice, we are a bit sloppy in Equation 3 and Equation 4 and apply the field addition operation between elements of different fields, but it should be interpreted as an implicit bit pattern preserving conversion between the fields.

Each time we update the LFSR part, we clock LFSR-A and LFSR-B 8 times, i.e., 256 bits of the total 512-bit state will be updated in a *single step*, and the two taps $T1$ and $T2$ will have fresh values. In Appendix A we give the proof that this circular construction gives the maximum cycle length of $2^{512} - 1$.

The tap $T1$ is formed by considering $(b_{15}, b_{14}, \ldots, b_8)$ as a 128-bit word where b_8 is the least significant part. Similarly, $T2$ is formed by considering $(a_7, a_6, \ldots, a_0)$ as a 128-bit word where a_0 is the least significant part. The mapping is pictured in Figure 2, and the expressions are given by

$$T1^{(t)} = (b^{(8t)}_{15}, b^{(8t)}_{14}, \ldots, b^{(8t)}_8), \tag{5}$$

$$T2^{(t)} = (a^{(8t)}_7, a^{(8t)}_6, \ldots, a^{(8t)}_0). \tag{6}$$

We will now turn to the FSM. The FSM takes the two blocks $T1$ and $T2$ from the LFSR part as inputs and produces a 128-bit keystream as output. $R1$, $R2$, and $R3$ are 128-bit registers, $\oplus$ denotes a bitwise XOR operation, and $\boxplus_{32}$ denotes an addition with carry, but split up into four 32-bit additions. So the four 32-bit parts of the 128-bit words are added with carry, but the carry does not propagate from a lower 32-bit word to the higher.

The output, $z^{(t)}$ at time $t \geq 0$, is given by the expression

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}. \tag{7}$$

Registers $R2$ and $R3$ are updated through a full AES encryption round function as shown in Figure 3, see [oST01] for details. Let us denote the AES encryption round function by $AES^R(IN, KEY)$. Then the update expressions for the registers are given by

$$R1^{(t+1)} = R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)}), \tag{8}$$

$$R2^{(t+1)} = AES^R(R1^{(t)}, C1), \tag{9}$$

$$R3^{(t+1)} = AES^R(R2^{(t)}, C2). \tag{10}$$

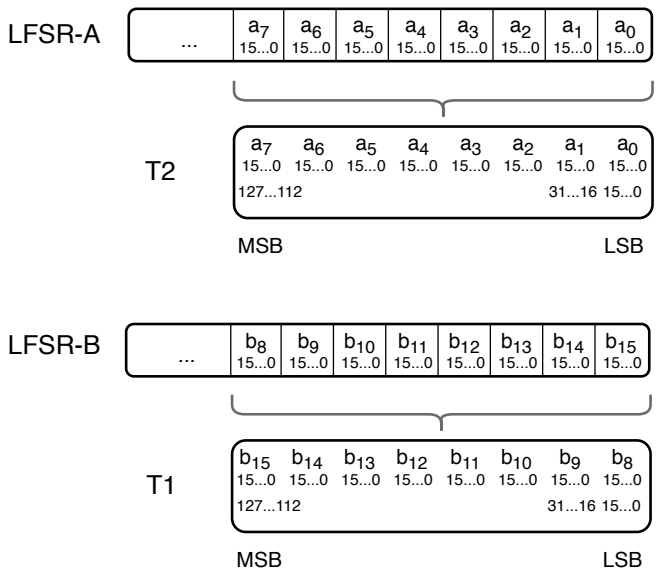

Fig. 2: Mapping the 16-bit words of the LFSRs into 128-bit words $T1$ and $T2$.

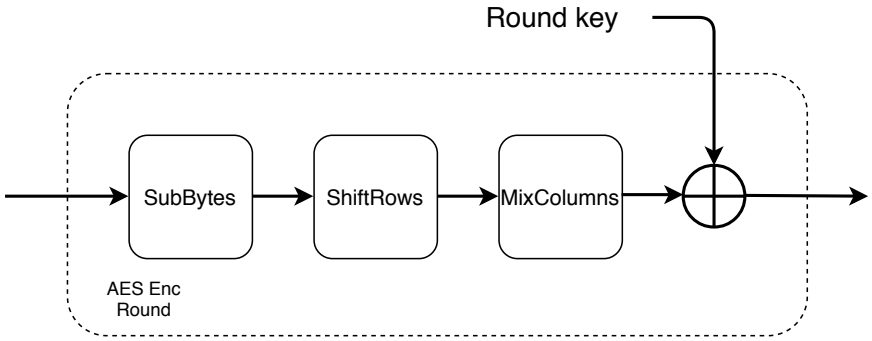

Fig. 3: Internal functions of the AES encryption round function.

The values of the two round key constants $C1$ and $C2$ are set to zero.

The mapping between the 128-bit registers and the state array of the AES round function follows the definition in [oST01], and is pictured in Figure 4.

2.1 Initialization

Initialization is done as described in this subsection. The algorithm has a 256-bit key K and a 128-bit IV vector as inputs. The key is denoted by

$$K = (k_{15}, k_{14}, \ldots, k_1, k_0),$$

where each k_i, $0 \leq i \leq 15$, is a 16-bit vector. The IV vector is denoted by

$$IV = (iv_7, iv_6, \ldots, iv_1, iv_0),$$

where again each iv_i, $0 \leq i \leq 7$, is a 16-bit vector.

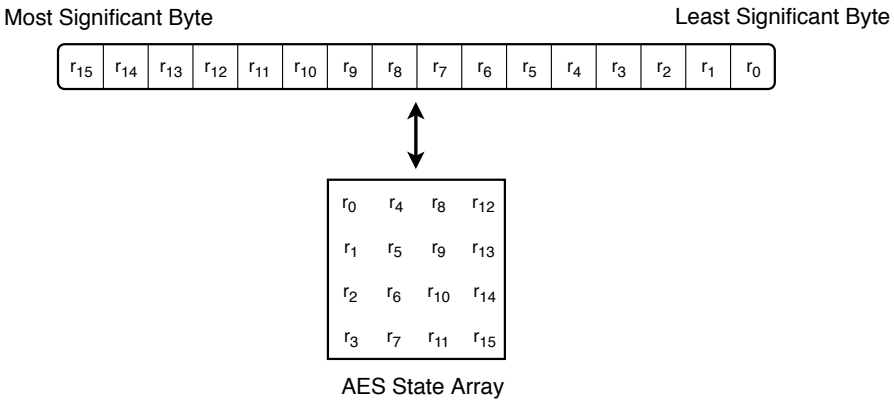

Fig. 4: Mapping between a 128-bit register value and the state array of the AES round function.

The first step of the initialization is to insert the key and IV into the LFSRs by assigning

$$(a_{15}, a_{14}, \ldots, a_0) = (k_7, k_6, \ldots, k_0, iv_7, iv_6, \ldots, iv_0)$$

and

$$(b_{15}, b_{14}, \ldots, b_0) = (k_{15}, k_{14}, \ldots, k_8, 0, 0, \ldots, 0).$$

Note that $(b_7, \ldots, b_0)$ will have a non-zero value when SNOW-V is used in AEAD-mode, see section 7.

Then the initialization consists of 16 steps where the cipher is updated in the same way as in the running-key mode, with the exception that the 128-bit output z is not an output but is xored into the LFSR structure to positions $(a_{15}, a_{14}, \ldots, a_8)$ in every step. Additionally, at the two last steps of the initialization phase, we xor the key into the $R1$ register, inspired by [HK18]. We also limit the keystream length to a maximum of 2^{64} for a single pair of key and IV vectors, and each key may be used with a maximum of 2^{64} different IV vectors.

The pseudocode in 1 clarifies the procedure.

Algorithm 1 SNOW-V initialization

procedure INITIALIZATION(K, IV)
  $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (k_7, k_6, \ldots, k_0)$
  $(a_7, a_6, \ldots, a_0) \leftarrow (iv_7, iv_6, \ldots, iv_0)$
  $(b_{15}, b_{14}, \ldots, b_8) \leftarrow (k_{15}, k_{14}, \ldots, k_8)$
  $(b_7, b_6, \ldots, b_0) \leftarrow (0, 0, \ldots, 0)$
  $R1, R2, R3 \leftarrow 0, 0, 0$
  for $t = 1...16$ **do**
    $z \leftarrow (R1 \boxplus_{32} T1) \oplus R2$
    $FSMupdate()$
    $LFSRupdate()$
    $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (a_{15}, a_{14}, \ldots, a_8) \oplus z$
    if $t = 15$ **then** $R1 \leftarrow R1 \oplus (k_7, k_6, \ldots, k_0)$
    if $t = 16$ **then** $R1 \leftarrow R1 \oplus (k_{15}, k_{14}, \ldots, k_8)$

This completes the description of SNOW-V, and the full algorithm can be summarized in the pseudocode as in 2, 3, and 4.

Algorithm 2 SNOW-V algorithm

procedure SNOW-V(K, IV)
  INITIALIZATION(K, IV)
  while more keystream blocks needed **do**
    $T1 \leftarrow (b_{15}, b_{14}, \ldots, b_8)$
    $z \leftarrow (R1 \boxplus_{32} T1) \oplus R2$
    $FSMupdate()$
    $LFSRupdate()$
    Output keystream symbol z

Algorithm 3 LFSR update algorithm

procedure $LFSRupdate()$
  for $i = 0...7$ **do**
    $tmp_a \leftarrow b_0 + \alpha a_0 + a_1 + \alpha^{-1} a_8 \bmod g^A(\alpha)$
    $tmp_b \leftarrow a_0 + \beta b_0 + b_3 + \beta^{-1} b_8 \bmod g^B(\beta)$
    $(a_{15}, a_{14}, \ldots, a_0) \leftarrow (tmp_a, a_{15}, \ldots, a_1)$
    $(b_{15}, b_{14}, \ldots, b_0) \leftarrow (tmp_b, b_{15}, \ldots, b_1)$

Algorithm 4 FSM update algorithm

procedure $FSMupdate()$
  $T2 \leftarrow (a_7, a_6, \ldots, a_0)$
  $tmp \leftarrow R2 \boxplus_{32} (R3 \oplus T2)$
  $R3 \leftarrow AES^R(R2)$ ▷ Note that the round keys for these AES
  $R2 \leftarrow AES^R(R1)$ ▷ encryption rounds are $C1 = C2 = 0$
  $R1 \leftarrow tmp$

3 Security analysis

The main and most important design criterion is the security of the design. This section contains a brief analysis for a number of possible standard attack approaches. Before going into the details of various attacks, we need to have a clear picture of the expected security. We have the target of providing 256-bit security in SNOW-V, by which we mean that we claim that the total cost of finding the secret key given some keystreams is not significantly smaller than 2^{256} simple operations.

The use of the algorithm is limited to keystreams of length at most 2^{64} and we also limit the number of different keystreams that are produced for a fixed key to be at most 2^{64}. There seem to be no use cases where it makes sense to violate this limitation. Although attacks beyond these

limits are certainly of academic interest, an attack claiming to break the cipher should meet this requirement.

We also frequently compare with AES-256 in the GCM mode. We note that exhaustive key search of AES-256 requires computational cost around 2^{256}. However, if used in the GCM mode, it actually takes complexity (and data) around 2^{64} to distinguish such keystreams from random. For SNOW-V, we claim that the security is never worse than the security of AES-256 in the GCM mode, for any kind of attack on the algorithmic level.

3.1 Initialization attacks through MDM/AIDA/cube attacks

Stream ciphers always have an initialization phase before producing keystream bits, during which the key and IV are loaded and a number of rounds (in the SNOW-V case, we use 16 rounds) are processed to fully mix the key and IV until the state becomes random-like. It should be difficult for the cryptanalyst to predict the generated keystream or to get some information about the initial key according to the output after initialization. Then it becomes vital to make sure that the key/IV loading has no fatal flaws and the initialization round is carefully designed in order not to result in a resource waste (too many rounds) or some weakness (too few rounds).

A chosen IV attack is one type of attacks targeting this problem [Mj06,EJT07], in which the adversary attempts to build a distinguishing attack to introduce randomness failures in the output by selecting and running through certain IV values. The rationale behind this idea is that: 1) the cipher can be regarded as a succession of "black box" Boolean functions f_i with the keystream as the output and key/IV as the input, and 2) any monomial coefficient in the algebraic normal form (ANF) representations of these Boolean functions should appear to be 1 (or 0) with probability 1/2 if f_i is drawn uniformly at random (see [Sta13] for more details). In this attack, the adversary fixes the key and a subset of IV bits and runs through all possible values of the non-fixed IV bits. The truth tables of the Boolean functions can be obtained after that, which are further used to compute the monomial coefficients in the ANF and compared with expected values. The best and most commonly used monomial is the maximum degree monomial (MDM) and the corresponding test is called MDM test. In [Sta10] one even allows setting arbitrary key values to build a non-randomness detector to further check whether the initialization is robust enough. It should be noted that the MDM test and AIDA (algebraic IV differential attack)/cube distinguishers [Vie07,DS09] are various forms of using higher order differentials [Lai94] on stream ciphers.

We employ the greedy MDM test algorithm in [Sta10] to test the SNOW-V initialization. We start with the worst 3-bit set under which the randomness result deviates the most from the expected value and gradually increase to a 24-bit set. Every time when we add one more bit from the remaining bits, we select the bit leading to the worst randomness result. Continuing such steps until we get a 24-bit set (sets with more bits can be tested on more powerful computers). Figure 5 shows the maximum number of initialization rounds failing the MDM test under different bit set sizes. The results for 1, 2 and 3-bit sets are based on the exhaustive search, while for the sets with other sizes, the results are based on a greedy search from the initial worst 3-bit set. It can be seen that roughly the first 7 rounds out of 16 fail the MDM test. One can also note that the number of rounds that the MDM test can detect grows very slowly with the size of the set of key/IV bits that are exhausted. In an attack, one could consider sets of sizes up to 64 bits. This indicates that the 16 initialization rounds in SNOW-V should be enough for the cipher and that the output of the cipher has become random-like after the initialization. It also indicates that significantly reducing the number of rounds might be dangerous.

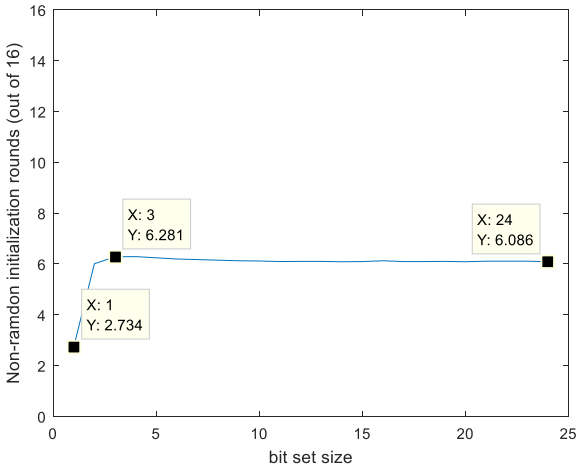

Fig. 5: The maximum number of initialization rounds failing the MDM test under different bit set sizes.

3.2 Other initialization attacks

Another attack possibility is to launch a differential attack, either in the IV bits only, or in combination with key bits. The latter would then lead to a related-key attack. Since the initialization contains 16 rounds, each including two applications of the AES encryption round function, the differential would have to go through a lot of highly nonlinear operations, which makes this approach less successful.

Finally, a further option is the slide attacks [BW99]. Such sliding properties have been considered on previous versions in the SNOW family [KY11]. The idea is to have the same initial state for two different key/IV pairs in different time instances. Then they will produce the same keystream with the difference of a shift in time. Since the required IV values vary with the choice of key bits, it is questionable whether such an approach is useful at all in cryptanalysis, but at least it indicates that the cipher is not to be considered as a random function of both the key and IV. For SNOW-V such properties would still be much more difficult to find, due to the update of 128-bit blocks in each time instance and the use of the $FP(1)$-mode [HK18] in the initialization.

3.3 Time/Memory/Data tradeoff attacks

A Time/Memory/Data tradeoff (TMD-TO) attack is a generic method of inverting ciphers by balancing between spent time, required memory and obtained data, which can be much more efficient and applicable than an exhaustive key search attack. Some stream ciphers are vulnerable to TMD-TO attacks, and their effective key lengths (e.g., n-bit) could then be reduced towards the birthday bound (i.e., $n/2$), typically happening if the state size is small. A well known such attack on A5/1 was given in [BSW01].

The TMD-TO attacks have two phases: a preprocessing phase, during which the mapping table from different secret keys or internal states to keystreams is computed and stored with time complexity P and memory M; and a real-time phase, when attackers have intercepted D keystreams and search them in the table with time complexity T, expecting to get some matches and further recover the corresponding input. By balancing between parameters P, D, M, and

T under some tradeoff curves, attackers can launch attacks according to their available time, memory and data resources. The most popular tradeoffs are Babbage-Golic (BG) [Bab95,Gol97] and Biryukov-Shamir (BS) [BS00] tradeoff with curves $TM = N,\ P = M$ with $T \leq D$; and $MT^2D^2 = N^2$, $P = N/D$ with $T \geq D^2$, where N is the input space, respectively. Attackers can try to reconstruct the internal state at a specific time or recover the secret key.

The rationale behind the TMD-TO attacks that try to reconstruct the internal state is that in many stream ciphers, the internal state update process is invertible, which means that if an attacker manages to reconstruct an internal state at any specific time, it can not only obtain subsequent new keystreams by running the cipher forwards, but also recover previous states iteratively and further get the underlying secret key by running backwards. But for the SNOW-V case, attackers have no obvious ways to reconstruct the internal state, since SNOW-V has a large internal state with 894 bits (2×256-bit LFSRs + 3×128-bit registers), which is 3.5 times the secret key length. The best attack complexity achieved is under BG tradeoff with point $T = M = D = N^{1/2} = 2^{447}$, which is still much worse than the exhaustive key search attack. Actually, SNOW-V satisfies the rule derived from TMD-TO attacks in [Gol97] and widely applied in the design of new ciphers, that the size of the internal state should be at least twice the size of the secret key to get the expected security level.

Moreover, in SNOW-V, attackers would get even less although they reconstructed an internal state. While computing subsequent new keystreams corresponding to that specific IV is still possible, they can not trivially recover the secret key or keystreams under other IV values. This is due to the key masking to the register R1 at the last two rounds of initialization, which represents a form of an instantiation of the $FP(1)$-mode introduced in [HK18].

Attackers can also try to recover the secret key directly. To do so, some mappings from different key/IV pairs to generated keystream segments are firstly pre-computed and stored [HS05,DK08]. If attackers get some keystream data under different secret keys corresponding to these IV values, they can search them in the table to expect a collision and further recover some of the secret keys directly. The tradeoff curves are still the same in that to recover the internal states except N is now changed to be the size of the set of all possible (K, IV) pairs. In the SNOW-V case, the sizes of key and IV spaces are 2^{256} and 2^{128}, respectively. The typical points for BG and BS attacks are $T = D = M = 2^{192}$ and $T = 2^{256}$, $D = M = 2^{128}$, which are both unrealistic to achieve in practice. Someone would question that the efficient size of the key in the first tradeoff is reduced from 256 to 192 bits, but actually, no ciphers including AES-256 can be immune to this as long as their IV sizes are smaller than the key sizes. In any case, the corresponding multikey attacks on AES-256 are not more costly.

3.4 Linear distinguishing attacks and correlation attacks

Traditionally, the main threat against stream ciphers has been various types of linear attacks, either in the form of distinguishing attacks on the keystreams, or state recovery attacks through correlation attacks. The basic foundations of correlation attacks can be found in papers like [CJS01,CJM02] and an overview of distinguishing attacks is to be found in [HJB09].

The basic technique for these types of attacks is to use linear approximations of the nonlinear operations used in the cipher and then derive a linear relationship between output values from different time instances. Such a relationship will then hold only as a very rough approximation, which in turn can be thought of as a linear function of some given output bits being considered as a sample drawn from a nonuniform distribution. This approach may give a distinguishing property for the keystream. If the relationship also involves state bits, the same arguments may give samples that are highly noisy observations of state bits, which in turn may be linear

combinations of the original initial state. This may give a way to recover the state and that is the foundation of a correlation attack.

For SNOW 2.0, several distinguishing attacks and correlation attacks have been proposed [NW06,ZXM15]. The basic idea has been to approximate the FSM part through linear masking and then to cancel out the contributions of the registers by combining expressions for several keystream words. We should note that this kind of attacks tend to require an extremely large length of the keystream. Also, no significant attack of this type on SNOW 3G has been published. We now consider a similar approach for making some basic arguments on SNOW-V.

Since we always set $C1 = C2 = 0$ we can simplify the notation of the output function and the update:

$$\begin{aligned} z^{(t)} &= (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}, \\ R1^{(t+1)} &= R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)}), \\ R2^{(t+1)} &= AES^R(R1^{(t)}), \\ R3^{(t+1)} &= AES^R(R2^{(t)}). \end{aligned}$$

A linear approximation of the FSM would then try to cancel out the contribution from the registers, leaving keystream symbols and the LFSR contribution. Assume that value of the registers at some time t is $(\hat{R1}, \hat{R2}, \hat{R3})$. Then we have

$$\begin{aligned} z^{(t)} &= (\hat{R1} \boxplus_{32} T1^{(t)}) \oplus \hat{R2}, \\ R1^{(t+1)} &= \hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)}), \\ R2^{(t+1)} &= AES^R(\hat{R1}), \\ R3^{(t+1)} &= AES^R(\hat{R2}). \end{aligned}$$

For time $t + 1$,

$$\begin{aligned} z^{(t+1)} &= ((\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})) \boxplus_{32} T1^{(t+1)}) \oplus AES^R(\hat{R1}), \\ R1^{(t+2)} &= AES^R(\hat{R1}) \boxplus_{32} (AES^R(\hat{R2}) \oplus T2^{(t+1)}), \\ R2^{(t+2)} &= AES^R(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})), \\ R3^{(t+2)} &= AES^R(AES^R(\hat{R1})) \end{aligned}$$

and the next keystream block is

$$z^{(t+2)} = (AES^R(\hat{R1}) \boxplus_{32} (AES^R(\hat{R2}) \oplus T2^{(t+1)})) \boxplus_{32} T1^{(t+2)}) \oplus AES^R(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})).$$

Let us now consider $\boxplus_{32}$ being approximated by $\oplus$ and the $AES^R(X)$ operation approximated as $X \cdot M$ for some 128×128 binary matrix M. Then we could express the keystream blocks as

$$\begin{aligned} z^{(t)} &= R1^{(t)} \oplus T1^{(t)} \oplus R2^{(t)} \oplus N_1, \\ z^{(t+1)} &= \hat{R2} \oplus \hat{R3} \oplus T2^{(t)} \oplus N_2 \oplus T1^{(t+1)} \oplus N_3 \oplus \hat{R1} \cdot M \oplus N_1', \\ z^{(t+2)} &= \hat{R1} \cdot M \oplus N_1' \oplus \hat{R2} \cdot M \oplus N_2' \oplus T2^{(t+1)} \oplus \\ &\quad T1^{(t+2)} \oplus N_4 \oplus N_5 \oplus (\hat{R2} \cdot M \oplus \hat{R3} \oplus T2^{(t)} \oplus N_5) \oplus N_3'. \end{aligned}$$

Here each random variable N_i represents the noise introduced by approximating the ith $\boxplus_{32}$ by writing $\boxplus_{32} = \oplus + N_i$. Similarly, N_i' represents the noise introduced by writing $AES^R(X) =$

$X \cdot M + N'_i$ for the ith approximated AES round function. By rewriting as

$$(z^{(t)}, z^{(t+1)}, z^{(t+2)}) = \left(\hat{R1}\ \hat{R2}\ \hat{R3}\right) \begin{pmatrix} I & M & M \\ I & I & 0 \\ 0 & I & M \end{pmatrix} + T + N,$$

where T is the contribution from $T1$, $T2$ values and N is the sum of all noise values N_i, N'_j. Examining the matrix, one sees that it is not possible to have reduced rank for any meaningful approximation matrix M. So we conclude that based on the direct approach of approximation as above, it would require 4 consecutive keystream blocks in order to cancel the contribution from the registers in the FSM. Such an approach would then involve even more noise variables and one will have the form $N_j \cdot M^2$, where N_j is the noise from approximating one $\boxplus_{32}$ with $\oplus$. Such a linear approximation of the FSM could then be used in a correlation attack. However, since such an attack would need to use a combination of several linear approximations from different time instances that would add the corresponding noise, it does not seem to be a fruitful way of attacking the cipher as the noise will be very strong. If one would devise a distinguishing attack, one would instead have to cancel the contribution from the LFSR part, which again will give a noise very close to the uniform distribution. We do not see a path to identify a strongly biased approximation in this way.

3.5 Algebraic attacks

In an algebraic attack the attacker derives a number of nonlinear equations in either unknown key bits or unknown state bits and solves the system of equations. In general, the problem of solving a system of nonlinear equations is not known to be solvable in polynomial time (even for quadratic equations), but some special cases might be solved efficiently [CKPS00].

For SNOW 2.0 there was a very interesting algebraic attack on a simplified version, given in [BG05]. However, due to the use of three FSM registers instead of two, applying such an approach on SNOW-V does not give such a nice quadratic system as in [BG05].

So for a general algebraic attack, we should either target the key or the state. For the latter, one would need to use equations from 7 keystream blocks to be able to solve for the $7 * 128$ bit internal state. That would involve nonlinearity from 11 AES encryption round functions and 13 $\boxplus_{32}$ operations. Instead, targeting the key bits would require stepping through the equations of the 16 initialization rounds together with the equations of two keystream blocks. Both these approaches are giving systems of nonlinear equations that appear to be much more difficult to solve than corresponding equations for AES-256. This is due to the use of the $\boxplus_{32}$ operation.

3.6 Guess-and-determine attacks

In a guess-and-determine attack one guesses part of the state and from the keystream equations, and determines the value of other parts of the state. The goal is to guess as few bits as possible and determine as many as possible through keystream equations. For the case of SNOW-V, the equation $z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}$ involves three unknown values, each of size 128 bits. In order to determine some state bits, one then has to guess two of them, i.e. guessing 256 bits. Then looking at the equation for $z^{(t+1)}$, it would require the guess of one more 128 bit value. This indicates that a guess-and-determine attack would not be successful.

3.7 Other attacks

We have not made any specific design choices to explicitly support implementations that should protect against side-channel attacks and fault attacks. So such attacks, if relevant for an application, have to be considered when the algorithm is implemented. In particular, information leakage from the CPU in a software implementation must be carefully considered.

4 Hardware implementation aspects

When designing new algorithms targeting existing systems, reusability of hardware components is important to reduce area and cost of the ASICs. Many systems dealing with network communication security implement some form of AES acceleration, either in a specialized ASIC or as specialized CPU instructions. SNOW-V leverages this co-existence by using two full AES encryption rounds as the main nonlinear element. A hardware implementation of SNOW-V can utilize either one or two external AES cores, if present, or implement its own AES encryption rounds in a stand-alone design for maximum speed. Although a 128-bit implementation is straight-forward from the algorithm description, it has some drawbacks when we only have one single external AES core available, as is the case in many constraint implementations. In this section we will consider how to implement SNOW-V using a single AES core with a 64-bit hardware architecture. We will refer to the 64-bit and 128-bit hardware implementations as the 64-SNOW-V and 128-SNOW-V respectively.

4.1 SNOW-V 64-bit Hardware Architecture

In this section we propose a 64-bit hardware architecture where SNOW-V requires a *single* AES encryption core (external or built-in), and each clocking of 64-SNOW-V produces 64 bits of the keystream.

Cons: an additional 64-bit delay register D is needed; the logic needs additional 5 64-bit multiplexers; two clocks to produce 128 bits of keystream that actually halves the speed.

Pros: a single AES encryption core is needed; produces 64 bits of keystream at *each clock*; all basic operations in both FSM and LFSR, such as XOR and ADD, are now halved in size.

In order to utilize a single AES core the FSM update function should be split into two steps. The *main critical path* is the AES EncRound, which means that while splitting FSM into two stages we should avoid any extra logic on the input and output signals of the AES core. Thus, input to and output from the AES core must be registers.

Let us split all 128-bit registers and all 128-bit signals of the FSM block, say X, into two 64-bit halves as X_a (low) and X_b (high). We also assume that the tap values $T1$ and $T2$ from the LFSRs also arrive in 64-bit chunks, such that every even clock FSM gets $T1_a$ and $T2_a$, and every odd clock $T1_b$ and $T2_b$.

In Figure 6 we propose a possible way to split the FSM such that it contains the two circuits for even and odd steps, 0 and 1 resp. (excluding the gates needed for initialization). One can notice that after these two steps the content of the registers $R1, R2, R3$ become updated to new 128-bit values $'R1,' R2,' R3$, and ready to process the next 128 bits of data with the same two steps. The above two circuits are then combined into a single circuit using multiplexers.

In Figure 7 the complete hardware architecture for 64-bit SNOW-V is presented. There are 6 64-bit multiplexers in total, and we denote the control signal to them by $M_1..M_6$, respectively. There are also 5 64-bit AND gates, the purpose of which is to either bypass the signal or block it. Those AND blocks are controlled by four signals G_A, G_Z, G_K, G_F, the latter controls 2x64

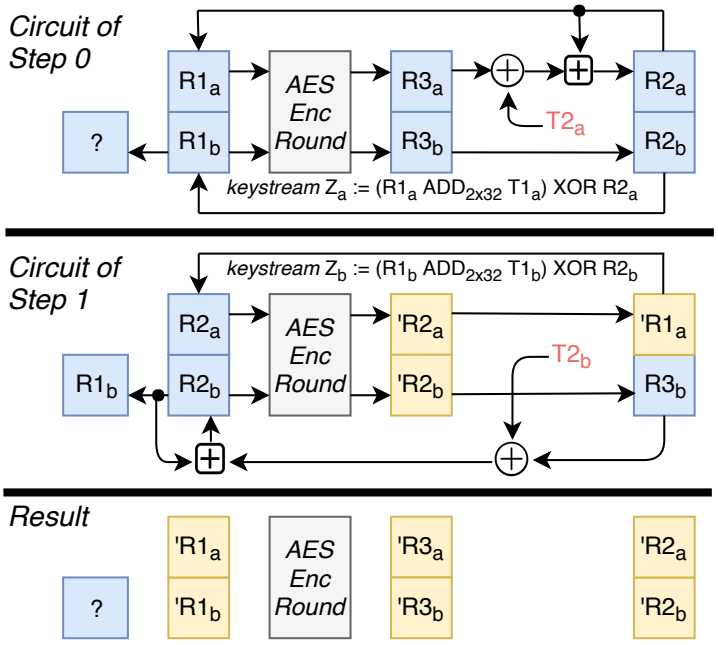

Fig. 6: Splitting of FSM into two steps in order to utilize only one AES core.

AND-blocks. The Control Unit in Figure 7 generates the control signals for the multiplexers and AND gates depending on the state of SNOW-V.

Critical path. Our primary assumption is that the AES encryption round would be the *main critical path (MCP)*. However, one can easily determine that the *secondary critical path (SCP)* would be the sequence MUX-ADD-XOR-AND-XOR over 2x32-bit integers, denoted by red wires in Figure 7. Thus, when selecting 32-bit adders one should make sure that they are fast enough so that the *MCP* is sustained.

The algorithm has 3 stages:

Stage 1 – Loading. The design is constructed such a way that the registers do not need to have any `RESET` signal. Instead, all registers will be sequentially loaded with the key and IV, and the remaining registers will be zeroized, during this stage.

The stage begins with a strobe signal on `LOAD`, and the first 64-bit chunk of data is expected on the `IN_DATA` bus. In total, the stage expects to receive 8 64-bit words each clock in the following order: $\{iv_0, iv_1, k_0, k_1, 0, 0, k_2, k_3\}$.

In this stage, the control unit should block AND gates $G_Z = G_A = 0$, and set $M_6 = 1$, in order to concatenate LFSRs A and B into a single large LFSR while shifting in the initialization data. In order to zeroize FSM registers, the control unit should block $G_F = 0$ and also enforce the multiplexer inputs $M_4 = 1, M_5 = 0$. G_K is set to 0.

After the 8 clocks where the key and IV are loaded, we proceed to stage 2.

Stage 2 – Initialization. In this stage, the FSM works in the same way as when it produces keystream output symbols, i.e. the multiplexer control signals switches according to even/odd clock cycle as explained previously. The LFSRs are connected together by setting $G_Z = G_A = 1$ and switching $M_6 = 0$ to disable any external input.

Note that we placed the AND gating *after* the registers $R3_a, R3_b$, so that we do not add extra depth to the critical path of AES core, hence these registers will not be zeroized. To overcome

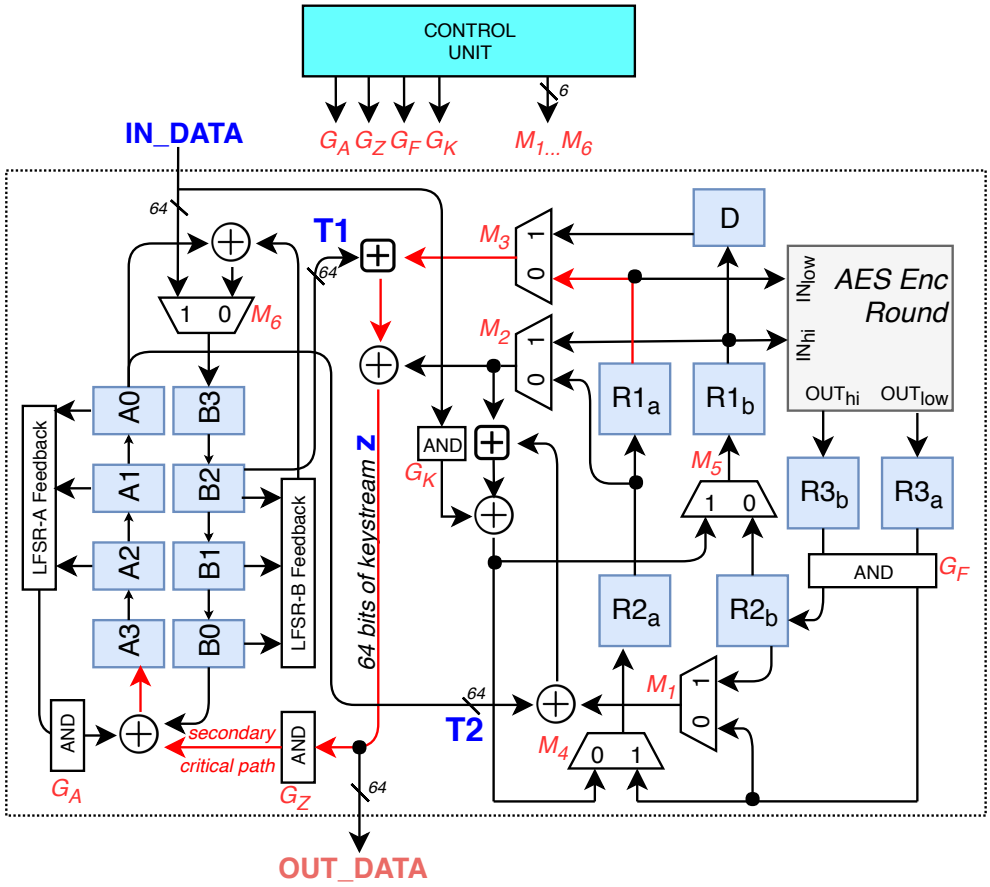

Fig. 7: Hardware architecture of 64-bit SNOW-V with a single AES core.

this problem the control unit generates $G_F = 0$ in the first clock of this stage, and then sets $G_F = 1$ until the end of stage 2. We keep $G_K = 0$ for the first 28 clocks. In the remaining 4 clocks we need to XOR the key K to $R1$ according to the initialization procedure. So we enable $G_K = 1$ and expect to receive $\{k_0, k_1, k_2, k_3\}$ consecutively from the input bus `IN_DATA`. After this, the circuit is ready to produce keystream words.

Stage 3 – Keystream generation. Both LFSR and FSM operate normally. The control unit in this stage detaches the Z signal from being feeded into LFSR-A by setting $G_Z = 0$. The input bus is also detached by setting $G_K = M_6 = 0$.

4.2 Theoretical Analysis of 64/128-bit SNOW-V in Hardware

The area will be estimated in terms of **gate equivalence (GE)**, where 1GE = size of a NAND gate. The speed will be estimated in terms of Gigabits per second (Gbps), based on known speed results of AES circuits. We will use GE values given in [Sam00] for 1-speed technology elements.

For comparison with AES, we will use one of the more recent results from [UMHA16] where an area-speed optimized AES-128 (10 rounds) on NanGate 15nm technology runs with the speed 71.19 Gbps and has the area **17232 GE**. This means that having the same design, AES-256 (14 rounds) would run with the speed of **50.85 Gbps**.

Our basic assumption is that the AES core is the critical path of the SNOW-V circuit. Thus, if SNOW-V would utilize a single AES core as above, the speed of 64-SNOW-V could be as high as **356 Gbps**. The speed of 128-SNOW-V with two AES cores is therefore as high as **712 Gbps**. What remains is to calculate the hardware cost of SNOW-V, excluding the external AES core, but including the cost of integration into that external AES core. We will also exclude the control unit, as this can be implemented with a very few gates and latches and every implementation will have slightly different needs of control and ready signaling.

State Registers. For 64-SNOW-V, there are 512 registers for the LFSR and 6x64+64 registers for the FSM. Since our 64-bit implementation does not require complex latches (e.g., no `RESET`), we can use the simplest D-latch with Q-output only from [Sam00] [FD1Q]. The total cost is $960 * 4.33 =$ **4157 GE**.

For 128-SNOW-V we also need 512 registers for the LFSRs without reset, and 3x128 registers with `RESET` [FD2Q], thus resulting in 512*4.33+(3*128)*5.67 = **4394 GE**.

For **arithmetical 32-bit adders** we suggest to take, for example, a Han-Carlson 32-bit adder, as it has a low area overhead (15%-25% larger than Ripple-Carry adders) and a very small delay $O(log(n))$ – which is important in order to keep the critical path upper bounded by the AES round function. We can estimate these components as 4x(30FADD3 + 2HADD2)+20%= $4(30 * 6.33 + 2 * 3.67) * 1.20 =$ **947 GE** for 64-SNOW-V and **1894 GE** for 128-SNOW-V.

The remaining part of the **FSM update logic** therefore contains 3x64AND2 + 5x64MUX2 + 3x64XOR2 $= 3 * 64 * 1.33 + 5 * 64 * 2.33 + 3 * 64 * 2.33 =$ **1448 GE** for 64-SNOW-V and (128AND2+3x128XOR2)=**1065 GE** for 128-SNOW-V.

LFSR Update logic involves two circuits for the feedback functions. 16-bit field multiplications by $\alpha, \alpha^{-1}, \beta, \beta^{-1}$ can be done with 8 XORs in each case, since the Hamming weight of both $g^A(\alpha)$ and $g^B(\beta)$ is 8.

However, let us have a closer look on how each bit of, e.g. a_{16} is calculated. Each bit $a_{16}[i], 14 \geq i \geq 1$ is unconditionally depending on four bits, namely

$$a_{16}[i] : a_0[i-1] + a_1[i] + a_8[i+1] + b_0[i] \tag{11}$$

The end bits are easy to work out too. Some of the bits of a_{16} are also depending on $a_0[15]$ and $a_8[0]$, due to the multiplication with α and α^{-1}. Table 2 gives a full overview of the dependencies for both a_{16} and b_{16}.

i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Depending on
$a_{16}[i]$	✓			✓	✓			✓					✓	✓	✓	✓	$a_0[15]$
	✓	✓			✓	✓			✓					✓	✓	✓	$a_8[0]$
$b_{16}[i]$	✓	✓			✓			✓		✓	✓				✓	✓	$b_0[15]$
	✓	✓	✓			✓			✓		✓	✓				✓	$b_8[0]$

Table 2: Bit dependencies due to multiplications for a_{16} and b_{16}.

This means that in order to compute $a_{16}[i]$, we have to XOR 4, 5, or 6 different input bits. For example, in the table above we see that the $a_{16}[13]$ is only dependent on the basic input bits in Equation 11, and the XOR gate needs 4 inputs:

$$a_{16}[13] = a_0[12] + a_1[13] + a_8[14] + b_0[13].$$

On the other hand, $a_{16}[11]$ needs to XOR 6 inputs:

$$a_{16}[11] = a_0[10] + a_1[11] + a_8[12] + b_0[11] + a_0[15] + a_8[0].$$

since the multiplication with α and α^{-1} will both influence that bit.

Following the hardware architecture of 64-SNOW-V given in Figure 7 we have to split the calculation of the feedback function LFSR-A due to the control AND-gateway. Also, the circuit should compute 4 16-bit updates in parallel. Summarizing, we get (a) LFSR-A feedback function, *excluding* input from b_0: 4x(5XOR3 + 6XOR4 + 5XOR5) $\approx 4 * (5 * 4.00 + 6 * 6.00 + 5 * 8.00)$ = **384 GE** ; (b) LFSR-B feedback function, *including* input from a_0: 4x(4XOR4 + 8XOR5 + 4XOR6) $\approx 4 * (4 * 6.00 + 8 * 8.00 + 4 * 10.00)$ = **512 GE**; (c) the remaining part of LFSR block: 2x64AND2 + 64XOR3 + 64MUX2 = $64 * (2 * 1.33 + 4.00 + 2.33)$ = **575 GE**. For 128-SNOW-V we simply double the above numbers.

Integration into an external AES Engine requires input multiplexers for 128 bits of the plaintext and 128 bits for the round key. However, the AES round keys $C1$ and $C2$ are zeroes so that we can use 128AND gates, instead. In total we get 128MUX2 + 128AND2 = $128 * (2.33 + 1.33)$ = **468 GE** for 64-SNOW-V. 128-SNOW-V requires two such integration circuits.

In case we decide to implement SNOW-V with its own internal AES EncRound, the hardware cost could be as small as 16 AES SBoxes, plus some logic for MixColumn. Also note that in this case the critical path decreases since we only need the forward SBox and thus any outer multiplexing logic for a combined forward and inverse SBox can be removed. This could lead to a potential speed up for 128-SNOW-V.

The part MixColumn of AES encryption round, applied to the AES state $\{r_{i,j}\}$ for $0 \leq i, j \leq 3$, is the following matrix multiplication.

$$\begin{bmatrix} r'_{0,j} \\ r'_{1,j} \\ r'_{2,j} \\ r'_{3,j} \end{bmatrix} = \begin{bmatrix} 2\ 3\ 1\ 1 \\ 1\ 2\ 3\ 1 \\ 1\ 1\ 2\ 3 \\ 3\ 1\ 1\ 2 \end{bmatrix} \cdot \begin{bmatrix} r_{0,j} \\ r_{1,j} \\ r_{2,j} \\ r_{3,j} \end{bmatrix}, 0 \leq j \leq 3.$$

That can be computed in a depth 2 circuit, for each $0 \leq j \leq 3$, as $t_0 = r_0 + r_1$, $t_1 = r_1 + r_2$, $t_2 = r_2 + r_3$, $t_3 = r_3 + r_0$, and then $r'_0 = 2t_0 + t_2 + r_1$, $r'_1 = 2t_1 + t_3 + r_2$, $r'_2 = 2t_2 + t_0 + r_3$, $r'_3 = 2t_3 + t_1 + r_0$, where multiplication $2t_i$ is the multiplication by x in the Rijndael field and can be implemented with 3XOR2. The cost of MixColumn is therefore 4x4x(8XOR2 + 8XOR3 + 3XOR2) = **922 GE**.

The cost of a single forward SBox is around 220 GE (see, e.g., [RMTA18]). Thus, for a **single internal AES EncRound** the total cost is $16 * 220 + 922$ = **4442 GE**.

Summarizing the above we can derive the comparison given in Table 3 .

Hardware design	AES-256 from [UMHA16]	64-SNOW-V 1xAES ext. core	64-SNOW-V 1xAES int. round	128-SNOW-V 2xAES ext. cores	128-SNOW-V 2xAES int. rounds
Area	17232 GE	8491 GE	12465 GE	11231 GE	19179 GE
Speed	50.85 Gbps	358 Gbps	358+ Gbps	712 Gbps	712+ Gbps

Table 3: Theoretical comparison of four SNOW-V versions vs AES-256 in hardware.

5 Software implementation aspects

One important change in future telecom networks is the virtualization of the network functions. This puts new requirements on the crypto algorithms used to protect the traffic in that it needs to execute fast in a pure software implementation on modern CPUs. According to [ITU17], the minimum requirements related to 5G radio interface are 10 Gbps uplink and 20 Gbps downlink, at peak data rates. Classical encryption algorithms cannot reach these high speeds in pure software without any hardware support.

Nowadays, most of CPU vendors provide large registers and vectorized SIMD instructions, such as AVX2 set of instructions (intrinsics) that can execute over registers of up to 256 bits. Typical instructions include such functions as XOR, AND, nADD32, etc., applied to long registers, where, depending on the instruction, a single register can be represented as a vector of 8/16/32/64-bit values.

AES is one of the most widely used crypto algorithms and it has received special support by CPU vendors in the form of SIMD instructions (AES-NI for Intel) that makes it possible to execute AES quite fast even on user-grade laptops. Crypto ciphers SNOW 3G and ZUC, standardized in 4G, and other ciphers (to our knowledge), cannot reach the speed even close to AES when AES-NI is used.

SNOW-V is designed to perform very fast in software, with the aim to utilize *currently available* SIMD instructions. However, even without AES-NI, SNOW-V can be implemented quite efficiently with 16 64-bit registers. Our take-away is that if a given platform supports AES-NI then other SIMD instructions are also likely supported. If AES-NI is not available then AES-256 will be much slower than SNOW-V, and actually, slower than SNOW 3G as well. This section is written with Intel intrinsics notation, but similar implementations can likely be made on other CPUs, e.g. AMD and ARM. A comprehensive guide on Intel's intrinsics can be found in [Int18].

The FSM part of SNOW-V is quite straightforward to implement using 128-bit registers `__m128i` and AES-NI intrinsic function `_mm_aesenc_si128()`. For 4 parallel arithmetic additions one can use `_mm_add_epi32()`[3].

The key to an efficient implementation of the LFSRs is choosing the right data structures. We propose to store the content of the two LFSRs in two 256-bit registers `__m256i hi, lo`, such that:

$$\texttt{lo[127..0 bits]} = \{a_7, \ldots, a_0\} \qquad \texttt{hi[127..0 bits]} = \{a_{15}, \ldots, a_8\}$$
$$\texttt{lo[255..128 bits]} = \{b_7, \ldots, b_0\} \qquad \texttt{hi[255..128 bits]} = \{b_{15}, \ldots, b_8\}$$

To perform a single LFSR update (8 steps), we only need to calculate new values for one register, `hi_new=update(lo, hi)` while the other register update is a copy `lo_new=hi`.

Let `gA=0x990f` represents the generating polynomial $g^A(\alpha)$ of the field $\mathbb{F}_{2^{16}}^A$, without the term α^{16}. Then, multiplication of x by α in $\mathbb{F}_{2^{16}}^A$ can be done as follows: we first shift `x<<1`, then, based on the 15th bit of the original `x`, we XOR the result with `gA`. This may be done with only 4 instructions, using 16-bit values

```
mul_alpha(uint16 x, uint16 gA) := (x<<1) xor ( ((signed int16)x >> 15) and gA)
```

Note that the condition wether to `xor` with `gA` or not is implemented with the help of the 16-bit `mask = (signed int16)x >> 15`, where the mask is created by the *arithmetical* shift of

[3] This intrinsic is intended for addition of signed integers but because most CPUs use two's complement representation for negative numbers, it will produce the correct results also for the unsigned addition needed in SNOW-V.

the *signed* `x` to the right by 15 positions. The arithmetical shift to the right results in propagation of the sign (15th) bit, thus forming the mask either `0xffff` in case the bit 15 was 1, or `0x0000`, otherwise.

The above trick can be applied to the combined 256-bit vector `lo` $= (b_7, \ldots, b_0, a_7, \ldots, a_0)$ to multiply the first half with α from the first base field $\mathbb{F}_{2^{16}}^A$ and the high part with β from the second base field $\mathbb{F}_{2^{16}}^B$, simultaneously. Here we need to use `_mm256_srai_epi16()` that performs *arithmetical* shift to the right of 16 16-bit *signed* integers represented in the combined 256-bit register `lo`. Obviously, the `and` operand should be done with the constant where the low 8 x 16-bit values are `gA=0x990f` and the second half contains `gB=0xc963`.

A similar idea is applied for multiplication of `hi` by α^{-1} and β^{-1}. In our reference implementation we found the way with only 4 instructions with the help of a non-trivial intrinsic `_mm256_sign_epi16()` – however, if that intrinsic is not available then there is an alternative solution with 5 instructions.

The results of the above two steps should be XORed together with the values at tap offsets 1 and 3 for LFSRs A and B, respectively. The latter part is just byte shuffling that can be done with `_mm256_blend_epi32()` and `_mm256_alignr_epi8()`, three instructions in total.

6 Performance results

6.1 In Software

The natural algorithm to compare with is AES-256, implemented with AES-NI intrinsics. We have done a number of performance tests of SNOW-V and AES-256 (CBC) on a user-grade laptop with i7-8650U CPU @1.90GHz with Turbo Boost up to @4.20GHz, testing each algorithm on a single thread and with different sizes of the input plaintext. Before each encryption process, we perform a key/IV setup procedure for both SNOW-V and AES-256. The results are presented in Table 4.

Algorithm	Size of plaintext (bytes)				
	$2^{32}+$	2048	256	64	16
AES-256	9.17 Gbps	8.48 Gbps	7.98 Gbps	6.75 Gbps	2.62 Gbps
SNOW-V	61.18 Gbps	56.55 Gbps	27.55 Gbps	10.46 Gbps	3.04 Gbps

Table 4: Performance comparison of SNOW-V and AES-256 both with AVX2

6.2 Use case scenarios

For a **large plaintext** SNOW-V outperforms AES-256 by around 6 times, even with an AES-NI implementation of AES-256. Some block cipher modes (e.g. CTR) can be parallelized and in order to reach a similar speed as SNOW-V running on 1 CPU, AES requires at least 6 CPUs.

Let us consider the scenario with **short fragments**, where a large message is split into short messages, say 2048 bytes, and sent over the channel. The encryption is performed with the same key K and different IVs for each fragment – $IV1$, $IV2$, etc. In this case there is a generic approach to speed up *any* encryption algorithm by precomputing the keystreams for (K, $IV1$, $IV2$, ...). This technique can be applied to both AES and SNOW-V and from Table 4 we conclude that SNOW-V outperforms AES-256 also in this case.

The only scenario where SNOW-V is slower than AES-256 (in a single core setup) is when AES-256 performs the key setup only once, then uses the same context to prepare keystream for various IVs, with the speed 9Gbps. This relevant modes of operation are e.g. OFB, CTR, and GCM. In this case, SNOW-V is slower than AES-256 when the plaintext size is less than approximately 64 bytes.

6.3 Future AVX512

AVX512 is a new set of intrinsics utilizing wider 512-bit registers, and a subset of the AVX512 instructions is currently only available on high-end Intel CPUs. It is expected to be supported by consumer-grade CPUs in the near future. In this new set of intrinsics, there is an instruction to perform 4 AES encryption rounds in parallel `_mm512_aesenc_epi128()`, which would speed up AES by approximately x4 times.

SNOW-V will benefit from AVX512 as well. In the FSM, where we have to apply two AES encryption rounds, double XOR, and double ADD4x32 (today all done over 128-bit registers), we can in the future use wider registers, and the number of instructions could approximately be halved.

For the LFSRs, the new intrinsics will shrink the number of instructions as well. For example, AVX512 has the function `_mm512_ternarylogic_epi32()` that implements any user-defined 3-input Boolean function. Hence, an expression like `XOR(XOR(a, b), c)` can be substituted with a single `_mm512_ternarylogic_epi32(a, b, c, 0x96)`.

Both FSM and LFSR would utilize only half of a 512-bit registers while the number of instructions is reduced. Note that the second half of the registers can be used to perform another SNOW-V instance in parallel, with its own key and IV.

Thus, as a rough estimate, the speed of SNOW-V could be increased by x2-4 times.

7 AEAD mode of operation

The GMAC integrity and authentication algorithm specified in [Dwo07] can easily be adopted to work with SNOW-V to define an AEAD mode of operation. We will use notations from [Dwo07] in the following. In GCM, an unspecified block cipher is used in counter mode to encrypt the plaintext. Additionally, the block cipher is used to produce the final authentication tag T, and to derive the key H used in the function $GHASH_H$.

When using SNOW-V together with the $GHASH_H$ algorithm, the key H is the very first keystream output $z^{(0)}$. Then we continue to encrypt the n plaintext blocks using keystream output $z^{(1)}, \ldots, z^{(n)}$, feeding the ciphertext blocks into $GHASH_H$. Finally, we use keystream output $z^{(n+1)}$ as the final masking for the tag, similarly to the encrypted value of J_0 in [Dwo07].

SNOW-V works as described in section 2 with a single change. During initialization of the LFSRs, we set the lower part of the LFSR-B to the following hex values:

$$(b_7, b_6, \ldots, b_0) = (6D6F, 6854, 676E, 694A, 2064, 6B45, 7865, 6C41). \tag{12}$$

The hex values are the UTF8 encoding of the names of the authors.

An overview of how SNOW-V is used together with the $GHASH_H$ algorithm is shown in Figure 8. The padding of the Additional Authenticated Data (AAD) and how to concatenate the length of the AAD and the length of the ciphertext C and all other restrictions on plaintext length and change of IV from [Dwo07] remain. We have only defined a new way to derive the counter mode keystream, and the additional key and xor-value needed in the GCM algorithm.

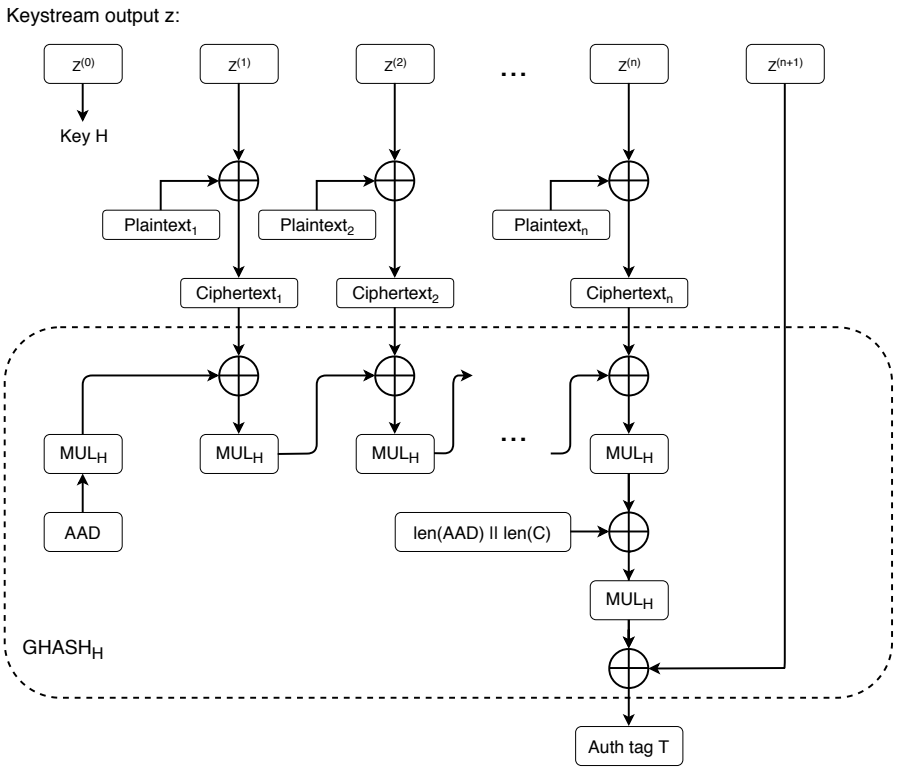

Fig. 8: How SNOW-V is used together with $GHASH_H$ to enable AEAD.

8 Conclusions

A new 128-bit stream cipher called SNOW-V is presented. It follows the design principles of the previous ciphers in the SNOW family, but leverages the AES round function instruction support found in many modern CPUs. Both hardware and software implementation aspects are discussed and especially a very compact 64-bit implementation using a single AES encryption round is given as an implementation alternative. Theoretical arguments are presented that implies a very high speed, reaching above 700 Gbps, for a full 128-bit implementation. In single core implementations in software, SNOW-V outperforms AES by a factor of approximately 6 for plaintext lengths above 2kB. Basic cryptanalysis of the new design is presented and SNOW-V is argued to be resistant against these attacks. Finally, an AEAD mode of operation based on the well known GCM scheme is given.

Test vectors and reference implementations are given in Appendices.

References

3GP. 3GPP. Work item on network functions virtualisation. `http://www.3gpp.org/more/1584-nfv`.

Bab95. Steve Babbage. Improved "exhaustive search" attacks on stream ciphers. In *European Convention on Security and Detection*. IET, 1995.

BG05. Olivier Billet and Henri Gilbert. Resistance of SNOW 2.0 against algebraic attacks. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes*

in Computer Science, pages 19–28, San Francisco, CA, USA, February 14–18, 2005. Springer, Heidelberg, Germany.

Bin. Bin Zhang et al. The ZUC-256 stream cipher. `http://www.is.cas.cn/ztzl2016/zouchongzhi/201801/W020180126529970733243.pdf`.

BS00. Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 1–13, Kyoto, Japan, December 3–7, 2000. Springer, Heidelberg, Germany.

BSW01. Alex Biryukov, Adi Shamir, and David Wagner. Real time cryptanalysis of A5/1 on a PC. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 1–18, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

BW99. Alex Biryukov and David Wagner. Slide attacks. In Lars R. Knudsen, editor, *Fast Software Encryption – FSE'99*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259, Rome, Italy, March 24–26, 1999. Springer, Heidelberg, Germany.

CHJ02. D. Coppersmith, S. Halevi, and C.S. Jutla. Cryptanalysis of stream ciphers with linear masking. In M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, volume 2442, pages 515–532, 2002.

CJM02. Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

CJS01. Vladimor V. Chepyzhov, Thomas Johansson, and Ben J. M. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

CKPS00. Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

DK08. Orr Dunkelman and Nathan Keller. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. *Information Processing Letters*, 107(5):133–137, 2008.

DS09. Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299, Cologne, Germany, April 26–30, 2009. Springer, Heidelberg, Germany.

Dwo07. Morris J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, Gaithersburg, MD, United States, 2007.

EJ01. Patrik Ekdahl and Thomas Johansson. SNOW – a new stream cipher. In *Proceedings of First Open NESSIE Workshop, KU-Leuven*, 2001.

EJ02. Patrik Ekdahl and Thomas Johansson. A New Version of the Stream Cipher SNOW. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2002.

EJT07. Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology - INDOCRYPT 2007: 8th International Conference in Cryptology in India*, volume 4859 of *Lecture Notes in Computer Science*, pages 268–281, Chennai, India, December 9–13, 2007. Springer, Heidelberg, Germany.

Gol97. Jovan Dj Golić. Cryptanalysis of alleged A5 stream cipher. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 239–255. Springer, 1997.

HJB09. Martin Hell, Thomas Johansson, and Lennart Brynielsson. An overview of distinguishing attacks on stream ciphers. *Cryptography and Communications*, 1(1):71–94, 2009.

HK18. Matthias Hamann and Matthias Krause. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. *Cryptography and Communications*, 10(5):959–1012, 2018.

HR02. P. Hawkes and G.G. Rose. Guess-and-determine attacks on SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595, pages 37–46, 2002.

HS05. Jin Hong and Palash Sarkar. New applications of time memory data tradeoffs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 353–372. Springer, 2005.

Int18. Intel Corporation. Intel Intrinsics Guide. Technical report, 2018. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

ITU17. ITU. Minimum requirements related to technical performance for IMT-2020 radio interface(s). Version 1.0, ITU, 2017. `https://www.itu.int/pub/R-REP-M.2410-2017`.

KY11. Aleksandar Kircanski and Amr M Youssef. On the sliding property of SNOW 3G and SNOW 2.0. *IET Information Security*, 5(4):199–206, 2011.

Lai94. Xuejia Lai. Higher order derivatives and differential cryptanalysis. In *Communications and Cryptography*, pages 227–233. Springer, 1994.

Mj06. O Saarinen Markku-juhani. Chosen-IV statistical attacks on eSTREAM stream ciphers. In *eSTREAM, ECRYPT Stream Cipher Project, Report 2006/013*. Citeseer, 2006.

NW06. Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In Matthew J. B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 144–162, Graz, Austria, March 15–17, 2006. Springer, Heidelberg, Germany.

oST01. National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

RMTA18. Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. Smashing the implementation records of aes s-box. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):298–336, May 2018. `https://tches.iacr.org/index.php/TCHES/article/download/884/835/`.

SA318. 3GPP SA3. TR 33.841 study on supporting 256-bit algorithms for 5G., 2018. `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422`.

SAG06. SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Version 1.1, ETSI/SAGE, 2006. `https://www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf`.

SAG11. SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. document 2: ZUC specification. Version 1.6, ETSI/SAGE, 2011. `https://www.gsma.com/aboutus/wp-content/uploads/2014/12/eea3eia3zucv16.pdf`.

Sam00. Samsung Electronics Co., Ltd. STD90/MDL90 0.35μm 3.3V CMOS Standard Cell Library for Pure Logic/MDL Products Databook, 2000. `https://www.digchip.com/datasheets/download_datasheet.php?id=935791&part-number=STD90`.

Sta10. Paul Stankovski. Greedy distinguishers and nonrandomness detectors. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010: 11th International Conference in Cryptology in India*, volume 6498 of *Lecture Notes in Computer Science*, pages 210–226, Hyderabad, India, December 12–15, 2010. Springer, Heidelberg, Germany.

Sta13. Paul Stankovski. *Cryptanalysis of Selected Stream Ciphers*, volume 50. Department of Electrical and Information Technology, Lund University, 2013.

UMHA16. Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. A high throughput/gate aes hardware architecture by compressing encryption and decryption datapaths — toward efficient cbc-mode implementation. Cryptology ePrint Archive, Report 2016/595, 2016. `https://eprint.iacr.org/2016/595`.

Vie07. Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. Cryptology ePrint Archive, Report 2007/413, 2007. `http://eprint.iacr.org/2007/413`.

ZXM15. Bin Zhang, Chao Xu, and Willi Meier. Fast correlation attacks over extension fields, large-unit linear approximation and cryptanalysis of SNOW 2.0. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

A Remarks about the maximum period of the LFSR structure

We can denote the LFSRs' state at time $t \geq 0$ as

$$S^{(t)} = (a_0^{(t)}, a_1^{(t)}, ..., a_{14}^{(t)}, a_{15}^{(t)}, b_0^{(t)}, b_1^{(t)}, ..., b_{14}^{(t)}, b_{15}^{(t)})$$

with 32 16-bit cells, i.e., 512 bits in total. If we consider the binary representation of the state, then the next state at $t+1$, $S^{(t+1)}$ can be written as,

$$S^{(t+1)} = S^{(t)}M$$

where M is the 512×512 state transition matrix.

Every part of the next state except $a_{15}^{(t+1)}, b_{15}^{(t+1)}$ is determined by a shift, that is $a_i^{(t+1)} = a_{i+1}^{(t)}, b_i^{(t+1)} = b_{i+1}^{(t)}$ for $i = 0, 1, ...14$, and the corresponding binary state transition submatrix for such update is identity matrix M_I with size 16×16. As for a_{15}, b_{15}, we can rewrite them in the polynomial form. Suppose the bases for finite field A and B are respectively $(1, \alpha, ..., \alpha^{15})$,$(1, \beta, ..., \beta^{15})$, then every state element can be expressed as a polynomial corresponding to the two bases.

For instance, a certain element e in the field A can be interpreted as $e = e_0 + e_1\alpha + , ..., + e_{14}\alpha^{14} + e_{15}\alpha^{15}$, where e_i denotes the i-th bit of e. Then,

$$e\alpha \bmod g^A(\alpha) = (e_{15}\alpha^{16} + e_{14}\alpha^{15} + , ..., + e_1\alpha^2 + e_0\alpha) \bmod g^A(\alpha)$$

Since

$$\alpha^{16} \bmod g^A(\alpha) = \alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1,$$

$e\alpha \bmod g^A(\alpha)$ can be expanded and rearranged as,

$$\begin{aligned}
&= e_{15}(\alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1) + e_{14}\alpha^{15} + , ..., + a^1\alpha^2 + a^0\alpha \\
&= (e_{15} + e_{14})\alpha^{15} + e_{13}\alpha^{14} + e_{12}\alpha^{13} + (e_{15} + e_{11})\alpha^{12} + (e_{15} + e_{10})\alpha^{11} + \\
&a^9\alpha^{10} + a^8\alpha^9 + (e_{15} + a^7)\alpha^8 + a^6\alpha^7 + a^5\alpha^6 + a^4\alpha^5 + a^3\alpha^4 + (e_{15} + a^2)\alpha^3 \\
&+ (e_{15} + a^1)\alpha^2 + (e_{15} + a^0)\alpha + e_{15} \\
&= (a^0, a^1, ..., e_{15})M_\alpha(1, \alpha, ..., \alpha^{15})^{\mathrm{T}}
\end{aligned}$$

From which we can deduce the matrix

$$M_\alpha = \begin{array}{c} 0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1 \\ 1\,1\,1\,1\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0\,0\,1 \end{array}$$

With the same method, we can also derive $M_{\alpha^{-1}}, M_\beta, M_{\beta^{-1}}$. Then we can rewrite the update for $a_{15}^{(t+1)}, b_{15}^{(t+1)}$ in a matrix form,

$$\begin{aligned} a_{15}^{(t+1)} &= b_0^{(t)} M_I + a_0^{(t)} M_\alpha + a_1^{(t)} M_I + a_8^{(t)} M_{\alpha^{-1}} \\ b_{15}^{(t+1)} &= a_0^{(t)} M_I + b_0^{(t)} M_\beta + b_3^{(t)} M_I + b_8^{(t)} M_{\beta^{-1}} \end{aligned}$$

Then the elaborate binary transition matrix for the LFSR structure can be written as,

$$M = \begin{array}{c|cccccccccccccccc} & 0 & 1 & \dots & 7 & \dots & 14 & 15 & \dots & 18 & \dots & 23 & \dots & 30 & 31 \\ 0 & & & & & & & M_\alpha & & & & & & & M_I \\ 1 & M_I & & & & & & M_I & & & & & & & \\ 2 & & M_I & & & & & & & & & & & & \\ \dots & & & \dots & & & & & & & & & & & \\ 8 & & & & M_I & & & M_{\alpha^{-1}} & & & & & & & \\ \dots & & & & & \dots & & & & & & & & & \\ 15 & & & & & & M_I & & & & & & & & \\ 16 & & & & & & & M_I & & & & & & & M_\beta \\ \dots & & & & & & & & \dots & & & & & & \\ 19 & & & & & & & & & M_I & & & & & M_I \\ \dots & & & & & & & & & & \dots & & & & \\ 24 & & & & & & & & & & & M_I & & & M_{\beta^{-1}} \\ \dots & & & & & & & & & & & & \dots & & \\ 31 & & & & & & & & & & & & & M_I & \end{array}$$

where every element in the 32×32 matrix is a 16×16 matrix and all the other blank places are 16×16 zero matrices. Then we can get the 512×512 transition matrix and some mathematical tools such as Sagemath, Matlab, Maple, et.al., can be employed to verify whether it is primitive. We employ the built-in function charpoly.() in Sagemath to get the characteristic polynomial,

which is,

$$
\begin{aligned}
&m(x) = \\
&x^{512} + x^{491} + x^{489} + x^{480} + x^{478} + x^{475} + x^{474} + x^{473} + x^{472} + x^{468} + x^{467} + \\
&x^{466} + x^{464} + x^{455} + x^{453} + x^{452} + x^{445} + x^{444} + x^{443} + x^{441} + x^{438} + x^{437} + \\
&x^{434} + x^{433} + x^{429} + x^{426} + x^{425} + x^{424} + x^{423} + x^{422} + x^{420} + x^{419} + x^{418} + \\
&x^{417} + x^{416} + x^{415} + x^{410} + x^{409} + x^{407} + x^{405} + x^{404} + x^{402} + x^{394} + x^{393} + \\
&x^{391} + x^{390} + x^{385} + x^{384} + x^{383} + x^{382} + x^{381} + x^{380} + x^{374} + x^{371} + x^{369} + \\
&x^{368} + x^{367} + x^{366} + x^{365} + x^{363} + x^{361} + x^{360} + x^{358} + x^{357} + x^{354} + x^{351} + \\
&x^{345} + x^{344} + x^{341} + x^{339} + x^{337} + x^{336} + x^{334} + x^{330} + x^{325} + x^{324} + x^{321} + \\
&x^{317} + x^{315} + x^{314} + x^{313} + x^{311} + x^{310} + x^{309} + x^{308} + x^{307} + x^{305} + x^{302} + \\
&x^{299} + x^{296} + x^{292} + x^{291} + x^{284} + x^{283} + x^{281} + x^{280} + x^{279} + x^{276} + x^{275} + \\
&x^{273} + x^{271} + x^{267} + x^{264} + x^{263} + x^{262} + x^{260} + x^{259} + x^{258} + x^{257} + x^{256} + \\
&x^{254} + x^{253} + x^{251} + x^{249} + x^{248} + x^{247} + x^{246} + x^{245} + x^{243} + x^{242} + x^{240} + \\
&x^{238} + x^{236} + x^{229} + x^{225} + x^{218} + x^{217} + x^{216} + x^{215} + x^{214} + x^{209} + x^{208} + \\
&x^{207} + x^{205} + x^{204} + x^{203} + x^{201} + x^{198} + x^{193} + x^{192} + x^{190} + x^{189} + x^{187} + \\
&x^{186} + x^{185} + x^{180} + x^{178} + x^{176} + x^{173} + x^{170} + x^{169} + x^{167} + x^{165} + x^{164} + \\
&x^{163} + x^{162} + x^{160} + x^{159} + x^{155} + x^{152} + x^{151} + x^{150} + x^{149} + x^{148} + x^{147} + \\
&x^{145} + x^{144} + x^{142} + x^{141} + x^{136} + x^{134} + x^{131} + x^{126} + x^{125} + x^{123} + x^{122} + \\
&x^{121} + x^{118} + x^{117} + x^{114} + x^{113} + x^{109} + x^{106} + x^{105} + x^{104} + x^{103} + x^{101} + \\
&x^{100} + x^{96} + x^{95} + x^{94} + x^{91} + x^{87} + x^{86} + x^{85} + x^{83} + x^{82} + x^{81} + x^{78} + \\
&x^{76} + x^{74} + x^{73} + x^{69} + x^{68} + x^{67} + x^{66} + x^{64} + x^{63} + x^{62} + x^{61} + x^{59} + \\
&x^{56} + x^{54} + x^{53} + x^{50} + x^{49} + x^{47} + x^{42} + x^{38} + x^{36} + x^{35} + x^{33} + x^{25} + \\
&x^{24} + x^{23} + x^{20} + x^{16} + x^{15} + x^{14} + x^{13} + x^{11} + x^{9} + x^{6} + x + 1
\end{aligned}
$$

Then we can verify it primitive by Sagemath, which indicates the LFSR structure has the maximum period $2^{512}-1$.

B Test Vectors

This section presents test vectors for SNOW-V with three different keys and IVs. The vectors are written with the **least significant byte** of the 128-bit word appearing to the left in the row. For the keys, the lower 128-bit part is written on the first row, followed by the high part on the second row.

```
key=
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
iv=
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Initialization z=
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
ea eb eb eb ea eb eb eb ea eb eb eb ea eb eb eb
55 e8 e8 dd 55 e8 e8 dd 55 e8 e8 dd 55 e8 e8 dd
c7 92 23 76 c7 92 23 76 c7 92 4e 97 64 34 43 17
3a 4a d9 94 33 45 c1 83 30 40 37 aa 80 76 3f ff
73 e6 32 79 d9 31 c5 5c 0a 9f 0e e4 3e 44 29 b3
16 05 29 4b 82 4d 86 dd c8 71 1c ee 4c 98 15 4f
a2 e2 b3 99 5a f2 50 cf 16 79 c8 e6 77 f3 04 da
```

69 5f f4 64 85 8e 2a 83 ca 90 e3 02 cf 08 fb 37
9e f6 60 ad 6b 29 4e 2d b3 a3 e3 fb c1 0c 1d 26
64 7d 11 16 ce 6d 25 23 72 f9 a7 ec af 9f 2f d6
2c 55 fc f0 cb c0 0c aa 13 47 a7 66 65 33 03 4f
43 e9 09 5d fd fe 09 1b dd 24 94 3f 2c 81 86 83
dd 03 fd 3a 57 69 04 ab ec bc b5 ff 78 ce da a7
Keystream z=
e8 49 de fe fd 57 2f ad d1 00 d2 17 b1 3e 6e 41
77 6b 21 9a d6 d9 d2 b2 44 e0 b8 dc 94 05 fe b3
70 6d 99 c0 54 b6 ea 83 0e 99 52 bd 8d a1 a7 5d
66 a7 78 10 cc fa 07 b1 1f cc 76 ea 20 7b 6c 48
58 81 6a 79 f1 a2 20 62 42 eb fb 4e 0a 8e f4 98
99 b4 d2 3b 09 c6 3d 8f 15 54 02 08 16 4e fe 48
4f 5e e6 90 ab 8b 35 af ab 7a 9c d2 db b1 09 42
33 a2 98 52 42 31 71 f0 14 9b 7a 8a c3 a7 cd fc

key=
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
iv=
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Initialization z=
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
d3 07 d2 07 d3 07 d2 07 d3 07 2d f8 2e f8 2d f8
65 f6 62 f6 65 f6 62 f6 65 f6 62 f6 65 f6 62 f6
fe 5a fe 5a 31 2a 3e 2a 31 2a d7 e8 d6 e8 d6 e8
8b 7e 8d 7e 7d f6 7e f6 7d f6 36 44 3d 44 c4 a3
21 53 ca 70 99 1e 0d d7 77 3c b6 cb 1f be 59 dd
46 71 82 0e d3 a0 21 1b d8 34 7e d6 e1 3d fa 15
81 65 cc e7 8e cf ac 77 19 02 ff bd 10 e4 10 87
c6 39 35 b3 e6 41 fb 2e a5 06 a6 44 20 54 0c 65
a9 f7 7a 9a e2 18 11 b1 51 1c 99 31 46 4b 40 67
64 98 4a f1 06 fc 61 f8 5d 6e 72 91 99 a5 88 be
96 d9 f0 28 f1 0d 1d e2 63 26 71 1c c4 07 5f 5e
80 35 43 9a e8 a6 0f 0d 8f df 40 86 10 76 e1 75
4d 5a 39 ab 8a e9 e7 48 94 c5 61 52 c8 2f 81 83
bf 8f 1a 4a 97 bb 10 34 96 2b c7 e6 99 e7 aa c1
72 56 b4 22 f0 9e 53 0d 44 84 7d a7 01 55 f2 fa
Keystream z=
cb d0 c5 56 03 be 71 f9 40 2f 9b c0 db 52 b9 11
e0 ea 84 37 4a bd fa 9d d2 95 88 4a 7f 1b c5 18
41 ad 99 e2 3c aa 66 c8 e2 f6 02 7b f4 96 af be
23 a6 be 7d 3d fb d2 72 04 de 26 d2 c6 3d b3 0b
6d 33 a7 2c 25 d8 ab 92 81 ac 33 c1 49 08 61 38
f8 d3 b4 47 d0 90 ba 1e dc 8e 19 79 42 ac 78 fe
1d d1 d1 77 0b 45 4a f5 89 f8 64 46 78 1b cd 8e
85 70 ba 21 bf e7 64 78 13 83 07 60 e5 6e 49 75

key=
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
iv=
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initialization z=

```
0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
66 32 f6 82 56 51 a5 a1 da 0d c3 1b cc 34 5c 24
a2 e8 90 89 b4 e1 3f e1 fb 07 fa e0 99 8e a6 e5
96 03 42 48 a8 a1 1d ed 88 5b 37 04 71 8b 0f 94
ea ac 99 ac e5 12 81 42 43 1b 4a 1b d8 74 b5 ef
c1 0a 6e b3 69 00 a6 db 08 15 bf 41 25 b8 b0 70
a0 d5 2c f1 75 e4 9e 93 72 e7 e5 55 6c 2d f4 ee
f6 8c 1a ec 8c a1 fb 02 ea 00 1b b0 8c 9c 3e 16
fe bf 40 93 33 cf 42 3d 16 a0 9e 17 ad 6d e0 96
75 d7 e7 1e d2 66 03 b1 1f 01 05 19 e2 f2 cb 96
05 86 bf 9a 4d ae 84 6d 3b bf 72 dc d6 31 4c 3e
7d d0 f3 bf c7 7b ab 4d ee ec db 50 81 f2 e0 cb
b7 82 cf ee 7e f8 7b 48 8d 5b 29 70 ed de d9 5e
7c a5 ce 30 91 2d 9e 3e 1a 66 57 db 23 4a fd 5c
cf 51 16 13 e2 c9 8a b2 cf 70 5a 61 94 8f 34 23
00 ea b0 79 bc 2b 21 08 01 18 fb 94 18 5b 16 dc
Keystream z=
de 6d 96 ed bd 71 70 75 43 29 9e 96 dc fd bf 9d
22 2f 7e 41 d1 74 5c de da 17 24 e4 10 92 6f 98
24 5c e4 de 2b 14 a9 e0 ff 85 26 8c 50 88 28 24
64 04 96 5b d2 10 c2 06 9e 68 23 51 80 bf ad 66
2a ec 4a f7 e9 3c ab 39 af 7e 61 76 69 98 04 9f
3b 0d 5c 99 6b 30 ed b2 51 a3 1c 30 c5 ab ac 3b
5a 96 37 0a 6c 6c a4 ab fd 0d e4 75 5f b9 06 53
c6 6b 21 a7 3e 6c b6 f8 81 55 2a cd 30 77 d1 65
```

Listing 1: Test vectors for three Key/IV pairs.

C SNOW-V 32-bit Reference Implementation in C/C++

```
// SNOW-V 32-bit Reference Implementation (Endianness-free)
#include <stdint.h>
#include <stdlib.h>

typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;

u8 SBox[256] =
{
    0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
    0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
    0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
    0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
    0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
    0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
    0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
    0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
    0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
    0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
    0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
    0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
```

```
    0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
    0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
    0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
    0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16
};

u32 AesKey1[4] = { 0, 0, 0, 0 };
u32 AesKey2[4] = { 0, 0, 0, 0 };

#define MAKEU32(a, b) (((u32)(a) << 16) | ((u32)(b) ))
#define MAKEU16(a, b) (((u16)(a) << 8) | ((u16)(b) ))

struct SnowV32
{
    u16 A[16], B[16];        // LFSR
    u32 R1[4], R2[4], R3[4]; // FSM

    void aes_enc_round(u32 * result, u32 * state, u32 * roundKey)
    {
#define ROTL32(word32, offset)   ((word32 << offset) | (word32 >> (32 - offset)))
#define SB(index, offset)        (((u32)(sb[(index) % 16])) << (offset * 8))
#define MKSTEP(j)\
    w = SB(j * 4 + 0, 3) | SB(j * 4 + 5, 0) | SB(j * 4 + 10, 1) | SB(j * 4 + 15, 2);\
    t = ROTL32(w, 16) ^ ((w << 1) & 0xfefefefeUL) ^ (((w >> 7) & 0x01010101UL) * 0x1b);\
    result[j] = roundKey[j] ^ w ^ t ^ ROTL32(t, 8)

        u32 w, t;
        u8 sb[16];
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 4; j++)
                sb[i * 4 + j] = SBox[(state[i] >> (j * 8)) & 0xff];

        MKSTEP(0);
        MKSTEP(1);
        MKSTEP(2);
        MKSTEP(3);
    }

    u16 mul_x(u16 v, u16 c)
    {   if (v & 0x8000)
            return(v << 1) ^ c;
        else
            return (v << 1);
    }

    u16 mul_x_inv(u16 v, u16 d)
    {   if (v & 0x0001)
            return(v >> 1) ^ d;
        else
            return (v >> 1);
    }

    void fsm_update(void)
```

```
{   u32 R1temp[4];
    memcpy(R1temp, R1, sizeof(R1));

    for (int i = 0; i < 4; i++)
    {   u32 T2 = MAKEU32(A[2 * i + 1], A[2 * i]);
        R1[i] = (T2 ^ R3[i]) + R2[i];
    }

    aes_enc_round(R3, R2, AesKey2);
    aes_enc_round(R2, R1temp, AesKey1);
}

void lfsr_update(void)
{
    for (int i = 0; i < 8; i++)
    {   u16 u = mul_x(A[0], 0x990f) ^ A[1] ^ mul_x_inv(A[8], 0xcc87) ^ B[0];
        u16 v = mul_x(B[0], 0xc963) ^ B[3] ^ mul_x_inv(B[8], 0xe4b1) ^ A[0];

        for (int j = 0; j < 15; j++)
        {   A[j] = A[j + 1];
            B[j] = B[j + 1];
        }

        A[15] = u;
        B[15] = v;
    }
}

void keystream(u8 * z)
{
    for (int i = 0; i < 4; i++)
    {   u32 T1 = MAKEU32(B[2 * i + 9], B[2 * i + 8]);
        u32 v = (T1 + R1[i]) ^ R2[i];
        z[i * 4 + 0] = (v >>  0) & 0xff;
        z[i * 4 + 1] = (v >>  8) & 0xff;
        z[i * 4 + 2] = (v >> 16) & 0xff;
        z[i * 4 + 3] = (v >> 24) & 0xff;
    }

    fsm_update();
    lfsr_update();
}

void keyiv_setup(u8 * key, u8 * iv)
{
    for (int i = 0; i < 8; i++)
    {   A[i] = MAKEU16(iv[2 * i + 1], iv[2 * i]);
        A[i + 8] = MAKEU16(key[2 * i + 1], key[2 * i]);
        B[i] = 0x0000;
        B[i + 8] = MAKEU16(key[2 * i + 17], key[2 * i + 16]);
    }

    for (int i = 0; i < 4; i++)
```

```
            R1[i] = R2[i] = R3[i] = 0x00000000;

        for (int i = 0; i < 16; i++)
        {   u8 z[16];
            keystream(z);

            for (int j = 0; j < 8; j++)
                A[j + 8] ^= MAKEU16(z[2 * j + 1], z[2 * j]);

            if (i == 14)
                for (int j = 0; j < 4; j++)
                    R1[j] ^= MAKEU32(MAKEU16(key[4 * j + 3], key[4 * j + 2]),
                                     MAKEU16(key[4 * j + 1], key[4 * j + 0]));
            if (i == 15)
                for (int j = 0; j < 4; j++)
                    R1[j] ^= MAKEU32(MAKEU16(key[4 * j + 19], key[4 * j + 18]),
                                     MAKEU16(key[4 * j + 17], key[4 * j + 16]));
        }
    }
};
```

D SNOW-V Reference Implementation with AVX2

```
// SNOW-V Reference Implementation with AVX2 (Little endian)
#include <immintrin.h>
#define vpset16(value) _mm256_set1_epi16(value)

struct SnowV256
{
    // Constants
    __m256i _mul, _inv; // for mul by [β | α] and [β^-1 | α^-1]
    __m128i zero128;    // AES RoundKeys

    // State
    __m256i hi, lo;     // LFSR
    __m128i R1, R2, R3; // FSM

    SnowV256()
    {
        _mul = _mm256_blend_epi32(vpset16( 0x990f), vpset16( 0xc963), 0xf0);
        _inv = _mm256_blend_epi32(vpset16(-0xcc87), vpset16(-0xe4b1), 0xf0);
        zero128 = _mm_setzero_si128();
    }

    inline __m256i mul_x(__m256i s)
    {
        return _mm256_xor_si256(
                    _mm256_and_si256(_mul,
                        _mm256_srai_epi16(s, 15)),
                    _mm256_slli_epi16(s, 1));
    }

    inline __m256i mul_x_inv(__m256i s)
```

```
    {
        return _mm256_xor_si256(
                    _mm256_sign_epi16(_inv,
                        _mm256_slli_epi16(s, 15)),
                    _mm256_srli_epi16(s, 1));
    }

    inline void lfsr_update(void)
    {
        __m256i hi_old = hi;
        hi = _mm256_xor_si256(
                _mm256_xor_si256(
                    _mm256_blend_epi32(
                        _mm256_alignr_epi8(hi, lo, 1 * 2),
                        _mm256_alignr_epi8(hi, lo, 3 * 2), 0xf0),
                    _mm256_permute4x64_epi64(lo, 0x4e)),
                _mm256_xor_si256(mul_x_inv(hi), mul_x(lo)));
        lo = hi_old;
    }

    inline void fsm_update(void)
    {
        __m128i T2 = _mm256_castsi256_si128(lo);
        __m128i newR1 = _mm_add_epi32(R2, _mm_xor_si128(R3, T2));
        R3 = _mm_aesenc_si128(R2, zero128);
        R2 = _mm_aesenc_si128(R1, zero128);
        R1 = newR1;
    }

    inline __m128i keystream(void)
    {
        __m128i T1 = _mm256_extracti128_si256(hi, 1);
        __m128i z = _mm_xor_si128(R2, _mm_add_epi32(R1, T1));
        fsm_update();
        lfsr_update();
        return z;
    }

    inline void keyiv_setup(const unsigned char * key, const unsigned char * iv)
    {
        hi = _mm256_lddqu_si256((const __m256i*)key);
        lo = _mm256_zextsi128_si256(_mm_lddqu_si128((const __m128i*)iv));
        R1 = R2 = R3 = _mm_setzero_si128();

        for (int i = 0; i < 15; ++i)
            hi = _mm256_xor_si256(hi, _mm256_zextsi128_si256( keystream() ));

        R1 = _mm_xor_si128(R1, _mm_lddqu_si128((const __m128i*)(key + 0)));
        hi = _mm256_xor_si256(hi, _mm256_zextsi128_si256( keystream() ));
        R1 = _mm_xor_si128(R1, _mm_lddqu_si128((const __m128i*)(key + 16)));
    }
};
```