Overview of SNOW-V [3]

We will get a brief overview of SNOW-V - A New Member in the SNOW family of stream ciphers , successor of SNOW 3G :

Introduction –

In this paper a new SNOW stream cipher, called SNOW-V, is proposed. For the next generation system, called 5G, we needed some fundamental changes in system architecture and security level which cannot be fulfilled by SNOW-3G. So, the author introduced SNOW-V (V for Virtualization), in order to provide atleast 20 Gbps throughput needed for 5G communication.
Current SNOW-3G provides a speed of roughly 9Gbps in pure software implementation, which is not enough for 5G communication. So while keeping the basic SNOW-3G architecture plus adding some new features, a faster stream cipher with same security level as AES can be obtained, called as SNOW-V.
SNOW-V architecture is based on SNOW-3G additionally having acceleration instructions using AES round functions as nowadays new CPUs from both AMD & Intel comes with AES cores and SIMD instructions making large vector calculations easier to handle. Hence, the major changes in SNOW-V design are AES round function 128-bit registers in FSM with two LFSRs operating 8 times faster than the speed of FSM.
The author also proposed an AEAD (Authenticated Encryption with Associated Data) operational mode to provide both confidentiality and integrity protection making it more secure than other algorithms.

Design & Operation –

The SNOW-V consists of two 256-bit Linear-Feedback Shift Registers (LFSRs) that are continuously fed into a Finite-State Machine (FSM), consisting of two chained AES-128 Round Functions, that produces the key stream.

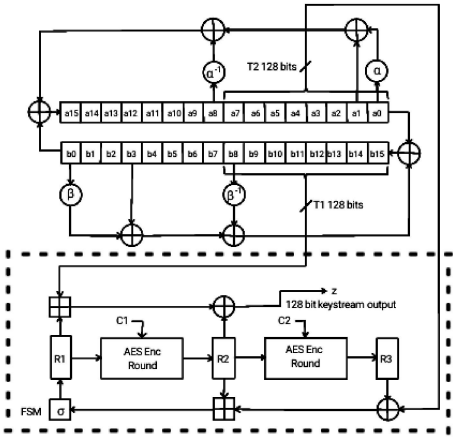

Figure 7: A schematic of SNOW-V

Linear-Feedback Shift Registers (LFSR) –

The SNOW-V consists two 256-bit LFSRs. Each of which consists if sixteen 16-bit cells. Let the two LFSRs be LFSR-A & LFSR-B containing cells $a_0, ..., a_{15}$ & $b_0, ..., b_{15}$ respectively. Each cell is a finite-field element over the extension $F^a_{2^{16}} = F_2[x]/g^A(x)$ for LFSR-A and $F^b_{2^{16}} = F_2[x]/g^B(x)$ for LFSR-B such that $g^A(x)$ is a primitive polynomial of the form:-

$$g^A(x) \quad = \quad x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \epsilon F_2[x].$$

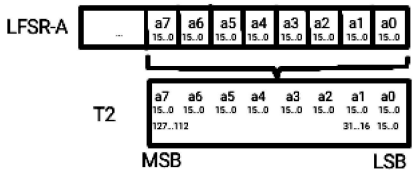

Figure 8: LFSR-A

Similarly, $g^B(x)$ is given by the primitive polynomial:-

$$g^B(x) \quad = \quad x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \epsilon F_2[x].$$

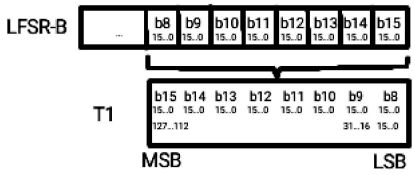

Figure 9: LFSR-B

The feedback function of LFSR-A is given by the equation:-

$$a_{16} \quad = \quad b_0 \ \oplus \ \alpha a_0 \ \oplus \ a_1 \ \oplus \ \alpha^{-1} a_8,$$

where α is a root of $g^A(x)$ and α^{-1} its inverse over $F^A_{2^{16}}$. Similarly, we can denote the feedback function of LFSR-B by:-

$$b_{16} \quad = \quad a_0 \ \oplus \ \beta b_0 \ \oplus \ b_1 \ \oplus \ \beta^{-1} b_8,$$

where β is a root of g^B and β^{-1} its inverse over $F^B_{2^{16}}$.

An important property of both $F^A_{2^{16}}$ $F^B_{2^{16}}$ is the efficient execution of the field multiplication by $\alpha, \alpha^{-1} and \beta, \beta^{-1}$, respectively. As a matter of fact, the operation can be encoded by two simple routines both consisting of a bit-wise shift and a XOR. Since α is a primitive element in $F^A_{2^{16}}$, it is possible to write each element $\omega \epsilon F^A_{2^{16}}$ in the basis $(1, \alpha, ..., \alpha^{15})$ such that , where ω_i are the bits of ω. As a consequence, we can rewrite the multiplication $\omega\alpha$ as:-

$$\begin{aligned}\omega\alpha &= (\omega_{15}\alpha^{15} + ... + \omega_1\alpha + \omega_0)\alpha \\ &= \omega_{15}(\alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^3 + \alpha^2 + \alpha + 1) \\ &\quad + \omega_{14}\alpha^{15} + ... + \omega_0\alpha.\end{aligned}$$

The upper byte of the above product can be written as $[\omega_{14}, \omega_{13}, ..., \omega_7] + \omega_{15} \cdot C_H$ and the lower byte as $[\omega_6, \omega_5, ..., \omega_0] + \omega_{15} \cdot C_L where C_H = 10011001 and C_L = 00001111$. A similar reasoning can be followed for the multiplication by the inverse element α^{-1} and for the field $F_{2^{16}}^B$.

Finite-State Machine (FSM) –

The Finite-State Machine consists of two chained AES-128 round function that are continuously fed with the updated LFSR states and three 128-bit registers namely R1, R2 R3 which can be seen in Fig.[]. Key addition is not performed in the AES cores, i.e. the computation only consists of the substitution layer, the Shiftrows and Mixcolumn procedures. We denote by $\boxplus_{32}$ an adder that operates on 32-bit blocks. Furthermore, σ represents a byte-wise permutation of the form:-

$$\sigma = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15].$$

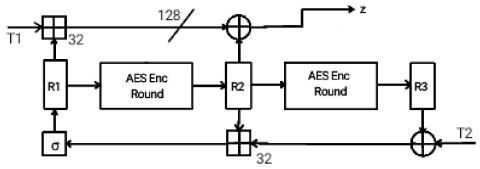

Figure 10: Schematic of Finite-State Machine (FSM)

Initialization Routine –

The initialization phase lasts for 16 rounds during which the FSM and LFSR are iteratively updated. Furthermore, the usual keystream output z is mixed into LFSR-A in each iteration. Finally, the state R1 is additionally updated with the key during the last two rounds.

Keystream Routine –

The keystream routine only differs from the initialization procedure in Initialization Routine Algorithm by omitting the supplementary updates of LFSR-A and R1.

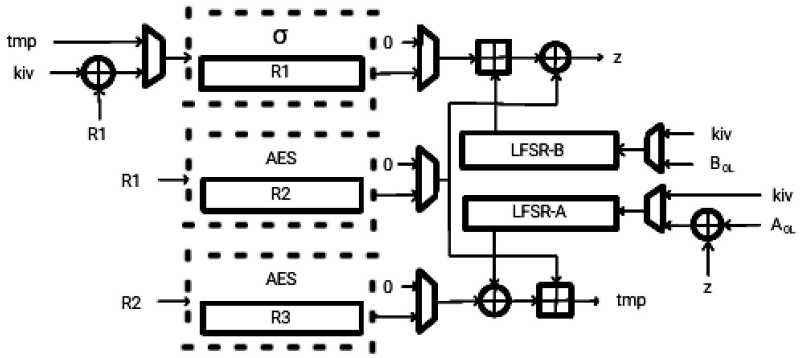

Figure 11: Byte-serial low-area architecture of SNOW-V

AEAD Mode of Operation –

Authenticated Encryption with Associated Data (AEAD), provides confidentiality, integrity, and authenticity assurances on the data. GMAC (Galois Message Authentication Code) is used to generate authentication tag. Keystream generation process is the same as in the normal mode, except $C^1 = 0x0024406480A4C0E40420446084A0C4E0$

Sender: $Ciphertext = Keystream - 1 \oplus Plaintext$
T = GMAC (Keystream-2, AAD, Ciphertext)

Receiver: T' = GMAC (Keystream-2, AAD, Ciphertext),
if T' = T
$Plaintext = Keystream - 1 \oplus Ciphertext$
else
Output Fail (data might be tampered)

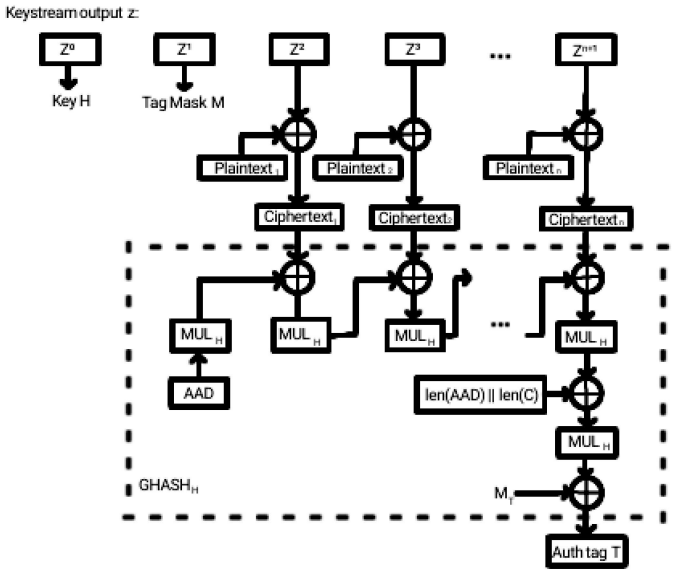

Figure 12: Use of SNOW-V with $GHASH_H$ to enable AEAD

Software Implementation Performance –

Since, 5G radio interface requires virtualization of the network functions with 10 Gbps as uplink speed and 20 Gbps as downlink speed, at peak data rates. Classical encryption algorithms cannot reach these high speeds in pure software environment without any good hardware support.

Nowadays new CPUs from both CPU vendors such as AMD Intel comes with AES cores and SIMD instructions making large vector calculations easier to handle. They also include typical instructions such 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 as shown in fig. below:- Using these features such as SIMD instructions, AES round functions and other basic

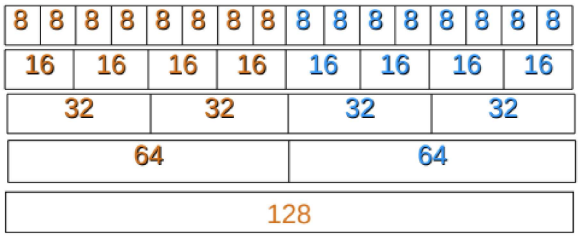

Figure 13: SIMD Structure

functions like XOR, AND, etc., SNOW-V is designed to perform very fast in software and also give

great security.
Advantages of modern CPUs:-

1. SIMD Structure
    - Two LFSRs can fit into 2x 256-bit registers: __m256i
    - Registers in FSM can fit into 3x 128-bit registers: __m128i
2. Intrinsic Instructions
    - AES round: _mm_aesenc_si128(__m128i a, __m128i RoundKey)
    - Arithmetic additions: _mm_add_epi32(__m128i a, __m128i b)

Security Analysis –

Common Attacks on Stream Ciphers:-

- Attack on Initialization
    - Differential Attacks: trace differences' transfer and discover where the cipher behaves non-random
    - Chosen-IV attack: adversary attempts to build a distinguisher to introduce randomness failures in the ouput by setting arbitrary IV values, e.g., MDM attack.
- Linear Distinguishing Attacks
    - Distinguish the cipher from random oracle
- Time-Memory-Data Tradeoff Attacks
    - Balance/reduce one/two parameters in favor of the others
- Slide Attacks
    - Analyze the key schedule and exploit weaknesses in it to break the cipher
- Attacks on the Authentication Mode
    - Maximum Degree Monomial (MDM) Attack

MDM Attack on SNOW-V –

- Select 1 to 24 bits from the (K,IV) space
- Run through all possible values, other bits are set 0
- XOR all the outputs to get the MDM
- The results have a long zeros before random-like, e.g., 000...00010110...
- The outputs of the first 7 rounds are not random, it would be not safe if we reduce the initialization rounds to 7 or fewer
- 16 rounds of initialization looks safe, it is not likely that an attacker would be able to build a distinguisher after 16 rounds

Conclusion –

The author proposed a new 128-bit stream cipher called SNOW-V. The design of this cipher was based on the predecessor stream cipher - SNOW-3G but use the AES round function instruction support present in most of the modern CPUs. SNOW-V is found to be faster than AES in all comparable modes of operation(single thread implementation in software). SNOW-V is found to be resistant against basic and advanced attacks possible. The paper finally gives an AEAD mode of operation based on GCM scheme. The SNOW-V architecture is found to be theoretically faster than both AES-256 SNOW-3G with security level not lower tha AES-256 SNOW-3G. Full design along with test vectors and reference implementation can be found in the official paper [3].

Hardware Implementation of SNOW-V

SNOW-V (Top Module)

```
`timescale 1ns / 1ps

module SNOW_V(
    input [255:0] s_key,
    input [127:0] IV,
    input clk,
    output [127:0] keystream
    );
    LFSR l1(.s_key(s_key), .IV(IV), .clk(clk), .keystream(keystream));
endmodule
```

Linear Feedback Shift Register (LFSR)

```
`timescale 1ns / 1ps

module LFSR(
    input [255:0] s_key,
    input [127:0] IV,
    input clk,
    output wire [127:0] keystream
    );

    reg [15:0] a0;
    reg [15:0] a1;
    reg [15:0] a2;
    reg [15:0] a3;
    reg [15:0] a4;
    reg [15:0] a5;
    reg [15:0] a6;
    reg [15:0] a7;
    reg [15:0] a8;
    reg [15:0] a9;
    reg [15:0] a10;
    reg [15:0] a11;
    reg [15:0] a12;
    reg [15:0] a13;
    reg [15:0] a14;
    reg [15:0] a15;

    reg [15:0] b0;
    reg [15:0] b1;
    reg [15:0] b2;
    reg [15:0] b3;
    reg [15:0] b4;
    reg [15:0] b5;
    reg [15:0] b6;
    reg [15:0] b7;
    reg [15:0] b8;
    reg [15:0] b9;
    reg [15:0] b10;
    reg [15:0] b11;
    reg [15:0] b12;
    reg [15:0] b13;
    reg [15:0] b14;
    reg [15:0] b15;

    wire [127:0] T1;
    wire [127:0] T2;

    function [15 : 0] MUL_x(input [15 : 0] V, input [15 : 0] c);
    begin
        if(V & 16'h8000)
            MUL_x = (V << 1) ^ c;
```

```
        else
            MUL_x = (V << 1);
    end
    endfunction

    function [15 : 0] MUL_x_inv(input [15 : 0] V, input [15 : 0] d);
    begin
        if(V & 16'h0001)
            MUL_x_inv = (V >> 1) ^ d;
        else
            MUL_x_inv = (V >> 1);
    end
    endfunction

    always@(posedge clk)
    begin
        b15 = s_key[255:240];
        b14 = s_key[239:224];
        b13 = s_key[223:208];
        b12 = s_key[207:192];
        b11 = s_key[191:176];
        b10 = s_key[175:160];
        b9 = s_key[159:144];
        b8 = s_key[143:128];
        b7 = 16'd0;
        b6 = 16'd0;
        b5 = 16'd0;
        b4 = 16'd0;
        b3 = 16'd0;
        b2 = 16'd0;
        b1 = 16'd0;
        b0 = 16'd0;
        #640;
        b15 <= (a0 ^ (MUL_x_inv(b8,16'hE4B1) ^ (MUL_x(b0,16'hC963) ^ b3)));
        b14 <= b15;
        b13 <= b14;
        b12 <= b13;
        b11 <= b12;
        b10 <= b11;
        b9 <= b10;
        b8 <= b9;
        b7 <= b8;
        b6 <= b7;
        b5 <= b6;
        b4 <= b5;
        b3 <= b4;
        b2 <= b3;
        b1 <= b2;
        b0 <= b1;
    end

    always@(posedge clk)
    begin
        a15 = s_key[127:112];
        a14 = s_key[111:96];
        a13 = s_key[95:80];
        a12 = s_key[79:64];
        a11 = s_key[63:48];
        a10 = s_key[47:32];
        a9 = s_key[31:16];
        a8 = s_key[15:0];
        a7 = IV[127:112];
        a6 = IV[111:96];
        a5 = IV[95:80];
        a4 = IV[79:64];
        a3 = IV[63:48];
        a2 = IV[47:32];
        a1 = IV[31:16];
        a0 = IV[15:0];
```

```
        #640;
        a15 <= (b0 ^ (MUL_x_inv(a8,16'hCC87) ^ (MUL_x(a0,16'h990F) ^ a1)));
        a14 <= a15;
        a13 <= a14;
        a12 <= a13;
        a11 <= a12;
        a10 <= a11;
        a9 <= a10;
        a8 <= a9;
        a7 <= a8;
        a6 <= a7;
        a5 <= a6;
        a4 <= a5;
        a3 <= a4;
        a2 <= a3;
        a1 <= a2;
        a0 <= a1;
    end

    assign T1[127:112] = b8;
    assign T1[111:96] = b9;
    assign T1[95:80] = b10;
    assign T1[79:64] = b11;
    assign T1[63:48] = b12;
    assign T1[47:32] = b13;
    assign T1[31:16] = b14;
    assign T1[15:0] = b15;

    assign T2[127:112] = a7;
    assign T2[111:96] = a6;
    assign T2[95:80] = a5;
    assign T2[79:64] = a4;
    assign T2[63:48] = a3;
    assign T2[47:32] = a2;
    assign T2[31:16] = a1;
    assign T2[15:0] = a0;

    FSM f1(.T1(T1), .T2(T2), .clk(clk), .keystream(keystream));

endmodule
```

Finite-State Machine (FSM)

```
`timescale 1ns / 1ps

module FSM (
      input [127:0] T1,
      input [127:0] T2,
      input clk,
      output [127:0] keystream
    );
      reg [127:0] R1 = 128'd0;
      reg [127:0] R2 = 128'd0;
      reg [127:0] R3 = 128'd0;
      wire [127:0] x0;
      wire [127:0] x1;
      wire [127:0] x2;
      wire [127:0] x3;
      wire [127:0] x4;
      wire [127:0] temp;

      always@(posedge clk)
      begin
        R1 <= x4;
        R2 <= x2;
        R3 <= x3;
      end

```

```
      assign temp = (R3 ^ T2);

      Modulo_Addition mt1(.a(temp[127:96]), .b(R2[127:96]), .sum(x1[127:96]));
      Modulo_Addition mt2(.a(temp[95:64]), .b(R2[95:64]), .sum(x1[95:64]));
      Modulo_Addition mt3(.a(temp[63:32]), .b(R2[63:32]), .sum(x1[63:32]));
      Modulo_Addition mt4(.a(temp[31:0]), .b(R2[31:0]), .sum(x1[31:0]));

      sigma s1(.in(x1), .out(x4));

      AES_round_func a1(.in(R1), .round_key(128'd0), .out(x2));

      AES_round_func a2(.in(R2), .round_key(128'd0), .out(x3));

      Modulo_Addition tm1(.a(T1[127:96]), .b(R1[127:96]), .sum(x0[127:96]));
      Modulo_Addition tm2(.a(T1[95:64]), .b(R1[95:64]), .sum(x0[95:64]));
      Modulo_Addition tm3(.a(T1[63:32]), .b(R1[63:32]), .sum(x0[63:32]));
      Modulo_Addition tm4(.a(T1[31:0]), .b(R1[31:0]), .sum(x0[31:0]));

      assign keystream = (x0 ^ R2);

endmodule
```

AES Round Function

```
`timescale 1ns / 1ps

module AES_round_func(
                       input  [127 : 0]  in,
                       input  [127 : 0]  round_key,
                       output  [127 : 0] out
                      );

  function [7 : 0] sbox(input [7 : 0] b);
    begin
      case (b)
        8'h00: sbox = 8'h63;
        8'h01: sbox = 8'h7c;
        8'h02: sbox = 8'h77;
        8'h03: sbox = 8'h7b;
        8'h04: sbox = 8'hf2;
        8'h05: sbox = 8'h6b;
        8'h06: sbox = 8'h6f;
        8'h07: sbox = 8'hc5;
        8'h08: sbox = 8'h30;
        8'h09: sbox = 8'h01;
        8'h0a: sbox = 8'h67;
        8'h0b: sbox = 8'h2b;
        8'h0c: sbox = 8'hfe;
        8'h0d: sbox = 8'hd7;
        8'h0e: sbox = 8'hab;
        8'h0f: sbox = 8'h76;
        8'h10: sbox = 8'hca;
        8'h11: sbox = 8'h82;
        8'h12: sbox = 8'hc9;
        8'h13: sbox = 8'h7d;
        8'h14: sbox = 8'hfa;
        8'h15: sbox = 8'h59;
        8'h16: sbox = 8'h47;
        8'h17: sbox = 8'hf0;
        8'h18: sbox = 8'had;
        8'h19: sbox = 8'hd4;
        8'h1a: sbox = 8'ha2;
        8'h1b: sbox = 8'haf;
        8'h1c: sbox = 8'h9c;
        8'h1d: sbox = 8'ha4;
        8'h1e: sbox = 8'h72;
        8'h1f: sbox = 8'hc0;
        8'h20: sbox = 8'hb7;
```

```
45          8'h21: sbox = 8'hfd;
46          8'h22: sbox = 8'h93;
47          8'h23: sbox = 8'h26;
48          8'h24: sbox = 8'h36;
49          8'h25: sbox = 8'h3f;
50          8'h26: sbox = 8'hf7;
51          8'h27: sbox = 8'hcc;
52          8'h28: sbox = 8'h34;
53          8'h29: sbox = 8'ha5;
54          8'h2a: sbox = 8'he5;
55          8'h2b: sbox = 8'hf1;
56          8'h2c: sbox = 8'h71;
57          8'h2d: sbox = 8'hd8;
58          8'h2e: sbox = 8'h31;
59          8'h2f: sbox = 8'h15;
60          8'h30: sbox = 8'h04;
61          8'h31: sbox = 8'hc7;
62          8'h32: sbox = 8'h23;
63          8'h33: sbox = 8'hc3;
64          8'h34: sbox = 8'h18;
65          8'h35: sbox = 8'h96;
66          8'h36: sbox = 8'h05;
67          8'h37: sbox = 8'h9a;
68          8'h38: sbox = 8'h07;
69          8'h39: sbox = 8'h12;
70          8'h3a: sbox = 8'h80;
71          8'h3b: sbox = 8'he2;
72          8'h3c: sbox = 8'heb;
73          8'h3d: sbox = 8'h27;
74          8'h3e: sbox = 8'hb2;
75          8'h3f: sbox = 8'h75;
76          8'h40: sbox = 8'h09;
77          8'h41: sbox = 8'h83;
78          8'h42: sbox = 8'h2c;
79          8'h43: sbox = 8'h1a;
80          8'h44: sbox = 8'h1b;
81          8'h45: sbox = 8'h6e;
82          8'h46: sbox = 8'h5a;
83          8'h47: sbox = 8'ha0;
84          8'h48: sbox = 8'h52;
85          8'h49: sbox = 8'h3b;
86          8'h4a: sbox = 8'hd6;
87          8'h4b: sbox = 8'hb3;
88          8'h4c: sbox = 8'h29;
89          8'h4d: sbox = 8'he3;
90          8'h4e: sbox = 8'h2f;
91          8'h4f: sbox = 8'h84;
92          8'h50: sbox = 8'h53;
93          8'h51: sbox = 8'hd1;
94          8'h52: sbox = 8'h00;
95          8'h53: sbox = 8'hed;
96          8'h54: sbox = 8'h20;
97          8'h55: sbox = 8'hfc;
98          8'h56: sbox = 8'hb1;
99          8'h57: sbox = 8'h5b;
100         8'h58: sbox = 8'h6a;
101         8'h59: sbox = 8'hcb;
102         8'h5a: sbox = 8'hbe;
103         8'h5b: sbox = 8'h39;
104         8'h5c: sbox = 8'h4a;
105         8'h5d: sbox = 8'h4c;
106         8'h5e: sbox = 8'h58;
107         8'h5f: sbox = 8'hcf;
108         8'h60: sbox = 8'hd0;
109         8'h61: sbox = 8'hef;
110         8'h62: sbox = 8'haa;
111         8'h63: sbox = 8'hfb;
112         8'h64: sbox = 8'h43;
113         8'h65: sbox = 8'h4d;
```

```
114         8'h66: sbox = 8'h33;
115         8'h67: sbox = 8'h85;
116         8'h68: sbox = 8'h45;
117         8'h69: sbox = 8'hf9;
118         8'h6a: sbox = 8'h02;
119         8'h6b: sbox = 8'h7f;
120         8'h6c: sbox = 8'h50;
121         8'h6d: sbox = 8'h3c;
122         8'h6e: sbox = 8'h9f;
123         8'h6f: sbox = 8'ha8;
124         8'h70: sbox = 8'h51;
125         8'h71: sbox = 8'ha3;
126         8'h72: sbox = 8'h40;
127         8'h73: sbox = 8'h8f;
128         8'h74: sbox = 8'h92;
129         8'h75: sbox = 8'h9d;
130         8'h76: sbox = 8'h38;
131         8'h77: sbox = 8'hf5;
132         8'h78: sbox = 8'hbc;
133         8'h79: sbox = 8'hb6;
134         8'h7a: sbox = 8'hda;
135         8'h7b: sbox = 8'h21;
136         8'h7c: sbox = 8'h10;
137         8'h7d: sbox = 8'hff;
138         8'h7e: sbox = 8'hf3;
139         8'h7f: sbox = 8'hd2;
140         8'h80: sbox = 8'hcd;
141         8'h81: sbox = 8'h0c;
142         8'h82: sbox = 8'h13;
143         8'h83: sbox = 8'hec;
144         8'h84: sbox = 8'h5f;
145         8'h85: sbox = 8'h97;
146         8'h86: sbox = 8'h44;
147         8'h87: sbox = 8'h17;
148         8'h88: sbox = 8'hc4;
149         8'h89: sbox = 8'ha7;
150         8'h8a: sbox = 8'h7e;
151         8'h8b: sbox = 8'h3d;
152         8'h8c: sbox = 8'h64;
153         8'h8d: sbox = 8'h5d;
154         8'h8e: sbox = 8'h19;
155         8'h8f: sbox = 8'h73;
156         8'h90: sbox = 8'h60;
157         8'h91: sbox = 8'h81;
158         8'h92: sbox = 8'h4f;
159         8'h93: sbox = 8'hdc;
160         8'h94: sbox = 8'h22;
161         8'h95: sbox = 8'h2a;
162         8'h96: sbox = 8'h90;
163         8'h97: sbox = 8'h88;
164         8'h98: sbox = 8'h46;
165         8'h99: sbox = 8'hee;
166         8'h9a: sbox = 8'hb8;
167         8'h9b: sbox = 8'h14;
168         8'h9c: sbox = 8'hde;
169         8'h9d: sbox = 8'h5e;
170         8'h9e: sbox = 8'h0b;
171         8'h9f: sbox = 8'hdb;
172         8'ha0: sbox = 8'he0;
173         8'ha1: sbox = 8'h32;
174         8'ha2: sbox = 8'h3a;
175         8'ha3: sbox = 8'h0a;
176         8'ha4: sbox = 8'h49;
177         8'ha5: sbox = 8'h06;
178         8'ha6: sbox = 8'h24;
179         8'ha7: sbox = 8'h5c;
180         8'ha8: sbox = 8'hc2;
181         8'ha9: sbox = 8'hd3;
182         8'haa: sbox = 8'hac;
```

```
183         8'hab: sbox = 8'h62;
184         8'hac: sbox = 8'h91;
185         8'had: sbox = 8'h95;
186         8'hae: sbox = 8'he4;
187         8'haf: sbox = 8'h79;
188         8'hb0: sbox = 8'he7;
189         8'hb1: sbox = 8'hc8;
190         8'hb2: sbox = 8'h37;
191         8'hb3: sbox = 8'h6d;
192         8'hb4: sbox = 8'h8d;
193         8'hb5: sbox = 8'hd5;
194         8'hb6: sbox = 8'h4e;
195         8'hb7: sbox = 8'ha9;
196         8'hb8: sbox = 8'h6c;
197         8'hb9: sbox = 8'h56;
198         8'hba: sbox = 8'hf4;
199         8'hbb: sbox = 8'hea;
200         8'hbc: sbox = 8'h65;
201         8'hbd: sbox = 8'h7a;
202         8'hbe: sbox = 8'hae;
203         8'hbf: sbox = 8'h08;
204         8'hc0: sbox = 8'hba;
205         8'hc1: sbox = 8'h78;
206         8'hc2: sbox = 8'h25;
207         8'hc3: sbox = 8'h2e;
208         8'hc4: sbox = 8'h1c;
209         8'hc5: sbox = 8'ha6;
210         8'hc6: sbox = 8'hb4;
211         8'hc7: sbox = 8'hc6;
212         8'hc8: sbox = 8'he8;
213         8'hc9: sbox = 8'hdd;
214         8'hca: sbox = 8'h74;
215         8'hcb: sbox = 8'h1f;
216         8'hcc: sbox = 8'h4b;
217         8'hcd: sbox = 8'hbd;
218         8'hce: sbox = 8'h8b;
219         8'hcf: sbox = 8'h8a;
220         8'hd0: sbox = 8'h70;
221         8'hd1: sbox = 8'h3e;
222         8'hd2: sbox = 8'hb5;
223         8'hd3: sbox = 8'h66;
224         8'hd4: sbox = 8'h48;
225         8'hd5: sbox = 8'h03;
226         8'hd6: sbox = 8'hf6;
227         8'hd7: sbox = 8'h0e;
228         8'hd8: sbox = 8'h61;
229         8'hd9: sbox = 8'h35;
230         8'hda: sbox = 8'h57;
231         8'hdb: sbox = 8'hb9;
232         8'hdc: sbox = 8'h86;
233         8'hdd: sbox = 8'hc1;
234         8'hde: sbox = 8'h1d;
235         8'hdf: sbox = 8'h9e;
236         8'he0: sbox = 8'he1;
237         8'he1: sbox = 8'hf8;
238         8'he2: sbox = 8'h98;
239         8'he3: sbox = 8'h11;
240         8'he4: sbox = 8'h69;
241         8'he5: sbox = 8'hd9;
242         8'he6: sbox = 8'h8e;
243         8'he7: sbox = 8'h94;
244         8'he8: sbox = 8'h9b;
245         8'he9: sbox = 8'h1e;
246         8'hea: sbox = 8'h87;
247         8'heb: sbox = 8'he9;
248         8'hec: sbox = 8'hce;
249         8'hed: sbox = 8'h55;
250         8'hee: sbox = 8'h28;
251         8'hef: sbox = 8'hdf;
```

```
252         8'hf0: sbox = 8'h8c;
253         8'hf1: sbox = 8'ha1;
254         8'hf2: sbox = 8'h89;
255         8'hf3: sbox = 8'h0d;
256         8'hf4: sbox = 8'hbf;
257         8'hf5: sbox = 8'he6;
258         8'hf6: sbox = 8'h42;
259         8'hf7: sbox = 8'h68;
260         8'hf8: sbox = 8'h41;
261         8'hf9: sbox = 8'h99;
262         8'hfa: sbox = 8'h2d;
263         8'hfb: sbox = 8'h0f;
264         8'hfc: sbox = 8'hb0;
265         8'hfd: sbox = 8'h54;
266         8'hfe: sbox = 8'hbb;
267         8'hff: sbox = 8'h16;
268       endcase // case (b)
269     end
270   endfunction // sbox
271
272   function [127 : 0] subbytes(input [127 : 0] block);
273     begin
274       subbytes = {sbox(block[127 : 120]), sbox(block[119 : 112]),
275                   sbox(block[111 : 104]), sbox(block[103 : 096]),
276                   sbox(block[095 : 088]), sbox(block[087 : 080]),
277                   sbox(block[079 : 072]), sbox(block[071 : 064]),
278                   sbox(block[063 : 056]), sbox(block[055 : 048]),
279                   sbox(block[047 : 040]), sbox(block[039 : 032]),
280                   sbox(block[031 : 024]), sbox(block[023 : 016]),
281                   sbox(block[015 : 008]), sbox(block[007 : 000])};
282     end
283   endfunction
284
285   function [7 : 0] gm2(input [7 : 0] op);
286     begin
287       gm2 = {op[6 : 0], 1'b0} ^ (8'h1b & {8{op[7]}});
288     end
289   endfunction
290
291   function [7 : 0] gm3(input [7 : 0] op);
292     begin
293       gm3 = gm2(op) ^ op;
294     end
295   endfunction
296
297   function [31 : 0] mixw(input [31 : 0] w);
298     reg [7 : 0] b0, b1, b2, b3;
299     reg [7 : 0] mb0, mb1, mb2, mb3;
300     begin
301       b0 = w[31 : 24];
302       b1 = w[23 : 16];
303       b2 = w[15 : 08];
304       b3 = w[07 : 00];
305
306       mb0 = gm2(b0) ^ gm3(b1) ^ b2      ^ b3;
307       mb1 = b0      ^ gm2(b1) ^ gm3(b2) ^ b3;
308       mb2 = b0      ^ b1      ^ gm2(b2) ^ gm3(b3);
309       mb3 = gm3(b0) ^ b1      ^ b2      ^ gm2(b3);
310
311       mixw = {mb0, mb1, mb2, mb3};
312     end
313   endfunction
314
315   function [127 : 0] mixcolumns(input [127 : 0] data);
316     reg [31 : 0] w0, w1, w2, w3;
317     reg [31 : 0] ws0, ws1, ws2, ws3;
318     begin
319       w0 = data[127 : 096];
320       w1 = data[095 : 064];
```

```
      w2 = data[063 : 032];
      w3 = data[031 : 000];

      ws0 = mixw(w0);
      ws1 = mixw(w1);
      ws2 = mixw(w2);
      ws3 = mixw(w3);

      mixcolumns = {ws0, ws1, ws2, ws3};
    end
  endfunction // mixcolumns

  function [127 : 0] shiftrows(input [127 : 0] data);
    reg [31 : 0] w0, w1, w2, w3;
    reg [31 : 0] ws0, ws1, ws2, ws3;
    begin
      w0 = data[127 : 096];
      w1 = data[095 : 064];
      w2 = data[063 : 032];
      w3 = data[031 : 000];

      ws0 = {w0[31 : 24], w1[23 : 16], w2[15 : 08], w3[07 : 00]};
      ws1 = {w1[31 : 24], w2[23 : 16], w3[15 : 08], w0[07 : 00]};
      ws2 = {w2[31 : 24], w3[23 : 16], w0[15 : 08], w1[07 : 00]};
      ws3 = {w3[31 : 24], w0[23 : 16], w1[15 : 08], w2[07 : 00]};

      shiftrows = {ws0, ws1, ws2, ws3};
    end
  endfunction

  function [127 : 0] addroundkey(input [127 : 0] data, input [127 : 0] rkey);
    begin
      addroundkey = data ^ rkey;
    end
  endfunction


  reg [127 : 0] tmp_out;


  assign out = tmp_out;


  always @(*)
    begin : aes_round
      reg [127 : 0] subbytes_block, shiftrows_block, mixcolumns_block;

      subbytes_block   = subbytes(in);
      shiftrows_block  = shiftrows(subbytes_block);
      mixcolumns_block = mixcolumns(shiftrows_block);
      tmp_out          = addroundkey(mixcolumns_block, round_key);
    end
endmodule
```

Sigma Operation

```
`timescale 1ns / 1ps

module sigma(
    input [127:0] in,
    input clk,
    output [127:0] out
    );

    integer i,j;
    reg [127:0] temp = 128'd0;

    function [15 : 0] make16(input [15:0] a, input [15:0] b);
```

```
13      begin
14          assign make16 = ((a << 8) | b);
15      end
16      endfunction
17
18      function [31 : 0] make32(input [31:0] a, input [31:0] b);
19      begin
20          assign make32 = ((a << 16) | b);
21      end
22      endfunction
23
24      function [4 : 0] sigma(input [4:0] z);
25      begin
26          case(z)
27              4'd0: sigma = 4'd0;
28              4'd1: sigma = 4'd4;
29              4'd2: sigma = 4'd4;
30              4'd3: sigma = 4'd12;
31              4'd4: sigma = 4'd1;
32              4'd5: sigma = 4'd5;
33              4'd6: sigma = 4'd9;
34              4'd7: sigma = 4'd13;
35              4'd8: sigma = 4'd2;
36              4'd9: sigma = 4'd6;
37              4'd10: sigma = 4'd10;
38              4'd11: sigma = 4'd14;
39              4'd12: sigma = 4'd3;
40              4'd13: sigma = 4'd7;
41              4'd14: sigma = 4'd11;
42              4'd15: sigma = 4'd15;
43          endcase
44      end
45      endfunction
46
47      always @ (posedge clk)
48      begin
49          for (i = 0, j = 0 ; i < 128 ; i = i + 8, j = j + 1)
50          begin
51              temp[i +: 8] <= in[(sigma(j) >> 2) +: 8];
52          end
53      end
54
55      assign out = temp;
56
57  endmodule
```

Modulo Addition

```
`timescale 1ns / 1ps

module Modulo_Addition #(parameter two_32=40'd2147483648)(
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] sum
);
    integer a1,b1,sum1;

    always @ (*)
    begin
        a1 <= a;
        b1 <= b;
        sum1 <= a1 + b1;
        if (sum1 < two_32)
            sum <= sum1;
        else if (sum1 == two_32)
            sum <= 32'd0;
        else
            sum <= sum1 % two_32;
    end
endmodule
```

Testbench

```
`timescale 1ns / 1ps

module testbench;
    reg [255:0] s_key;
    reg [127:0] IV;
    reg clk;
    wire [127:0] keystream;

    SNOW_V tb(.s_key(s_key), .IV(IV), .clk(clk), .keystream(keystream));

    initial
    begin
        s_key = 256'h0f33b29436c2bd4f92e798f2ed0824f19b15acd9aacf80d8f3f46eea61237fc9;
        IV = 128'h122052e9c61e2e7e45208419998a007c;
        clk=0;
        forever #10 clk = ~clk;
    end
endmodule
```

Simulation of SNOW-V Algorithm

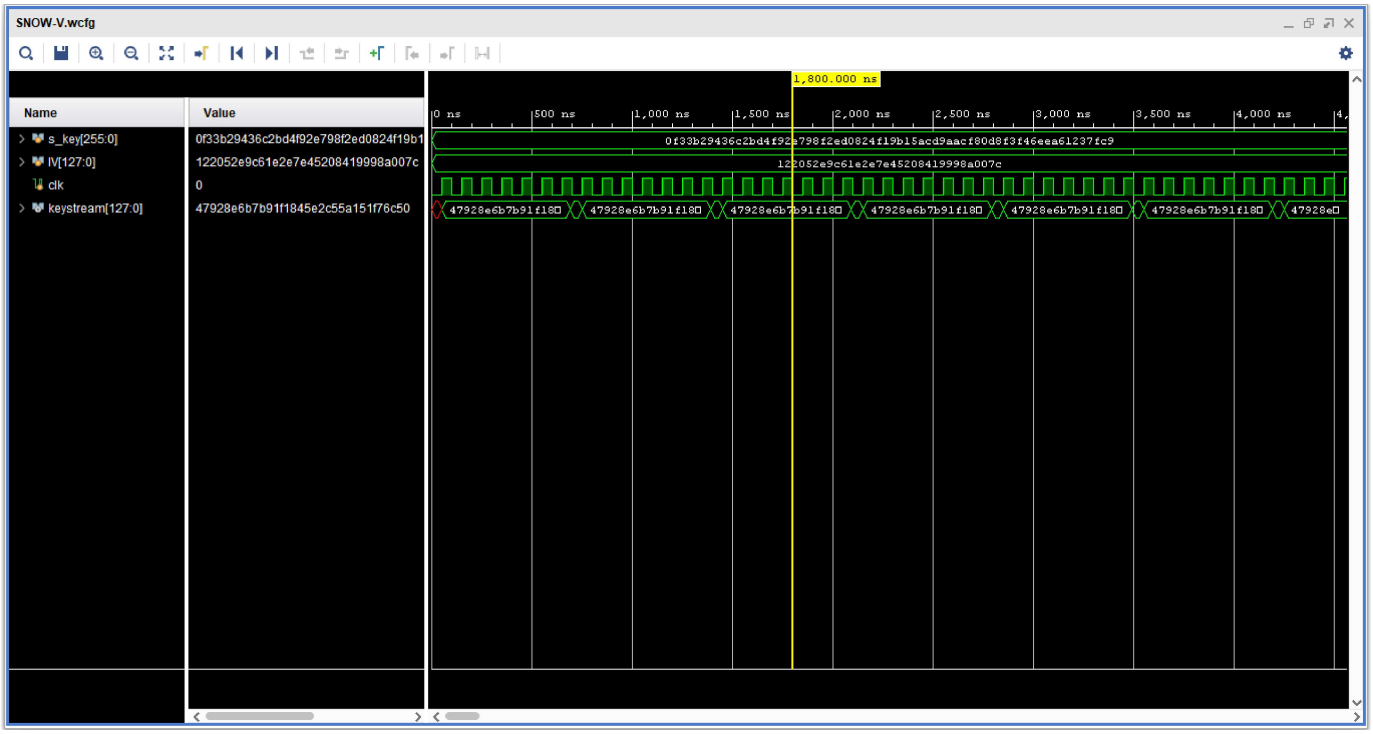

Figure 14: Simulation of SNOW-V Algorithm

"SNOW-V" – Test values Result of Simulation

Test values of s_key IV taken in testbench –

- S_key = 256'h0f33b29436c2bd4f92e798f2ed0824f19b15acd9aacf80d8f3f4 6eea61237fc9
- IV = 128'h122052e9c61e2e7e45208419998a007c
- Clock = 10 MHz (100 ns time period)

Result –

- From 0 ns to 50 ns, there is no output in on the keystream for security and initialization.
- From 50 ns, keystream starts giving output as : 47928e6b7b91f1845e2c55a151f76c50, 0e6bfb917184de2c55a1d1f76c5
and so on. on.