A new SNOW stream cipher called SNOW-V

Patrik Ekdahl\textsuperscript{1}, Thomas Johansson\textsuperscript{2}, Alexander Maximov\textsuperscript{1} and Jing Yang\textsuperscript{2}

\textsuperscript{1} Ericsson Research, Lund, Sweden
\{patrik.ekdahl,alexander.maximov\}@ericsson.com
\textsuperscript{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 one of the core algorithm for integrity and confidentiality in both UMTS and LTE. 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 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 (128-EIAx) and confidentiality (128-EEAx), based on three different primitives SNOW 3G [SAG06], AES [JoST01], and ZUC [SAG11]. The algorithms used in UMTS and LTE are all using the 128-bit key size, and are depicted in Table 1.
Table 1: Base algorithms used in UMTS and LTE for integrity and confidentiality.

<table>
<thead>
<tr>
<th></th>
<th>UMTS</th>
<th>LTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integrity</td>
<td>Encryption</td>
</tr>
<tr>
<td>Kasumi</td>
<td>UIA1</td>
<td>UEA1</td>
</tr>
<tr>
<td>SNOW 3G</td>
<td>UIA2</td>
<td>UEA2</td>
</tr>
<tr>
<td>AES</td>
<td>EIA1</td>
<td>EEA1</td>
</tr>
<tr>
<td>ZUC</td>
<td>EIA2</td>
<td>EEA2</td>
</tr>
<tr>
<td></td>
<td>EIA3</td>
<td>EEA3</td>
</tr>
</tbody>
</table>

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 128-EIAx and 128-EEAx 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 9 Gbps in a pure software implementation, which is far too low for the targeted speed of 20 Gbps downlink in the 5G system (see, e.g., [ITU17]).

In this paper we revise the SNOW 2.0/ SNOW 3G design 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 we describe how authentication can be included in an AEAD mode of operation. Then software implementation aspects are considered in Section 5, and in Section 6 software performance results and implementation aspects using future SIMD instruction set are presented. We end the paper with some conclusions in Section 7.
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.

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 \( a_{15} \ldots a_0 \) and \( b_{15} \ldots b_0 \) 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

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

and the elements of LFSR-B are generated by

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

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 \( (a_{15}^{(t)}, a_{14}^{(t)}, \ldots, a_0^{(t)}, b_0^{(t)}, b_1^{(t)}, b_{15}^{(t)}) \) for LFSR-A and LFSR-B. Referring to Figure 1, the elements \( a_0^{(t)} \) and \( b_0^{(t)} \) are the elements to first exit 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 $a_i^{(t)}$, $0 \leq i \leq 15$ are the values of the cells in the LFSR-A at time $t$. In the case of $\alpha$ and $\beta$, the notation $\alpha^{-1}$ and $\beta^{-1}$ 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
\[ T1^{(t)} = (b_{15}^{(8t)}, b_{14}^{(8t)}, \ldots, b_8^{(8t)}), \tag{5} \]
\[ T2^{(t)} = (a_7^{(8t)}, a_6^{(8t)}, \ldots, a_0^{(8t)}). \tag{6} \]

![Figure 2: Mapping the 16-bit words of the LFSRs into 128-bit words $T1$ and $T2$.](image)

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, $\odot$ denotes a bitwise XOR operation, and $B_{32}$ 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
\[
z(t) = (R1(t) \oplus_{32} T1(t)) \oplus R2(t).
\] (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

\[
AES^K(IN, KEY).
\]

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.

We can now write the update expressions for the registers as
\[
R1^{(t+1)} = \sigma(R2^{(t)} \oplus_{32} (R3^{(t)} \oplus T2^{(t)})),
\] (8)
\[
R2^{(t+1)} = AES^K(R1^{(t)}, C1),
\] (9)
\[
R3^{(t+1)} = AES^K(R2^{(t)}, C2).
\] (10)

The values of the two round key constants \( C1 \) and \( C2 \) are set to zero, and \( \sigma \) is a byte-oriented permutation given by
\[
\sigma = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15].
\] (11)

This should be interpreted as byte 0 is moved to position 0, byte 4 is moved to position 1, byte 8 is moved to position 2, and so on. Position 0 is the least significant byte in accordance to the mapping described above. The chosen \( \sigma \) implements the transposition of the mapped AES state matrix.
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

$$IV = (i v_7, i v_6, \ldots, i v_1, i v_0),$$

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

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

$$(a_{15}, a_{14}, \ldots, a_0) \leftarrow (k_7, k_6, \ldots, k_0, i v_7, i v_6, \ldots, i v_0)$$

and

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

Note that $(b_7, \ldots, b_0)$ will have a non-zero value when SNOW-V is used in AEAD-mode, see Section 4.

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 $R_1$ 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 Algorithm 1 clarifies the procedure.

**Algorithm 1 SNOW-V initialization**

1: **procedure** Initialization($K, IV$)
2:  $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (k_7, k_6, \ldots, k_0)$
3:  $(a_7, a_6, \ldots, a_0) \leftarrow (i v_7, i v_6, \ldots, i v_0)$
4:  $(b_{15}, b_{14}, \ldots, b_8) \leftarrow (k_{15}, k_{14}, \ldots, k_8)$
5:  $(b_7, b_6, \ldots, b_0) \leftarrow (0, 0, \ldots, 0)$
6:  $R_1, R_2, R_3 \leftarrow 0, 0, 0$
7:  **for** $t = 1 \ldots 16$ **do**
8:  $T_1 \leftarrow (b_{15}, b_{14}, \ldots, b_8)$
9:  $z \leftarrow (R_1 \oplus_{32} T_1) \oplus R_2$
10:  $FSM update ()$
11:  $LFSR update ()$
12:  $(a_{15}, a_{14}, \ldots, a_8) \leftarrow (a_{15}, a_{14}, \ldots, a_8) \oplus z$
13:  **if** $t = 15$ **then** $R_1 \leftarrow R_1 \oplus (k_7, k_6, \ldots, k_0)$
14:  **if** $t = 16$ **then** $R_1 \leftarrow R_1 \oplus (k_{15}, k_{14}, \ldots, k_8)$

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

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
Algorithm 2 SNOW-V algorithm

1: procedure SNOW-V($K, IV$)
2:   Initialization($K, IV$)
3:   while more keystream blocks needed do
4:     $T_1 \leftarrow (b_{15}, b_{14}, \ldots, b_8)$
5:     $z \leftarrow (R_1 \oplus 32 T_1) \oplus R_2$
6:     FSMupdate()
7:     LFSRupdate()
8:   Output keystream symbol $z$

Algorithm 3 LFSR update algorithm

1: procedure LFSRupdate()
2:   for $i = 0 \ldots 7$ do
3:     $tmp_a \leftarrow b_0 + \alpha a_0 + a_1 + \alpha^{-1} a_8 \pmod{g^4(\alpha)}$
4:     $tmp_b \leftarrow a_0 + \beta b_0 + b_3 + \beta^{-1} b_8 \pmod{g^B(\beta)}$
5:     $(a_{15}, a_{14}, \ldots, a_0) \leftarrow (tmp_a, a_{15}, \ldots, a_1)$
6:     $(b_{15}, b_{14}, \ldots, b_0) \leftarrow (tmp_b, b_{15}, \ldots, b_1)$

Algorithm 4 FSM update algorithm

1: procedure FSMupdate()
2:   $T_2 \leftarrow (a_7, a_6, \ldots, a_0)$
3:   $tmp \leftarrow R_2 \oplus R_3 \oplus T_2$
4:   $R_3 \leftarrow AES(R_2)$ \hspace{1cm} \triangleright Note that the round keys for these AES encryption rounds are $C1 = C2 = 0$
5:   $R_2 \leftarrow AES(R_1)$ \hspace{1cm} \triangleright encryption rounds are $C1 = C2 = 0$
6:   $R_1 \leftarrow \sigma(tmp)$

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. In Subsection 3.1 and Subsection 3.4 we give some cryptanalysis results also for the case when $\sigma$ is replaced by the identity mapping $\sigma_0$. This may aid the understanding of the strength of different methods of cryptanalysis.

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 mixed by running the cipher a few rounds (16 for SNOW-V) without giving outputs until the state becomes random-like. It should be difficult for a cryptanalyst to predict the keystream or get some information about the initial key according to the output after initialization. It then becomes vital to make sure
that the key/IV loading has no fatal flaws and the initialization round is carefully chosen 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 rationales behind are that: 1) a stream cipher can be regarded as a series of "black box" Boolean functions $f_i$ with the keystream bits as the outputs and key/IV as the inputs, 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 [Sta13]. If one can distinguish the output from a random distribution under some key/IV settings, the cipher is believed to be unsafe. 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, which are called a cube. The truth tables of the Boolean functions can be obtained, which are further used to compute the monomial coefficients of the ANF and compared with expected values. If there exists a big gap between them, the output is believed to be non-random. 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 one more bit is added from the remaining bits, we select the bit resulting in the worst randomness result until we get a 24-bit set (larger sets 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 for SNOW-V under permutation $\sigma$ and no permutation $\sigma_0$ (identity mapping). The results for 1, 2 and 3-bit sets are obtained through exhaustive search, while for the sets with larger sizes, the results are based on greedy search from the obtained worst 3-bit set. It can be seen that the performance under $\sigma$ is better, indicating the cipher is mixed better and faster under this case. Under both cases, roughly the first 7 rounds fail the MDM test, ensuring a large security gap to 16 rounds. One can also note that the number of rounds 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. Recently, cube attacks based on division property have got a lot of attention and research. Division property, proposed by Todo et al. in [Tod15], is a generalization of integral property being used to find integral distinguishers or launch cube attacks. Unlike traditional experimental attacks where the ciphers are regarded as black boxes, division property based attacks explore the internal structures of ciphers and trails of division property according to propagation rules for different components. These rules could be expressed with some (in)equalities and the attacks are modeled as MILP (Mixed Integer Linear Programming) problems with certain constrains and objective functions. Some optimization tools such as Gurobi, Cplex could then be employed to solve the problems very efficiently and identify if the attacks are feasible or not under certain cases.

In [TIHM17], division property was introduced into cube attacks to evaluate the set of the key bits $J$ involved in the superpoly given a certain cube $I$. After obtaining the set $J$, attackers are able to recover the superpoly by building the truth table and further to
recover partial key by querying the encryption oracle. The time complexity of recovering the superpoly is $2^{|I|+|J|}$, which is feasible when $|I| + |J|$ is smaller than the security bit level. Authors in [WHT+18] further improved the attack by exploiting various algebraic properties of superpolies. They introduced a technique to evaluate the upper bound of the algebraic degree, denoted as $d$, of the superpoly to avoid recovering the coefficients of monomials with degrees larger than $d$. Hence, only $(\frac{|J|}{d})$ coefficients need to be recovered and the time complexity reduces to $2^{|I|} \times (\frac{|J|}{d})$.

We evaluate SNOW-V with division property based cube attacks using the method described above. The MILP model of division property for SNOW-V which can evaluate all division trails when the initialization rounds are reduced to $R$ is illustrated in Algorithm 5. We first load key and IV bits to the LFSR state and initialize $R_1$, $R_2$, $R_3$ according to the initialization specification of SNOW-V (Algorithm 1). Since $K$ and $IV$ are loaded into LFSR states through one-by-one mapping, we do not introduce more intermediate variables in the model. In every round, we deal with the first and last 7 rounds of LFSR update differently since they have different propagation trails: the procedure $\text{LFSRupdate}$ for the last 7 LFSR updates consists of $\text{copy, xor, multiplication}$ (with $\alpha, \beta, \alpha^{-1}, \beta^{-1}$), $\text{shift}$ to update the LFSR state while for the first round, the function $\text{LFSRupdateFirstIter}$ involves in more complicated $\text{copy}$ trails to get $T1$ and $T2$. The $\text{funcAES}$ function is four parallel AES rounds consisting of $\text{sbox, shiftrow}$ and $\text{mixcolumn}$ whose propagation rules could be found in [Tod15]. The function $\text{funcModAdd}$ consists of 4 parallel modular additions whose propagation rule has been established in [SWW17] and $\text{funcXor}$ expresses $\text{xor}$ for 128 bits. The functions of $\text{multiplication}$ and $\text{mixcolumn}$ have consistent characteristic: the sum of input equals to the sum of output in terms of division property while the functions $\text{shift}$, $\text{shiftrow}$ and $\text{sigma}$ only permute the division property vectors.

<table>
<thead>
<tr>
<th>Rounds</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>cube size $</td>
<td>I</td>
<td>$</td>
<td>15</td>
<td>40</td>
<td>128</td>
</tr>
<tr>
<td>degree $d$</td>
<td>76(30)</td>
<td>248(79)</td>
<td>256(149)</td>
<td>256(254)</td>
<td>256</td>
</tr>
<tr>
<td>involved key size $</td>
<td>J</td>
<td>$</td>
<td>184(127)</td>
<td>256(96)</td>
<td>256</td>
</tr>
<tr>
<td>time complexity</td>
<td>$2^{125.5}(2^{112.1})$</td>
<td>$2^{256}(2^{136})$</td>
<td>$2^{256}$</td>
<td>$2^{256}$</td>
<td>$2^{256}$</td>
</tr>
</tbody>
</table>

We tried different cubes and Table 2 listed some examples under which adversaries
Algorithm 5 MILP model of division property for SNOW-V

1: procedure SnowCore(round $R$)
2: Prepare empty MILP Model $\mathcal{M}$
3: $\mathcal{M}.var \leftarrow s_i^R$ for $i \in \{0, 1, ..., 511\}$ and $R_{10}, R_2, R_3$ for $i \in \{0, 1, ..., 127\}$
4: $(\mathcal{M}, s^0, R_1^0, R_2^0, R_3^0) = \text{init}(\mathcal{M}, K, IV)$
5: for $r = 1$ to $R - 1$ do
6: $(\mathcal{M}, T_1, T_2, s^r) = \text{LFSRupdateFirstIter}(\mathcal{M}, s^{r-1})$
7: $(\mathcal{M}, X) = \text{funcModAdd}(\mathcal{M}, R_1^{r-1}, T_1)$
8: $(\mathcal{M}, Z^r) = \text{funcXor}(\mathcal{M}, X, R_2^{r-1})$
9: $(\mathcal{M}, Y) = \text{funcXor}(\mathcal{M}, R_3^{r-1}, T_2)$
10: $(\mathcal{M}, U) = \text{funcModAdd}(\mathcal{M}, R_2^{r-1}, Y)$
11: $(\mathcal{M}, R_1^r) = \text{sigma}(\mathcal{M}, U)$
12: $(\mathcal{M}, R_3^r) = \text{funcAES}(\mathcal{M}, R_2^{r-1})$
13: $(\mathcal{M}, R_2^r) = \text{funcAES}(\mathcal{M}, R_1^{r-1})$
14: for $i = 1$ to 7 do
15: $(\mathcal{M}, s^{r,i}) = \text{LFSRupdate}(\mathcal{M}, s^{r,i-1})$
16: $(\mathcal{M}, s_{128...255}^r) = \text{funcXor}(\mathcal{M}, s_{128...255}^{r-7}, Z^r)$
17: $(\mathcal{M}, s_{0...127}^{511}) = (\mathcal{M}, s_{0...127}^{511})$
18: $(\mathcal{M}, X) = \text{funcModAdd}(\mathcal{M}, R_1^{R-1}, s_{128...255}^{R-1})$
19: $(\mathcal{M}, Z^R) = \text{funcXor}(\mathcal{M}, X, R_2^{R-1})$
20: for $i = 0$ to 383 do
21: $\mathcal{M}.con \leftarrow s_{1}^{R-1} = 0$
22: for $i = 0$ to 127 do
23: $\mathcal{M}.con \leftarrow R_3_i^{R-1} = 0$
24: $\mathcal{M}.con \leftarrow \sum Z_i^R = 1$
25: return $\mathcal{M}$

have good advantages for permutations $\sigma$ and $\sigma_0$. The time complexity in the table shows the time complexity of superpoly recovery. One can see, all key bits are involved in the superpolies from the 4-th round under permutation $\sigma$ while 5-th round under $\sigma_0$, which indicates the good mixing effect after 4(or 5) rounds and guarantees enough security gap out of 16 rounds. The results match well with the research on division property based distinguishing attacks on AES in [Tod15], where only 4-round distinguisher could be found with $2^{120}$ plaintexts and the conclusion in [SWW16] that integral distinguishers for AES based on division property covering more than four rounds probably do not exist.

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 ($2 \times 256$-bit LFSRs + $3 \times 128$-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 as 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.

We recall the FSM equations:

\[ z(t) = (R_1^{(t)} \oplus_{32} T_1^{(t)}) \oplus R_2^{(t)}, \]
\[ R_1^{(t+1)} = \sigma(R_2^{(t)} \oplus_{32} (R_3^{(t)} \oplus T_2^{(t)})), \]
\[ R_2^{(t+1)} = AES^R(R_1^{(t)}), \]
\[ R_3^{(t+1)} = AES^R(R_2^{(t)}). \]

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 \( \hat{R}_1, \hat{R}_2, \hat{R}_3 \). Then we have

\[ z(t) = (\hat{R}_1 \oplus_{32} T_1^{(t)}) \oplus \hat{R}_2, \]
\[ R_1^{(t+1)} = \sigma(\hat{R}_2 \oplus_{32} (\hat{R}_3 \oplus T_2^{(t)})), \]
\[ R_2^{(t+1)} = AES^R(\hat{R}_1), \]
\[ R_3^{(t+1)} = AES^R(\hat{R}_2). \]

For time \( t + 1 \),

\[ z^{(t+1)} = (\sigma(\hat{R}_2 \oplus_{32} (\hat{R}_3 \oplus T_2^{(t)})) \oplus_{32} T_1^{(t+1)}) \oplus AES^R(\hat{R}_1). \]

It is now straight-forward to see that since \( z(t) \) depends only on \( \hat{R}_1 \) and \( z^{(t+1)} \) depends on both \( \hat{R}_1 \) and \( \hat{R}_2 \), there can be no biased linear approximation using only \( z(t) \) and \( z^{(t+1)} \). So the minimum number of equations that needs to be considered is three. To simplify coming derivations, we introduce the expressions for \( z^{(t-1)} \), i.e.,

\[ z^{(t-1)} = ((AES^R)^{-1}(\hat{R}_2 \oplus_{32} T_1^{(t-1)}) \oplus (AES^R)^{-1}(\hat{R}_3), \]

and seek a biased expression involving \( z^{(t-1)}, z(t), z^{(t+1)} \).
Consider all 128-bit variables as column vectors of 16 bytes. Then \( AES^R(X) \) can be written as \( L \cdot S(X) \), where \( S \) is an application of the 16 AES S-Boxes, one on each byte of \( X \), and \( L \) is a linear transformation over the 16 bytes (including both ShiftRow and MixColumn). Furthermore, \( (AES^R)^{-1}(X) = S^{-1}(L^{-1} \cdot X) \). We now introduce simplifications to the SNOW-V algorithm to show the best results on versions weaker than the proposed algorithm.

### 3.4.1 Analysis of the bias using \( \sigma_0 \) and using byte-wise addition \( \oplus_8 \)

We assume that there is no byte-wise permutation, i.e., \( \sigma_0 \) is just the identity. Furthermore, instead of \( \oplus_{12} \) we consider a modulo addition which is restricted to each byte, denoted \( \oplus_8 \). With this simplification all operations are byte-oriented and the investigation of linear approximations is easier.

Let us now seek a byte-oriented linear approximation. We examine the following three equations.

\[
\begin{align*}
z^{(t-1)} &= (S^{-1}(L^{-1} \cdot \hat{R}2) \oplus_{8} T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R}3), \\
z^{(t)} &= (\hat{R}1 \oplus_{8} T1^{(t)}) \oplus \hat{R}2, \\
L^{-1}z^{(t+1)} &= L^{-1}((\hat{R}2 \oplus_{8} (\hat{R}3 \oplus T2^{(t)})) \oplus_{8} T1^{(t+1)}) \oplus S(\hat{R}1).
\end{align*}
\]

Let \( X_i \) denote the \( i \)-th byte of a vector of bytes \( X \). Summing up the three \( z \)-terms on the left hand side and taking byte 0 gives us

\[
[z^{(t-1)} \oplus z^{(t)} \oplus L^{-1}z^{(t+1)}]_0 = [N1 \oplus N2 \oplus N3 \oplus T1^{(t)} \oplus T1^{(t-1)} \oplus L^{-1}(T2^{(t)} \oplus T1^{(t+1)})]_0,
\]

where

\[
\begin{align*}
N1 &= (S^{-1}(L^{-1} \cdot \hat{R}2) \oplus_{8} T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R}3) \oplus T1^{(t-1)}, \\
N2 &= (\hat{R}1 \oplus_{8} T1^{(t)}) \oplus S(\hat{R}1) \oplus T1^{(t)}, \\
N3 &= L^{-1}((\hat{R}2 \oplus_{8} (\hat{R}3 \oplus T2^{(t)})) \oplus_{8} T1^{(t+1)}) \oplus T2^{(t)} \oplus T1^{(t+1)} \oplus \hat{R}2 \oplus_{8} \hat{R}2 \oplus_{8} \hat{R}3 \\
&= L^{-1}(\hat{R}2) \oplus_{8} T1^{(t+1)} \oplus \hat{R}3.
\end{align*}
\]

The general idea is that \([N1 \oplus N2 \oplus N3]_0\) is a biased distribution. It is true because the (types of) noise variables \( n1, n2, n3 \) defined below are biased, which can be checked:

\[
\begin{align*}
n1 &= (x \oplus_{8} y) \oplus x \oplus y, \\
n2 &= (x \oplus_{8} y) \oplus S(x) \oplus y, \\
n3 &= (x \oplus_{8} y) \oplus x \oplus S(x) \oplus y.
\end{align*}
\]

Each of the noise terms \( N1, N2, N3 \) given above are of the above types and hence the sum of them can also be a biased distribution.

**Computation of the bias:** it remains to compute the bias and the \( N3_0 \) part is the most complicated case (although we are computing the bias of \([N1 \oplus N2 \oplus N3]_0\) as there is a dependence between \( N1 \) and \( N3 \)). The noise \( N3 \) at byte 0 can be rewritten as:

\[
N3_0 = \sum_{i=0}^{3}(c_i \cdot U_i) \oplus \hat{R}2_0 \oplus S^{-1}\left(\sum_{i=0}^{3}(c_i \cdot \hat{R}2_i)\right) \oplus S^{-1}\left(\sum_{i=0}^{3}(c_i \cdot \hat{R}3_i)\right),
\]

where the coefficients are \( c = [e \ b \ d \ 9] \). First note that \( U_i \) is the variable corresponding to \( U_i = (\hat{R}2_i \oplus_{8} (\hat{R}3_i \oplus T2^{(t)})) \oplus_{8} T1^{(t+1)} \oplus T2^{(t)} \oplus T1^{(t+1)} \), for \( 0 \leq i \leq 3 \). Note also that
We try all combinations of the first bytes of the inputs $\mathcal{R}_2, \mathcal{R}_3, T_2^{(t)}, T_1^{(t+1)}$ which leads to a partial sum as a 3-byte value denoted $A|B|C$, as marked in the equations above. Thus, we can compute the distribution $D_0(A|B|C)$. We also compute similar distributions for every "slice" of the inputs, $D_k(A|B|C), k = 0, 1, 2, 3$, corresponding to inputs $\mathcal{R}_2, \mathcal{R}_3, T_2^{(t)}, T_1^{(t+1)}$.

The XOR-convolution of the computed distributions of the partial linear expressions gives the combined distribution of the triple $A|B|C$ over all possible 32-bit inputs. Having that total distribution, it is then easy to construct the 8-bit distribution of $N_3$.

Let $D$ be a distribution of an noise variable $X$. Then we compute the bias denoted $\epsilon(X) = \epsilon(D)$ using the Squared Euclidean Imbalance as in [ZXM15], defined through

$$
\epsilon(D) = |D| \sum_{x=0}^{|D|-1} \left( D(x) - \frac{1}{|D|} \right)^2.
$$

The number of samples needed to distinguish a source of noise $D$ from random is roughly $O(1/\epsilon)$. The results when we use 8-bit adders $\oplus$ instead of the original 32-bit ones, keeping all operations within bytes. We try all combinations of the first bytes of the inputs $\mathcal{R}_2, \mathcal{R}_3, T_2^{(t)}, T_1^{(t+1)}$ which leads to a partial sum as a 3-byte value denoted $A|B|C$, as marked in the equations above. Thus, we can compute the distribution $D_0(A|B|C)$. We also compute similar distributions for every "slice" of the inputs, $D_k(A|B|C), k = 0, 1, 2, 3$, corresponding to inputs $\mathcal{R}_2, \mathcal{R}_3, T_2^{(t)}, T_1^{(t+1)}$.

The number of samples needed to distinguish a source of noise $D$ from random is roughly $O(1/\epsilon)$. The results when we use 8-bit adders $\oplus$ instead of the original 32-bit ones, keeping all operations within bytes.  

**3.4.2 Analysis of the bias using $\sigma_0$ and using 32-bit $\oplus$**

In order to deal with 32-bit adders $\oplus$, and $\sigma$ as identity are as follows: $\epsilon(N1) > 1, \epsilon(N2) \approx 2^{-2.9}, \epsilon(N3) \approx 2^{-46.3}$, and $\epsilon(N_{tot}) \approx 2^{-53.5}, \epsilon(2 \times N_{tot}) \approx 2^{-106.8}, \epsilon(3 \times N_{tot}) \approx 2^{-160.2}, \epsilon(4 \times N_{tot}) \approx 2^{-213.6}$.

Here the $N1, N2, N3$ denotes the partial noise as described above and $N_{tot}$ represents the full noise of a single approximation, i.e., $N_{tot} = \left[N1 \oplus N2 \oplus N3\right]_0$. Finally, $i \times N_{tot}$ denotes the noise obtained by a sum of $i$ such independent noise terms.

**3.4.3 Using the bias in a fast correlation attack**

A detected bias can be used in a distinguishing attack if one can find, say, a weight 3 or weight 4 multiple of the polynomial corresponding to the byte-oriented sequence $W(t) = [T1^{(t)} \oplus T1^{(t-1)} \oplus L^{-1}(T2^{(t)} \oplus T1^{(t+1)})]_0$. Since the LFSR feedback is defined in $F_{2^{16}}$, it can be rewritten in $F_{2^8}$ and there will be a linear recursion relation for $W(t)$ over $F_{2^8}$. The complexity of finding a weight 3 or weight 4 multiple with general methods is far more than that of exhaustive key search. Instead, a fast correlation attack looks more promising. In such an attack the LFSR state at some initial time is considered as a length 64 byte vector $v = (v_0, v_1, \ldots, v_{63}), v_i \in F_{2^8}$ and every $W(t)$ is written as a linear combination of initial state bytes, i.e., $W(t) = w_1 \cdot v$ where the vector $w_1$ is computed through the recursion for $W(t)$. If we now look for pairs of vectors $w_\text{t}$ and $w_\text{t'}$ such that $w_\text{t} \oplus w_\text{t'}$ is zero in the last $d$ entries, we can form the sum

$$
[z^{(t-1)} \oplus z^{(t)} \oplus L^{-1}z^{(t+1)}]_0 \oplus [z^{(t'-1)} \oplus z^{(t')} \oplus L^{-1}z^{(t'+1)}]_0
$$
which approximates $W(t) \oplus W(t')$ with a noise which is the sum of two noise variables of the form $N_{tot}$, which has a bias of $2^{-118.4}$. In the attack we guess $64 - d$ byte entries of the initial state and then we need to find in the order of $2^{118}$ pairs of vectors that have the same last $d$ entries. Through a birthday argument, assuming we generate all such values up to $t_0$, we need to have $t_0^2 \approx 2^{118+16d}$. For example, if $d = 36$ then we need to guess $28 \cdot 8 = 224$ bits and we need to generate output until time $t_0 = 2^{203}$. Following the complexity estimations of e.g. [ZXM15] we end up with a complexity slightly below exhaustive key search. This however requires a keystream of length $t_0 = 2^{203}$, which can be compared to the maximum keystream length allowed which is $2^{64}$. The attack also uses memory of size $2^{203}$ units, each storing the necessary information of one time unit. So the relevance of such attacks is indeed questionable.

### 3.3.4.4 Analysis of the bias using $\sigma$ as proposed

In this scenario, we use $\sigma$ as given in Section 2. This presumably makes the bias of linear approximations for the FSM to be much smaller, and we will sketch some ideas on how to find good linear approximations. Let us again return to the equations for a three word approximation, and use 8-bit adders $\mathbb{H}_8$ in order to simplify derivations:

$$z^{(t-1)} = (S^{-1}(L^{-1} \cdot \hat{R}2) \mathbb{H}_8 T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R}3),$$

$$z^{(t)} = (\hat{R}1 \mathbb{H}_8 T1^{(t)}) \oplus \hat{R}2,$$

$$z^{(t+1)} = (\sigma(\hat{R}2 \mathbb{H}_8 (\hat{R}3 \oplus T2^{(t)}))) \mathbb{H}_8 T1^{(t+1)} \oplus L \cdot S(\hat{R}1).$$

We consider a byte-oriented linear approximation with left hand side $\Lambda_0 z^{(t-1)} \oplus \Lambda_1 z^{(t)} \oplus \Lambda_2 z^{(t+1)}$, where $\Lambda_i$ are length 16 row vectors of bytes viewed as elements in $\mathbb{F}_2^8$.

Recall now that if a particular byte is only present once as a linear term or in an S-box expression in the right hand side, then the approximation will be unbiased. However, if it appears at least twice in different types of expressions, it is likely to give a biased contribution.

The first observation we can do is that the bytes in $\hat{R}1$ appear only in two different expressions: as the direct value $\hat{R}1$ in the expression for $z^{(t)}$ and as $L \cdot S(\hat{R}1)$ in the expression for $z^{(t+1)}$. In turn, this means that $\Lambda_1 z^{(t)} \oplus \Lambda_2 z^{(t+1)}$ depends on $\hat{R}1$ through $\Lambda_1 \hat{R}1 \oplus \Lambda_2 L \cdot S(\hat{R}1)$. It is biased only if every byte present in $\Lambda_1 \hat{R}1$ is also present in $\Lambda_2 L \cdot \hat{R}1$ and we come to the conclusion that $\Lambda_2 = \Lambda_1 L^{-1}$, in it’s straightforward case.

Now we consider the contributions from $\hat{R}2$ and $\hat{R}3$. For $\hat{R}2$ we have contributions $\Lambda_0 S^{-1}(L^{-1} \cdot \hat{R}2), \Lambda_1 \hat{R}2$ and $\Lambda_1 L^{-1} \sigma(\hat{R}2 \mathbb{H}_8 ... )$. When $\sigma_0$ was used, we could simply consider byte 0 ($\Lambda_0 = \Lambda_1 = (1, 0, \ldots, 0)$) because it involved four bytes (byte 0-3) and they all appeared at least twice in the above expression. But for the actual $\sigma$, this is no longer possible since $L^{-1} \sigma(\hat{R}2)$ includes different bytes in a position, compared to $L^{-1} \cdot \hat{R}2$.

We have not been able to find better approximations than the ones that include all 16 bytes of $\hat{R}2$ and $\hat{R}3$, and 4 bytes of $\hat{R}1$ in the approximation. One such example would be $\Lambda_0 = \Lambda_1 = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, which would nearly correspond to the sum of 4 approximations of the previous kind with $\sigma_0$ (no-permutation).

In our best attempt to approximate the FSM of the proposed algorithm, we have got the total noise (with $\mathbb{H}_8$) having the bias (more details can be found in Appendix G):

$$\epsilon(N_{tot}) \approx 2^{-214.8} \quad \text{and} \quad \epsilon(2 \times N_{tot}) \approx 2^{-429.7}.$$
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 $7 \times 128$ bit internal state. That would involve nonlinearity from 11 AES encryption round functions and 13 $⊞_{32}$ 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 give 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 $⊞_{32}$ 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 $z(t) = (R(t \underset{⊞_{32}}{\oplus} T1(t)) \oplus R2(t))$ 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 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 use keystream output $z(1)$ as the final masking for the tag, similarly to the encrypted value of $J_0$ in [Dwo07]. To encrypt the $n$ plaintext blocks, we use the keystream outputs $z(2), \ldots, z(n+1)$, feeding the ciphertext blocks into $GHASH_H$.

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:

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

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 6. 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 tag mask needed in the GCM algorithm.

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 16/32/64-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(). The 16-byte permutation σ can be done with _mm_shuffle_epi8().

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:

\[
\begin{align*}
lo[127..0 \text{ bits}] &= \{a_7,\ldots,a_0\} \\
hi[127..0 \text{ bits}] &= \{a_{15},\ldots,a_8\} \\
lo[255..128 \text{ bits}] &= \{b_7,\ldots,b_0\} \\
hi[255..128 \text{ bits}] &= \{b_{15},\ldots,b_8\}
\end{align*}
\]

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 \(gA(\alpha)\) of the field \(F_{2^{16}}\), without the term \(\alpha^{16}\). Then, multiplication of \(x\) by \(\alpha\) in \(F_{2^{16}}\) 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

\[
\text{mul}_\alpha(\text{uint16 } x, \text{uint16 } gA) := (x<<1) \ xor \ ((\text{signed int16})x >> 15) \ and \ gA)
\]

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 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 \(\alpha\) from the first base field \(F_{2^{16}}\) and the high part with \(\beta\) from the second base field \(F_{2^{16}}\), 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 \(\alpha^{-1}\) and \(\beta^{-1}\). 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 Software performance evaluation

In this section we give software performance benchmarks of SNOW-V-(GCM), implemented by us in C++ (Visual Studio 2017) utilizing AVX2/AES-NI/PCLMULQDQ intrinsics.

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.
All performance tests were carried out on a user-grade laptop with Intel i7-8650U CPU @1.90GHz with Turbo Boost up to @4.20GHz, testing each algorithm on a single process/thread and with various lengths of the input plaintext. Before each encryption process, we perform a key/IV setup procedure. In the first place, we should compare SNOW-V with AES since it demonstrates the fastest speed on commodity CPUs with available AES-NI instructions set. Perhaps, the second best choice algorithm, that is in various places serves as a backup for AES, is ChaCha20, and we should compare with that as well. We also include our best possible implementation of SNOW-3G in order to demonstrate the advantage of the new member of the SNOW family of stream ciphers.

For a fair and most challenging comparison we have downloaded the latest OpenSSL (3.0.0-dev, 2019-04-01) sources, and built it with the latest NASM and Visual Studio 2017 for that certain x64 machine, with most possible optimizations switched on.

Implementations of the algorithms AES-256-(CBC, CTR, GCM) and ChaCha20-(Poly1305) in OpenSSL are the most recent and highly optimized assembly codes that utilize AVX2/AES-NI/PCLMULQDQ, instructions stitching, and other best practice optimization techniques. OpenSSL’s command line tool was used for performance evaluation of the selected algorithms; it runs an algorithm for a chosen number of seconds and delivers the number of bytes per second processed, out of which we derive the speed in Gigabits/sec (Gbps).

In order to fully align our own measurements of SNOW-V with the numbers given by OpenSSL, we actually extracted and adopted the benchmarking code from OpenSSL’s sources and did exactly the same way of measurements of SNOW-V. To negotiate pitfalls from the system and the OS, we benchmarked every considered algorithm several times, for 1-3 seconds, then picked the best values (a similar method is used in SUPERCOP benchmarking approach). The results are presented in Table 3.

Table 3: Performance comparison of SNOW-V-(GCM) and best OpenSSL’s algorithms. Performance values are given in Gbps.

<table>
<thead>
<tr>
<th>Encryption only</th>
<th>Size of input plaintext (bytes)</th>
<th>16384</th>
<th>8192</th>
<th>4096</th>
<th>2048</th>
<th>1024</th>
<th>256</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNOW-3G-128 (C++)</td>
<td>9.22</td>
<td>9.07</td>
<td>8.89</td>
<td>8.50</td>
<td>7.81</td>
<td>5.38</td>
<td>2.37</td>
<td></td>
</tr>
<tr>
<td>AES-256-CBC (asm)</td>
<td>8.50</td>
<td>8.50</td>
<td>8.49</td>
<td>8.48</td>
<td>8.42</td>
<td>8.11</td>
<td>7.07</td>
<td></td>
</tr>
<tr>
<td>AES-256-CTR (asm)</td>
<td>35.06</td>
<td>34.82</td>
<td>34.16</td>
<td>32.94</td>
<td>30.95</td>
<td>22.67</td>
<td>11.32</td>
<td></td>
</tr>
<tr>
<td><strong>SNOW-V (C++)</strong></td>
<td><strong>58.25</strong></td>
<td><strong>56.98</strong></td>
<td><strong>54.60</strong></td>
<td><strong>50.70</strong></td>
<td><strong>45.28</strong></td>
<td><strong>26.37</strong></td>
<td><strong>9.85</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AEAD mode</th>
<th>Size of input plaintext (bytes)</th>
<th>16384</th>
<th>8192</th>
<th>4096</th>
<th>2048</th>
<th>1024</th>
<th>256</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20-Poly1305 (asm)</td>
<td>18.46</td>
<td>18.24</td>
<td>18.16</td>
<td>17.54</td>
<td>16.99</td>
<td>8.98</td>
<td>4.29</td>
<td></td>
</tr>
<tr>
<td>AES-256-GCM (asm)</td>
<td>34.42</td>
<td>33.86</td>
<td>32.74</td>
<td>30.49</td>
<td>27.22</td>
<td>17.32</td>
<td>8.54</td>
<td></td>
</tr>
<tr>
<td><strong>SNOW-V-GCM (C++)</strong></td>
<td><strong>38.91</strong></td>
<td><strong>37.66</strong></td>
<td><strong>34.86</strong></td>
<td><strong>30.71</strong></td>
<td><strong>26.16</strong></td>
<td><strong>13.93</strong></td>
<td><strong>5.16</strong></td>
<td></td>
</tr>
</tbody>
</table>

For a large plaintext, SNOW-V outperforms AES-256-CBC by around 6.5 times, even though AES-256-CBC is implemented in an optimized assembly code with AES-NI. SNOW-V is also 2 times faster than ChaCha20. An encryption in AES-256-CTR can be done in “parallel”, so that the technique called "instructions interleaving" makes it possible to speed up a lot. Even here, SNOW-V is 66% faster than AES-256-CTR.

We would like to note that running an algorithm for even 1 second includes a lot of system overhead such as OS’s scheduler, switching to other hundreds of OS’s processes and services, switching CPUs’ contexts and affinity for load balancing, etc. When we tried to measure SNOW-V for a very small fraction of a second (basically, measuring a single encryption), we have seen the speed goes up to 67Gbps (for encryption of 16384 bytes), which indicates that the OS’s overhead is a significant factor.

One could notice from our measurements that AEAD mode of OpenSSL’s AES-256-
A new SNOW 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. In a single thread implementation in software, SNOW-V outperforms AES in all comparable modes of operation for plaintext lengths above 256 bytes. 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 autorefannex:test-vectors, Appendix C, and Appendix D. We also provide a brief hardware evaluation of SNOW-V in Appendix E, including a 64-bit implementation utilizing a single AES core.

Tests were run on an Apple iPhone X.

Table 4: Performance comparison of SNOW-V and AES-CTR on an Apple A11 ARM processor. Performance values are given in Gbps.

<table>
<thead>
<tr>
<th>Size of input plaintext (bytes)</th>
<th>16384</th>
<th>8192</th>
<th>4096</th>
<th>2048</th>
<th>1024</th>
<th>256</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNOW-V (C)</td>
<td>23.59</td>
<td>23.24</td>
<td>22.38</td>
<td>21.31</td>
<td>19.39</td>
<td>12.31</td>
<td>5.0</td>
</tr>
<tr>
<td>AES-CTR (asm)</td>
<td>15.97</td>
<td>15.87</td>
<td>15.59</td>
<td>15.08</td>
<td>14.34</td>
<td>10.62</td>
<td>5.04</td>
</tr>
</tbody>
</table>
References


A new SNOW stream cipher called SNOW-V


A new SNOW stream cipher called SNOW-V


\section{Remarks about the maximum period of the LFSR structure}

We can denote the LFSRs’ state at time \( t \geq 0 \) as

\[ S^{(t)} = (a_0^{(t)}, a_1^{(t)}, \ldots, a_{14}^{(t)}, a_{15}^{(t)}, b_0^{(t)}, b_1^{(t)}, \ldots, b_{14}^{(t)}, b_{15}^{(t)}) \]

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 \times 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 \( i = 0, 1, \ldots, 14 \), and the corresponding binary state transition submatrix for such update is identity matrix \( I_8 \) with size 16 \times 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 \( (1, \alpha, \ldots, \alpha^{15}), (1, \beta, \ldots, \beta^{15}) \), 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 \( e = e_0 + e_1\alpha + \ldots, + e_{14}\alpha^{14} + e_{15}\alpha^{15} \), where \( e_i \) denotes the \( i \)-th bit of \( e \). Then,

\[ e\alpha \mod g^A(a) = (e_{15}\alpha^{16} + e_{14}\alpha^{15} + \ldots, + e_1\alpha^2 + e_0\alpha) \mod g^A(a) \]

Since

\[ \alpha^{16} \mod g^A(a) = \alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1, \]

\( e\alpha \mod g^A(a) \) can be expanded and rearranged as,

\[ = e_{15}(\alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1) + e_{14}\alpha^{15} + \ldots, + e_1\alpha^2 + e_0\alpha \]

\[ = (e_{15} + e_{14}\alpha^{15} + e_{13}\alpha^{14} + e_{12}\alpha^{13} + (e_{15} + e_{11})\alpha^{12} + (e_{15} + e_{10})\alpha^{11} + e_{9}\alpha^{10} + e_8\alpha^9 + (e_{15} + e_{7})\alpha^8 + e_6\alpha^7 + e_5\alpha^6 + e_4\alpha^5 + e_3\alpha^4 + (e_{15} + e_2)\alpha^3 + (e_{15} + e_1)\alpha^2 + (e_{15} + e_0)\alpha + e_{15} \]

\[ = (e_0, e_1, \ldots, e_{15})M_\alpha(1, \alpha, \ldots, \alpha^{15})^T \]

From which we can deduce the matrix

\[ M_\alpha = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 
\end{bmatrix} \]
With the same method, we can also derive $M_{\alpha^{-1}}, M_{\beta}, M_{\beta^{-1}}$. Then we can rewrite the update for $a_{15}^{(t+1)}, \beta_{15}^{(t+1)}$ in a matrix form,
\[
\begin{align*}
\alpha_{15}^{(t+1)} &= b_{0}^{(t)} M_{I} + a_{15}^{(t)} M_{\alpha} + a_{2}^{(t)} M_{I} + a_{0}^{(t)} M_{\alpha^{-1}} \\
\beta_{15}^{(t+1)} &= a_{1}^{(t)} M_{I} + b_{1}^{(t)} M_{\beta} + b_{3}^{(t)} M_{I} + b_{5}^{(t)} M_{\beta^{-1}}
\end{align*}
\]

Then the elaborate binary transition matrix for the LFSR structure can be written as,
\[
\begin{array}{cccccccccccc}
0 & 1 & \ldots & 7 & \ldots & 14 & 15 & \ldots & 18 & \ldots & 23 & \ldots & 30 & 31 \\
& & & & & & & & & & & & \\
0 & & & & & \alpha & & & & & & \beta & & \\
1 & & & & & M_{I} & & & & & & M_{I} & \\
2 & & & & & M_{I} & & & & & & M_{I} & \\
\vdots & & & & & \vdots & & & & & & \vdots & \\
8 & & & & & M_{I} & & & & & & M_{\alpha^{-1}} & \\
\vdots & & & & & \vdots & & & & & & \vdots & \\
M &= 15 & \ldots & 16 & \ldots & 19 & \ldots & 24 & \ldots & 31 \\
& & & & & & & & & & & & \\
& & & & & M_{I} & & & & & & M_{I} & \\
& & & & & M_{I} & & & & & & M_{\beta} & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots & \\
& & & & & \vdots & & & & & & \vdots &
\end{array}
\]

where every element in the $32 \times 32$ matrix is a $16 \times 16$ matrix and all the other blank places are $16 \times 16$ zero matrices. Then we can get the $512 \times 512$ transition matrix and some mathematical tools such as Sagemath, 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,
\[
m(x) =
\begin{align*}
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^{333} + 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} + \ldots + x^{202} + \\
x^{299} + x^{296} + x^{292} + x^{291} + x^{284} + x^{283} + x^{281} + x^{280} + x^{275} + x^{274} + x^{273} + \\
x^{271} + x^{267} + x^{264} + x^{263} + x^{262} + x^{260} + x^{259} + x^{258} + x^{257} + x^{256} + x^{255} + x^{253} + x^{251} + x^{249} + x^{248} + x^{247} + x^{246} + x^{245} + x^{244} + x^{243} + x^{242} + x^{240} + \\
x^{238} + x^{236} + x^{229} + x^{225} + x^{217} + x^{215} + x^{214} + x^{213} + x^{212} + x^{209} + x^{208} + x^{207} + x^{205} + x^{204} + x^{203} + x^{199} + x^{198} + x^{193} + x^{192} + x^{191} + x^{189} + x^{188} + 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^{69} + x^{68} + x^{67} + x^{66} + x^{64} + x^{63} + x^{62} + x^{61} + x^{59} + x^{56} + x^{54} + 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{align*}
\]

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.

== SNOW-V test vectors #1:
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 phase, z =

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 ea ea eb eb eb eb eb eb eb eb eb eb eb eb eb eb
55 f7 f7 c2 e8 e8 dd 4a e8 dd 4a e8 e8 dd 4a e8 dd 4a
6a 67 62 3e 6f 8a f9 79 ef cd 81 83 c5 86 8e 3a
45 10 1e 83 a2 c6 dd eb 40 86 3a 2d ac f9 3b 65
3c c4 df 66 cf b1 c1 06 d6 ac 02 c5 0a 68 3c fe
c7 a7 dd ca f3 13 87 61 02 6e ad f4 2b 54 e3 ef
cf a7 dd ca f3 13 87 61 02 6e ad f4 2b 54 e3 ef
cf 67 63 07 dd 9d 24 2d 31 7f 62 7c 64 3a 05 ff
dc 77 a4 97 23 eb 65 6a a1 bf 8f 28 ec f1 de 1d 00
Keystream phase, z =

69 ca 6d af 9a e3 b7 2d b1 34 a8 5a 83 7e 41 9d
eb 7a 7d 6b 2e 9c 0b 9b be 2b 6a e8 6a e8 6a e8
eb 65 f6 62 f6 65 f6 62 f6 65 f6 62 f6 65 f6 62 f6
6f 0a 3c e1 7d 0c 54 49 1e 8f 9c 06 0e 5f 2d 03
cb ca 60 82 14 a2 65 82 cf 68 09 16 b3 45 13 21
95 4d af 30 84 af 02 f6 a8 e2 48 1d e6 bf 83 79

== SNOW-V test vectors #2:
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 phase, z =

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
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
d3 07 d2 07 d3 07 d7 07 d3 07 d7 07 f8 02 f8 02 f8
6f 62 6f 65 6f 62 6f 65 6f 62 6f 65 6f 62 6f 65
fe be fe be be be be be be be be be be be be be
03 1c 15 9d cc dd a5 0c 4d 5d fd 51 15 d8 70 30
c0 d0 3c a1 37 0c 19 40 03 47 a0 b4 d2 e9 db e5
cb ca 6c 82 14 a2 65 82 cf 68 09 16 b3 45 13 21
9c 4f af 30 84 af 02 f6 a8 e2 48 1d e6 bf 83 79
Keystream phase, z =

30 76 09 fb 10 10 12 54 4b c1 75 e3 17 fb 25 ff
33 04 0d e2 5a f6 aa d1 05 05 b8 9b 1e 09 a8 ec
dd 46 72 cc bb 98 c7 f2 c4 e2 4a f5 27 28 36 c8
7c c7 3a 81 76 b3 9c e9 30 3b 7e 7e 4e 9b e3 e7
48 f7 65 1a 7c 7e 81 3f df 24 90 23 1e 56 f7 c1
A new SNOW stream cipher called SNOW-V

== SNOW-V test vectors #3:
key = 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
iv = 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initialization phase, z =
0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
66 d4 2d 92 ac b2 b6 44 63 3c c3 71 c3 91 c6 24
a2 d7 ea be 3f 04 8e 50 00 b1 7b 74 2f 34 5e 49
96 a7 34 ed fd 07 66 9d c8 f9 a2 91 fc 13 76 73
58 c8 70 73 d8 a2 a1 bd 03 e7 a1 4c c7 b7 db 89
7e 86 eb 71 d6 dc 00 99 d1 31 e3 1b 54 c5 3e f8
a8 cb ff 06 0d c0 9e 67 cc 95 62 16 17 19 8c f2
c0 99 3a 55 f3 e2 d7 8d 6a f7 e1 57 0f a1 63 02
39 8f a0 7e ab a2 73 89 94 f9 ac 3e 8e b1 ff 64
f3 32 31 6a 42 5c 12 a6 39 ce 79 cb 3b 40 37 1e
2e 7a 44 fd ad 23 77 5a f1 7c 6e 7f 9a be 3f 04
93 69 c8 20 a9 37 d5 c8 b6 7a df 84 45 93 1c 09
c1 0f 8d b5 fb 37 08 31 11 d1 c8 44 6e a2 ac 9e
13 ac 34 20 7b 01 b7 ab d3 57 02 a1 ed 98 bb dc
0b 15 43 a4 74 26 2c 76 e3 a2 73 57 28 4b dc 67
7b 79 91 96 cf 6b 76 27 18 dd a1 89 bb af dc 93
Keystream phase, z =
aa 81 ea fb 8b 86 16 ce 3e 5c e2 22 24 61 c5 0a
6a b4 48 77 56 de 4b d3 1c 90 4f 3d 97 8a fe 56
33 4f 10 dd df 2b 95 31 76 9a 71 05 0b e4 38 5f
c2 b6 19 2c 7a 85 7b e8 b4 fc 28 b7 09 f0 8f 11
f2 06 49 e2 ee f2 49 80 f8 6c 4c 11 36 41 fe d2
f3 f6 fa 2b 91 95 12 06 b8 01 db 15 46 65 17 a6
33 0a dd a5 b3 5b 26 5e fd 72 2e 86 77 b4 8b fc
15 b4 41 18 de 52 d0 73 b0 ad 0f ef 59 4d 62 91

Listing 1: Test vectors for SNOW-V.

== SNOW-V-GCM test vectors #1:
key = 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
aad =
plain =
key H = e9 c0 d9 30 07 99 d4 f6 70 23 08 78 cd 49 65 d5
depend = 02 9a 62 4c da a4 d4 6c b9 a0 ef 40 46 95 6c 9f
cipher = author = 02 9a 62 4c da a4 d4 6c b9 a0 ef 40 46 95 6c 9f

== SNOW-V-GCM test vectors #2:
key = 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
iv = 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
aad =
plain =
key H = a5 78 c7 e6 c9 dd e7 7f af b7 ae 37 fa 56 95 4a
depend = fc 7c ac 57 4c 49 fe ae 61 50 31 5b 96 85 42 4c
cipher = author = fc 7c ac 57 4c 49 fe ae 61 50 31 5b 96 85 42 4c

== SNOW-V-GCM test vectors #3:
key = 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
aad = 30 31 32 33 34 35 36 37 38 39 61 62 63 64 65 66
plain =
key $H = \text{e9 c0 d9 30 07 99 d4 f6 70 23 08 78 cd 49 65 d5}$
endpad = 02 9a 62 4c da a4 d4 6c b9 a0 ef 40 46 95 6c 9f
cipher =
auth = 5a 5a a5 fb d6 35 ef 1a e1 29 61 42 03 e1 03 84

== SNOW-V-GCM test vectors #4:

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
aad = 30 31 32 33 34 35 36 37 38 39 61 50 31 5b 96 85 42 4c
plain =
key $H = \text{a5 78 c7 e6 c9 dd e7 7f af b7 ae 37 fa 56 95 4a}$
endpad = fc 7c ac 57 4c 49 fe ae 61 50 31 5b 96 85 42 4c
cipher =
auth = 25 0e c8 d7 7a 02 2c 08 7a df 08 b6 5a dc bb 1a

== SNOW-V-GCM test vectors #5:

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
aad =
plain = 30 31 32 33 34 35 36 37 38 39
key $H = \text{58 c7 e6 c9 dd e7 7f af b7 ae 37 fa 56 95 4a}$
endpad = fc 7c ac 57 4c 49 fe ae 61 50 31 5b 96 85 42 4c
cipher =
auth = dd 7e 01 b2 b4 24 a2 ef 82 50

== SNOW-V-GCM test vectors #6:

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
aad = 41 41 44 20 74 65 73 74 70 76 61 6c 75 65 21
plain = 30 31 32 33 34 35 36 37 38 39 61 50 31 5b 96 85 42 4c
key $H = \text{a5 78 c7 e6 c9 dd e7 7f af b7 ae 37 fa 56 95 4a}$
endpad = fc 7c ac 57 4c 49 fe ae 61 50 31 5b 96 85 42 4c
cipher =
auth = 90 7e a6 a5 b7 3a 51 de 74 7c 3e 9a d9 ee 02 9b

Listing 2: Test vectors for SNOW-V-GCM.
C  SNOW-V 32-bit Reference Implementation in C/C++

#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, 0x02, 0xF6, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x18, 0x1F, 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8A, 0x8B, 0x8C, 0x8D, 0xB2, 0x8E, 0x8F,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9D, 0xA1, 0xA5, 0xA6, 0xA7, 0xA8, 0xA9, 0xAA, 0xAB, 0xAC, 0xB0, 0xB1, 0xB2, 0xB3, 0xB4, 0xB5, 0xB6, 0xB7, 0xB8, 0xB9, 0xBA, 0xBB, 0xBC, 0xBD, 0xBE, 0xBF,
    0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7, 0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF, 0xD0, 0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7, 0xD8, 0xD9, 0xDA, 0xDB, 0xDC, 0xDD, 0xDE, 0xDF,
    0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7, 0xE8, 0xE9, 0xEA, 0xEB, 0xEC, 0xED, 0xEE, 0xEF, 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
};

u8 Sigma[16] = { 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15};

u32 AesKey1[4] = { 0, 0, 0, 0 };  // round key 1
u32 AesKey2[4] = { 0, 0, 0, 0 };  // round key 2

#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) & 0xfee0000UL) ^ (((w >> 7) & 0x01000000UL) * 0x1b);
            result[j] = roundKey[j] ^ w ^ t ^ ROTL32(t, 8)
        #endif
        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(0);
}
return (v << 1);
}

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

void permute_sigma(u32 * state)
{
  u8 tmp[16];
  for (int i = 0; i < 16; i++)
    tmp[i] = (u8)(state[i >> 2] >> ((i & 3) << 3));

  for (int i = 0; i < 4; i++)
    state[i] = MAKEU32(MAKEU16(tmp[4 * i + 3], tmp[4 * i + 2]),
                      MAKEU16(tmp[4 * i + 1], tmp[4 * i]));
}

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];
  }
  permute_sigma(R1);
  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];
    }
  }
}

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, int is_aead_mode)
{ 
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]);
}

if(is_aead_mode == 1)
{ 
  B[0] = 0x6C41;
  B[1] = 0x7865;
  B[2] = 0x6B45;
  B[3] = 0x2064;
  B[4] = 0x694A;
  B[5] = 0x676E;
  B[6] = 0x6854;
  B[7] = 0x6D6F;
}

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]));
    }
  }
};
// AEAD mode: SNOW-V-GCM in C++ (Endianness-free)
#include <stdint.h>
#include <stdlib.h>
#include "SNOWV.h"
#include "ghash.h"

#define min(a, b) (((a) < (b)) ? (a) : (b))

void snowv_gcm_encrypt(u8 * A, u8 * ciphertext, u8 * plaintext, u64 plaintext_sz,
                    u8 * aad, u64 aad_sz, u8 * key32, u8 * iv16)
{
    u8 Hkey[16], endPad[16];
    struct SnowV32 snowv;
    memset(A, 0, 16);
    snowv.keyiv_setup(key32, iv16, 1);
    snowv.keystream(Hkey);
    snowv.keystream(endPad);
    ghash_update(Hkey, A, aad, aad_sz);

    for (u64 i = 0; i < plaintext_sz; i += 16)
    {
        u8 key_stream[16];
        snowv.keystream(key_stream);
        for(u8 j = 0; j < min(16, plaintext_sz - i); j++)
            ciphertext[i + j] = key_stream[j] ^ plaintext[i + j];
    }
    ghash_update(Hkey, A, ciphertext, plaintext_sz);
    ghash_final(Hkey, A, aad_sz, plaintext_sz, endPad);
}

void snowv_gcm_decrypt(u8 * A, u8 * ciphertext, u8 * plaintext, u64 ciphertext_sz,
                    u8 * aad, u64 aad_sz, u8 * key32, u8 * iv16)
{
    u8 Hkey[16], endPad[16], auth[16] = {0x00};
    struct SnowV32 snowv;
    snowv.keyiv_setup(key32, iv16, 1);
    snowv.keystream(Hkey);
    snowv.keystream(endPad);
    ghash_update(Hkey, auth, aad, aad_sz);
    ghash_update(Hkey, auth, ciphertext, ciphertext_sz);
    ghash_final(Hkey, auth, aad_sz, ciphertext_sz, endPad);
    for(int i = 0; i < 16; i++)
    {
        if(auth[i] != A[i])
        {
            printf("Authentication Failed!\n");
            exit(1);
        }
    }

    for (u64 i = 0; i < ciphertext_sz; i += 16)
    {
        u8 key_stream[16];
        snowv.keystream(key_stream);
        for(u8 j = 0; j < min(16, ciphertext_sz - i); j++)
            plaintext[i + j] = key_stream[j] ^ ciphertext[i + j];
    }
}
// Informative: an exampled implementation of GHASH core (C++)
#define XOR2x64(dst, src)   ((u64*)(dst))[0] ^= ((u64*)(src))[0],
                         ((u64*)(dst))[1] ^= ((u64*)(src))[1]
#define XOR3x64(dst, src1, src2) ((u64*)(dst))[0] = ((u64*)(src1))[0] ^ ((u64*)(src2))[0],
                         ((u64*)(dst))[1] = ((u64*)(src1))[1] ^ ((u64*)(src2))[1]

void ghash_mult(u8 * out, const u8 * x, const u8 * y)
{
  char tmp[17];
  u64 c0, c1, u0 = ((u64*)y)[0], u1 = ((u64*)y)[1];
  memset(out, 0, 16);
  for (int i = 0; i < 16; i++)
    for (int j = 7; j >= 0; j--)
      if ((x[i] >> j) & 1) ((u64*)out)[0] ^= u0, ((u64*)out)[1] ^= u1;
    c0 = (u0 << 7) & 0x8080808080808080ULL;
    c1 = (u1 << 7) & 0x8080808080808080ULL;
    u0 = (u0 >> 1) & 0x7f7f7f7f7f7f7f7fULL;
    u1 = (u1 >> 1) & 0x7f7f7f7f7f7f7f7fULL;
    ((u64*)(tmp + 1))[0] = c0;
    ((u64*)(tmp + 1))[1] = c1;
    tmp[0] = (tmp[16] >> 7) & 0xe1;
    u0 ^= ((u64*)tmp)[0];
    u1 ^= ((u64*)tmp)[1];
}

void ghash_update(const u8 * H, u8 * A, const u8 * data, long long length)
{
  u8 tmp[16];
  for(; length >= 16; length -= 16, data += 16)
    XOR3x64(tmp, data, A);
    ghash_mult(A, tmp, H);
  if(!length) return;
  memset(tmp, 0, 16);
  memcpy(tmp, data, length);
  XOR2x64(tmp, A);
  ghash_mult(A, tmp, H);
}

void ghash_final(const u8 * H, u8 * A, u64 lenAAD, u64 lenC, const u8 * maskingBlock)
{
  u8 tmp[16];
  lenAAD <<= 3;
  lenC <<= 3;
  for(int i=0; i<lenA; i++)
    { tmp[7-i] = (u8)(lenAAD >> (8 * i));
      tmp[15-i] = (u8)(lenC >> (8 * i));
    } XOR2x64(tmp, A);
    ghash_mult(A, tmp, H);
    XOR2x64(A, maskingBlock); /* The resulting AuthTag is in A[] */
// SNOW-V Reference Implementation with AVX2 (Little endian)
#define vpset16(value) _mm256_set1_epi16(value)
const __m256i _snowv_mul = _mm256_blend_epi32(vpset16( 0x990f), vpset16( 0xc963), 0xf0);
const __m256i _snowv_inv = _mm256_blend_epi32(vpset16(-0xcc87), vpset16(-0xe4b1), 0xf0);
const __m128i _snowv_aead = _mm_loadu_si128((__m128i*)"AlexEkd JingThom");
const __m128i _snowv_sigma= _mm_set_epi8(15,11,7,3,14,10,6,2,13,9,5,1,12,8,4,0);
const __m128i _snowv_zero = _mm_setzero_si128();

struct SnowV256
{
__m256i hi, lo; // LFSR
__m128i R1, R2, R3; // FSM

inline __m128i keystream(void)
{
    // Extract the tags T1 and T2
    __m128i T1 = _mm256_extracti128_si256(hi, 1);
    __m128i T2 = _mm256_castsi256_si128(lo);

    // LFSR Update
    __m256i mulx = _mm256_xor_si256(_mm256_slli_epi16(lo, 1),
    _mm256_and_si256(_snowv_mul, _mm256_srai_epi16(lo, 15)));
    __m256i invx = _mm256_xor_si256(_mm256_srli_epi16(hi, 1),
    _mm256_sign_epi16(_snowv_inv, _mm256_slli_epi16(hi, 15)));
    __m256i hi_old = hi;
    hi = _mm256_xor_si128(R2, _mm_add_epi32(R1, T1));
    __m256i lo = hi_old;

    // Keystream word
    __m128i z = _mm_xor_si128(R2, _mm_add_epi32(R1, T1));

    // FSM Update
    __m128i R3new = _mm_aesenc_si128(R2, _snowv_zero);
    __m128i R2new = _mm_aesenc_si128(R1, _snowv_zero);
    R1 = _mm_shuffle_epi8(_mm_add_epi32(R2, _mm_xor_si128(R3, T2)), _snowv_sigma);
    R3 = R3new;
    R2 = R2new;
    return z;
}

template<int aead_mode = 0>
inline void keyiv_setup(const unsigned char * key, const unsigned char * iv)
{
    R1 = R2 = R3 = _mm_setzero_si128();
    hi = _mm256_loadu_si256((const __m256i*)key);
    lo = _mm256_zextsi128_si256(_mm_loadu_si128((__m128i*)iv));
    if (aead_mode)
    lo = _mm_insertf128_si256(lo, _snowv_aead, 1);
    for (int i = 0; i < 15; ++i)
    hi = _mm256_xor_si256(hi, _mm256_zextsi128_si256( keystream() ));
    R1 = _mm_xor_si128(R1, _mm_loadu_si128((__m128i*)key + 0));
    hi = _mm256_xor_si256(hi, _mm_zextsi128_si256( keystream() ));
    R1 = _mm_xor_si128(R1, _mm_loadu_si128((__m128i*)key + 16));
}
A new SNOW stream cipher called SNOW-V

#define SNOWV_ENCDEC(snowv, out, in) _mm_storeu_si128((__m128i*)(out),
                _mm_xor_si128(_mm_lddqu_si128((__m128i*)(in)), snowv.keystream()))

// Any external implementation of GHASH
struct ghash_context;
extern void ghash_init (ghash_context & gh, __m128i H);
extern void ghash_update(ghash_context & gh, const u8 * data, long long length);
extern __m128i ghash_final (ghash_context & gh, u64 lenAAD, u64 lenC, __m128i endPad);

// Note: ciphertext must reserve plaintext_sz + 16 bytes
long long snowv_gcm_encrypt(u8 * ciphertext, const u8 * plaintext, u64 plaintext_sz,
                              const u8 * aad, u64 aad_sz, const u8 * key32, const u8 * iv16)
{
    ghash_context gh;
    SnowV256 snowv;
    snowv.keyiv_setup<1>(key32, iv16); // init with AEAD mode
    ghash_init(gh, snowv.keystream()); // GHASH key H
    __m128i endPad = snowv.keystream(); // ending pad
    ghash_update(gh, aad, aad_sz); // push AAD into GHASH
    for (long long i = 0; i < plaintext_sz; i += 16)
        SNOWV_ENCDEC(snowv, ciphertext + i, plaintext + i);
    ghash_update(gh, ciphertext, plaintext_sz); // Push ciphertext into GHASH
    __m128i auth = ghash_final(gh, aad_sz, plaintext_sz, endPad);
    auth = _mm_xor_si128(auth, _mm_lddqu_si128((__m128i*)(ciphertext + plaintext_sz)));
    if (!_mm_test_all_zeros(auth, auth))
        return -1; // auth tag is not correct? return a negative value

    return plaintext_sz + 16;
}

// Note: plaintext must reserve ciphertext_sz - 16 bytes
long long snowv_gcm_decrypt(u8 * plaintext, const u8 * ciphertext, u64 ciphertext_sz,
                             const u8 * aad, u64 aad_sz, const u8 * key32, const u8 * iv16)
{
    ghash_context gh;
    SnowV256 snowv;
    snowv.keyiv_setup<1>(key32, iv16); // init with AEAD mode
    ghash_init(gh, snowv.keystream()); // GHASH key H
    ghash_update(gh, aad, aad_sz); // push AAD into GHASH
    for (long long i = 0; i < ciphertext_sz; i += 16
        SNOWV_ENCDEC(snowv, ciphertext + i, plaintext + i);
    // Finalize GCM mode and verify the authorization tag
    if (!_mm_test_all_zeros(auth, auth))
        return -1; // auth tag is not correct? return a negative value

    return ciphertext_sz;
}
E 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.

E.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: additional two 64-bit delay registers $D_1$ and $D_2$ are needed; the logic needs additional 6 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 $T_1$ and $T_2$ from the LFSRs also arrive in 64-bit chunks, such that every even clock FSM gets $T_1a$ and $T_2a$, and every odd clock $T_1b$ and $T_2b$.

In Figure 7 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 $R_1$, $R_2$, $R_3$ become updated to new 128-bit values $R_1'$, $R_2'$, $R_3'$, 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 8 the complete hardware architecture for 64-bit SNOW-V is presented. There are 7 64-bit multiplexers in total, and we denote the control signal to them by $M_1..M_7$, 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 AND-blocks. The Control Unit in Figure 8 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 AND-MUX-XOR-ADD-XOR-MUX over 2x32-bit integers, denoted by red wires in Figure 8. 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:
Figure 7: Splitting of FSM into two steps in order to utilize only one AES core (excluding initialization steps).

Figure 8: Hardware architecture of 64-bit SNOW-V with a single AES core.
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 \(M_4 = 1, M_5 = M_7 = 0\). \(G_K\) 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 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 detonates the Z signal from being feeded into LFSR-A by setting \(G_Z = 0\). The input bus is also detached by setting \(G_K = M_6 = 0\).

E.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+2x64 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 1024 * 4.33 = 4434 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 $3x64AND2 + 6x64MUX2 + 4x64XOR2 = 3 * 64 * 1.33 + 6 * 64 * 2.33 + 4 * 64 * 2.33 = 1747$ 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 $g^4(\alpha)$ 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

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

(13)

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 $\alpha$ and $\alpha^{-1}$. Table 5 gives a full overview of the dependencies for both $a_{16}$ and $b_{16}$.

|    | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $a_{16}[i]$ | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  |   |
| $b_{16}[i]$ | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  |   |

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 13, and the XOR gate needs 4 inputs:


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


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

Following the hardware architecture of 64-SNOW-V given in Figure 8 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

\[
r'_{0,j} = 2t_0 + t_2 + r_1,
\]

\[
r'_{1,j} = 2t_1 + t_3 + r_2,
\]

\[
r'_{2,j} = 2t_2 + t_0 + r_3,
\]

\[
r'_{3,j} = 2t_3 + t_1 + r_0,
\]

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 \( 4 \times 4 \times (8 \text{XOR2} + 8 \text{XOR3} + 3 \text{XOR2}) = 922 \text{ GE} \).

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 \( 16 \times 220 + 922 = 4442 \text{ GE} \).

Summarizing the above we can derive the comparison given in Table 6.

<table>
<thead>
<tr>
<th>Hardware design</th>
<th>AES-256 from [UMHA16]</th>
<th>64-SNOW-V 1xAES ext. core</th>
<th>64-SNOW-V 1xAES int. round</th>
<th>128-SNOW-V 2xAES ext. cores</th>
<th>128-SNOW-V 2xAES int. rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>17232 GE</td>
<td>9067 GE</td>
<td>13041 GE</td>
<td>11231 GE</td>
<td>19179 GE</td>
</tr>
<tr>
<td>Speed</td>
<td>50.85 Gbps</td>
<td>358 Gbps</td>
<td>358+ Gbps</td>
<td>712 Gbps</td>
<td>712+ Gbps</td>
</tr>
</tbody>
</table>

Recent results in [KLSW17] suggest that MixColumn can be implemented with 4x97 gates. We found an even better circuit with 4x95 gates, see Appendix Appendix F for details. However, we believe that the proposed classical solution with 4x108 gates is a better choice since it has a lower depth of the critical path, thus allowing SNOW-V to perform a lot faster.
The part MixColumn of AES encryption round can be implemented with as small as 97 XOR gates \[\text{KLSW17}\] that implies the cost 97XOR2=226GE. The circuit needs to be repeated 4 times since MixColumn should be applied to each column of the AES state, resulting in 904GE. However, AES EncRound is on the critical path of SNOW-V and, thus, it is important to have the depth of the circuit as low as possible. The classical solution is a 2-depth circuit with \(4\times108\) gates which results in the area \(922\)GE, just 18GE larger, but having the smallest possible depth. Therefore, we decided to not use the mentioned recent result with \(4\times97\) gates. However, as a bonus to this study, we have improved the result in \[\text{KLSW17}\] with an even smaller circuit with 95 gates.

\(x\) is the 32-bit input value, and \(y\) is the 32-bit output value.

\[
\begin{align*}
t0 &= x23 \times x31 & t19 &= t14 + t15 & t38 &= x24 + t2 & t57 &= x17 + t15 & t76 &= t0 + t75 \\
t1 &= x7 + x15 & y26 &= x18 + t19 & y16 &= t37 + t38 & t58 &= x26 + t57 & y19 &= x11 + t76 \\
t2 &= x7 + x31 & t21 &= t10 + t11 & t60 &= t35 + y16 & y18 &= x25 + t58 & t78 &= t17 + t18 \\
t3 &= x15 + x23 & y21 &= x13 + t21 & y24 &= t1 + t40 & t60 &= x1 + t57 & t79 &= t2 + t78 \\
t4 &= x22 + x30 & t23 &= t12 + t13 & t42 &= x28 + t29 & y2 &= y10 + t60 & y28 &= x20 + t79 \\
t5 &= x16 + x24 & y10 &= x2 + t23 & t43 &= y21 + t42 & t62 &= x0 + t13 & t81 &= x3 + t1 \\
t6 &= x8 + t1 & t25 &= t4 + t8 & y13 &= x4 + t43 & t63 &= t38 + t62 & t82 &= x11 + t81 \\
y0 &= t5 + t6 & y14 &= x6 + t25 & t45 &= x12 + t42 & y26 &= x1 + t63 & t83 &= y28 + t82 \\
t8 &= x13 + x21 & t27 &= t9 + t10 & y29 &= x5 + t45 & t65 &= x25 + t6 & y20 &= t73 + t83 \\
t9 &= x6 + x14 & y30 &= x22 + t27 & t47 &= x30 + t3 & y1 &= t62 + t65 & t85 &= x12 + t82 \\
t10 &= x5 + x29 & t29 &= t8 + t17 & t48 &= y23 + t47 & t67 &= t0 + t14 & y4 &= t11 + t85 \\
t11 &= x20 + x28 & y5 &= x29 + t29 & y15 &= x14 + t48 & t68 &= x9 + t67 & t87 &= x19 + t18 \\
t12 &= x18 + x26 & t31 &= t3 + t9 & t50 &= x7 + t47 & y17 &= t5 + t68 & t88 &= t82 + t87 \\
t13 &= x9 + x17 & y7 &= x31 + t31 & y31 &= x6 + t50 & t70 &= t40 + t62 & y3 &= t15 + t88 \\
t14 &= x1 + x25 & t33 &= t2 + t4 & t52 &= x21 + t27 & y9 &= t68 + t70 & t90 &= x18 + t3 \\
t15 &= x2 + x10 & y23 &= x15 + t33 & t53 &= x29 + t52 & t72 &= t11 + t16 & t91 &= x10 + t90 \\
t16 &= x11 + x19 & t35 &= t3 + t5 & y6 &= y14 + t53 & t73 &= t3 + t72 & y11 &= t87 + t91 \\
t17 &= x4 + x12 & y8 &= x0 + t35 & t55 &= x30 + t53 & y12 &= x4 + t73 & t93 &= y3 + t91 \\
t18 &= x3 + x27 & t37 &= y0 + y8 & y22 &= t10 + t55 & t75 &= t12 + t18 & y27 &= t76 + t93 \\
\end{align*}
\]

Listing 3: MixColumn with 95 gates

\footnote{As it was shown in previous Section, the total number of gates for MixColumn in the classical solution is actually \(4\times4\times(8\text{XOR2} + 8\text{XOR3} + 3\text{XOR2})\), where XOR3 gates are utilized. The GE numbers are: 2.33GE for XOR2, and \(\sim4\)GE for XOR3. Therefore, the total cost of MixColumn in the classical solution is 922GE.}
G Details on the exampled linear approximation of the FSM for the proposed algorithm

In this Section we provide more details on the exampled approximation given in Section 3.4.4. We, again, assume 8-bit adders \( \boxplus_8 \), instead of \( \boxplus_{32} \). Recall the exampled approximation where
\[
\Lambda_0 = \Lambda_1 = [1 1 1 1 0 0 \ldots 0], \quad \Lambda_2 = \Lambda_1 \cdot L^{-1},
\]
and
\[
\Lambda_2 = \Lambda_1 \cdot L^{-1}.
\]
We can thus analyse the following expressions on three consecutive keystream words \( z(t-1), z(t), z(t+1) \):
\[
\begin{align*}
z(t-1) &= (S^{-1}(L^{-1} \cdot \hat{R}2) \boxplus_8 T1(t-1)) \oplus S^{-1}(L^{-1} \cdot \hat{R}3), \\
z(t) &= (\hat{R}1 \boxplus_8 T1(t)) \oplus \hat{R}2, \\
L^{-1}z(t+1) &= L^{-1}(\sigma(\hat{R}2 \boxplus_8 (\hat{R}3 \oplus T2(t)))) \boxplus_8 T1(t+1) \oplus S(\hat{R}1).
\end{align*}
\]

The exampled approximation is the sum (\( \oplus \)) of the first 4 bytes of the above 3 expressions. In order to make it easier to follow our derivations we give explicit matrices for \( L^{-1} \) and \( L^{-1} \sigma \) in Listing 4.

--- matrix \( 'L'\{\text{-}1\}' ---

\[
\begin{array}{cccccccc}
 e & b & d & 9 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
\end{array}
\]

--- matrix \( 'L'\{\text{-}1\} \cdot \sigma \) ---

\[
\begin{array}{cccccccc}
 e & b & d & 9 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
\end{array}
\]

Listing 4: Matrices \( L^{-1} \) and \( L^{-1} \sigma \).

We first note that with \( \boxplus_8 \) for any 16-byte expression \( W \) we have:
\[
\sum_{i=0}^{3} [L^{-1} \sigma, W]_i = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{4j+i, 4j+i, 4j+i, 4j+i} W_{4j+i, 4j+i, 4j+i, 4j+i},
\]
where the coefficients are:
\[
c = [e \ b \ d \ 9 \ b \ d \ 9 \ e \ d \ 9 \ e \ b \ 9 \ e \ b \ d].
\]
Then we get the following linear approximation of the FSM:

\[
\sum_{i=0}^{3} \left[ L^{-1}z^{(i+1)} \oplus z^{(i)} \oplus z^{(t-1)} \right]_i = \sum_{i=0}^{3} \left( S(\hat{R}_{1i}) \oplus (\hat{R}_{1i} \boxplus s T_1^{(t)} \oplus T_1^{(t)}) \right)_{\text{Noise } N1_i}
\]

\[
\oplus \sum_{i=0}^{3} \left[ \hat{R}_{2i} \oplus \sum_{j=0}^{3} c_{4i+j} [ (\hat{R}_{24i+j} \boxplus s (\hat{R}_{34i+j} \oplus \hat{X}_{i,j}) \boxplus s \hat{Y}_{i,j}) \oplus \hat{X}_{i,j} \oplus \hat{Y}_{i,j} \oplus \hat{X}_{i,j} \oplus \hat{Y}_{i,j} ] \right]_{\text{Linear part } C_i \text{ of } N2_i}
\]

\[
\oplus \sum_{i=0}^{3} \left[ S^{-1} \left( \sum_{j=0}^{3} c_{4i+j} \hat{R}_{24i+j} \right) \boxplus s T_1^{(t-1)} \right]_{\text{Linear part } A_i \text{ of } N2_i} \oplus S^{-1} \left( \sum_{j=0}^{3} c_{4i+j} \hat{R}_{34i+j} \right)_{\text{Linear part } B_i \text{ of } N2_i},
\]

where \( \hat{X}_{i,j} = T_2^{(t)} \) and \( \hat{Y}_{i,j} = (s^{-1}T_1^{(t)})_{4i+j} \).

In the above it now becomes clear that we need to compute byte-oriented distributions of two independent noise variables \( N1 \) and \( N2 \). The first noise is trivial to compute, while the second \( N2 \) is a bit more complicated, but it can be computed with the technique mentioned in Section 3.4.1. I.e., for every \( i \) we first compute 4 partial joint distributions \( D_{i,j}(A_i|B_j|C_j) \) of linear parts \( A|B|C \) for each \( j = 0 \ldots 3 \), then we use Fast Walsh-Hadamard Transform to perform the XOR-convolutions in order to get the complete 24-bit joint distribution \( D_i(A_i|B_i|C_i) \) for all possible 32-bit input arguments running over \( j = 0, 1, 2, 3 \), for that certain \( i \). Having that joint distribution \( D_i(A_i|B_i|C_i) \) being computed it is then trivial to compute the 8-bit sub-noise distribution \( N2_i \); we simply loop over all possible choices of \( A_i, B_i, C_i, T_1^{(t-1)} \), and approximate \( ... \boxplus s T_1^{(t-1)} \) by adding that term outside of \( S^{-1} \), near by the linear part \( A \). I.e., we compute

\[
\Pr\{N2_i = (S^{-1}(A_i) \boxplus s T_1^{(t-1)} \oplus T_1^{(t-1)} \oplus S^{-1}(B_i) \oplus C_i) + = \frac{1}{2^8} \cdot D_i(A_i|B_i|C_i).
\]

We repeat the above for each \( i \), and the total noise \( N2 \) is the XOR-convolution of the four independent sub-distributions \( N2_i \).

We would like to note that the first sub-distribution \( N2_0 \) has a smaller bias since 4 bytes of the term \( \hat{R}_{2i} \) in the linear part \( C \) of \( N2 \) are added to \( N2_0 \), while the remaining 3 sub-distributions are free from these terms. This means that \( N2_0 \) includes 4 approximations of type n3 with a smaller bias (\( \approx 2^{-3.3} \)) than those of the types n1 (\( \approx 2^{1.1} \)) and n2 (\( \approx 2^{-2.9} \)). Our computation results are as follows:

\[
\begin{align*}
\epsilon(N2_{i=0}) &\approx 2^{-53.828334} \\
\epsilon(N2_{i=1..3}) &\approx 2^{-30.382642} \\
\epsilon(N1_{i=0..3}) &\approx 2^{-2.920807} \\
\epsilon(\sum_{i=0}^{3} N1_i) &\approx 2^{-26.446376} \\
\epsilon(\sum_{i=0}^{3} N2_i) &\approx 2^{-187.562693} \\
\epsilon(N_{\text{tot}}) &\approx 2^{-214.48865} \\
\epsilon(2 \times N_{\text{tot}}) &\approx 2^{-429.674887}
\end{align*}
\]