Indian Institute of Information Technology Vadodara (Gandhinagar Campus)

Design Project Report-2021

on

Analysis of Grain and Atom Ciphers

Submitted By

Sagar Ved Bairwa , Yash Beniwal , Yash Jain
201951131 , 201951175 , 201951176

Under the supervision of
Dr. Dibyendu Roy

***Abstract*—In This paper, we have discussed design and implimentation of two light weight stream ciphers – Grain-V1 and Atom. The Results of the paper are following–**

1) **The design of Grain V1 targets hardware environments where gate count, power consumption and memory is very limited.It is based on two shift registers and a nonlinear output function. The cipher has the additional feature that the speed can be increased at the expense of extra hardware. The key size is 80 bits and no attack faster than exhaustive key search has been identified.**
2) **The stream cipher Atom that has an internal state of 159 bits and offers a security of 128 bits.The length of NFSR and LFSR are 90 bits and 69 bits, respectively. Atom uses two key filters simultaneously to thwart certain cryptanalytic attacks that have been recently reported against keystream generators.In additionally we found that, The design of Atom is one of the smallest stream ciphers that offers a very high security level.Atom is builds on the basic structure of the Grain family of stream ciphers.**

I. Introduction

One of the most important research directions in cryptography is the design and implementation of lightweight cryptographic algorithms. The use of low-power, resource-limited devices has exploded in the last two decades. The reduction of the state size reduces the power consumption of the cipher. So it becomes a very challanging work for the community.The design principle of these lightweight stream ciphers differs significantly from the design principle of the standard stream ciphers.

Grain is a stream cipher submitted to eSTREAM in 2004 by Martin Hell, Thomas Johansson and Willi Meier. The design of Grain v1 deals with less number of hardware required. The Cipher is very useful where gate count, power consuption and memory is very limited.

An RFID tag is a typical example of a product where the amount of memory and power is very limited. These are microchips capable of transmitting an identifying sequence upon a request from a reader. Forging an RFID tag can have devastating consequences if the tag is used e.g. in electronic payments and hence, there is a need for cryptographic primitives implemented in these tags. Today, a hardware implementation of e.g. AES on an RFID tag is not feasible due to the large number of gates needed. Grain is a stream cipher primitive that is designed to be very easy and small to implement in hardware.

It is based on two shift registers and a nonlinear output function. The cipher has the additional feature that the speed can be increased at the expense of extra hardware.The key size is 80 bits and the IV size is specified to be 64 bits. The cipher is designed such that no attack faster than exhaustive key search should be possible, hence the best attack should require a computational complexity not significantly lower than 2^{80}.

Grain cipher is designed on specific properties. we can understand that it is not possible to have a design that is perfect for our all purposes i.e., processors of all words lengths, all hardware applications, all memory constraints etc. Grain is designed to be very small in hardware, using less number of logic gates and maintain high security. we can use Grain in general application software, when high speed in software is required . Because of this it does make sense to compare Grain with other cipher.

Grain provides a higher security than several other well known ciphers intended to be used in hardware applications. Well known examples of such ciphers are E0 used in Bluetooth and A5/1 used in GSM.

In FSE 2015, Armknecht and Mikhalev however proposed the stream cipher Sprout with a Grain-like architecture, whose internal state was equal in size with its secret key and yet resistant against TMD attacks. Although Sprout had other weaknesses, it germinated a sequence of stream cipher designs like Lizard and Plantlet with short internal states. Both these designs have had cryptanalytic results reported against them. In this paper, we propose the stream cipher Atom that has an internal state of 159 bits and offers a security of 128 bits. Atom uses two key filters simultaneously to thwart certain cryptanalytic attacks that have been recently reported against keystream generators.

In addition, we found that our design is one of the smallest stream ciphers that offers this security level, and we prove in this paper that Atom resists all the attacks that have been proposed against stream ciphers so far in literature. On the face of it, Atom also builds on the basic structure of the Grain family of stream ciphers. However,

we try to prove that by including the additional key filter in the architecture of Atom we can make it immune to all cryptanalytic advances proposed against stream ciphers in recent cryptographic literature.

The paper is organized as follows. Section 2 provides a detailed discription of stream ciphers. Section 3 gives the Design of Grain V1 and The design and implimention of Atom is discussed in Section 4. In Section 5, concludes the paper.

II. Steam cipher

A stream cipher [1] is a symmetric key cipher in which the plaintext digits are combined with a pseudo-random cipher stream (key stream). In a stream cipher, each plaintext digit is individually encrypted with the corresponding digit of the key stream to produce one digit of the ciphertext stream. Since the encryption of each digit depends on the current state of the encryption, it is also known as state encryption. In practice, a digit is typically a bit and the combination operation is an exclusive or (XOR).

The pseudo-random key stream is typically generated serially from a random initial value using digital shift registers. The seed value is used as a cryptographic key to decrypt the ciphertext stream. Stream ciphers represent a different approach to symmetric ciphers than block ciphers. Block ciphers work with large blocks of digits with a fixed and immutable transformation. This distinction is not always clear cut: in some modes, a block cipher primitive is used in such a way that it functions effectively as a stream cipher. Stream ciphers typically run at a higher speed than block ciphers and have less hardware complexity. However, stream ciphers can be vulnerable to security breaches (see Stream cipher attacks); B. if the same start state (seed) is used twice.

A. Idea Behind the Stream Cipher

A surprisingly easy idea enters the picture: we can encrypt a message using a key by performing a basic XOR operation. imagine we have two parties, Alice and Bob. Alice is the sender & Bob is Receiver.

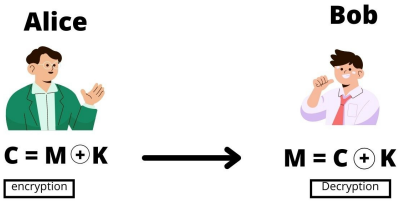

Fig. 1: Grain-V1 Basic Analogy

Basic encryption requires three main components:

1) A message, document or piece of data
2) A key
3) An encryption algorithm

The key typically used with a stream cipher is known as a one-time padding Algorithm. Mathematically, a one-time padding is unbreakable because it's always at least the exact same size as the message it is encrypting.

Here is an example to illustrate the one-timed padding process of stream ciphering: Person A attempts to encrypt a 10-bit message using a stream cipher. The one-time pad, in this case, would also be at least 10 bits long. This can become cumbersome depending on the size of the message or document they are attempting to encrypt.

M: 1001101001
K: 1000110111
C : Cipher Text

Encryption[C=M⊕K]
1001101001
⊕ 1000110111
0001011110

Decryption[M=C⊕K]
0001011110
⊕ 1000110111
1001101001

Fig. 2: Stream Cipher Demonstration

Security steps

1) len(K) $\geq$ len(M)
2) K can not be repeated to encrypting two different message.
3) K should be selected randomly.

III. Grain V1 Cipher

Martin Hell, Thomas Johansson, and Willi Meier submitted Grain [2] as a stream cipher to eSTREAM in 2004. The design of Grain v1 deals with less number of hardware required. The Cipher is very useful where gate count, power consuption and memory is very limited.

It is based on two shift registers and a nonlinear output function. The cipher has the additional feature that the speed can be increased at the expense of extra hardware.The key size is 80 bits and the IV size is specified to be 64 bits. The cipher is designed such that no attack faster than exhaustive key search should be possible, hence the best attack should require a computational complexity not significantly lower than 2^{80} .

Grain provides a higher security than several other well known ciphers intended to be used in hardware applications. Well known examples of such ciphers are E0 used in Bluetooth and A5/1 used in GSM.

A. Design Specification

The cipher consists of three main building blocks, namely an LFSR, an NFSR and an output function. The content of the LFSR is denoted by $s_i, s_{i+1}, ..., s_{i+79}$ and the content of the NFSR is denoted by $b_i, b_{i+1}, ..., b_{i+79}$. The feedback polynomial of the LFSR, $f(x)$ is a primitive polynomial of degree 80. It is defined as

$$f(x) = 1 + x^{18} + x^{29} + x^{42} + x^{57} + x^{67} + x^{80} \quad (1)$$

To remove any possible ambiguity we also define the update function of the LFSR as

$$s_{i+80} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i \quad (2)$$

The feedback polynomial of the NFSR is

$g(x) = 1 + x^{18} + x^{20} + x^{28} + x^{35} + x^{43} + x^{47} + x^{52} + x^{59} + x^{66} + x^{71} + x^{80} + x^{17}x^{20} + x^{43}x^{47} + x^{65}x^{71} + x^{20}x^{28}x^{35} + x^{47}x^{52}x^{59} + x^{17}x^{35}x^{52}x^{71} + x^{20}x^{28}x^{43}x^{47} + x^{17}x^{20}x^{59}x^{65} + x^{17}x^{20}x^{28}x^{35}x^{43} + x^{47}x^{52}x^{59}x^{65}x^{71} + x^{28}x^{35}x^{43}x^{52}x^{59}$

Again, to remove any possible ambiguity we also write the update function of the NFSR. $b_{i+80} = s_i + b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + b_{i+14} + b_{i+9} + b_i + b_{i+36}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + b_{i+63}b_{i+45}b_{i+28}b_{i+9} + b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21}$ The contents of the two shift registers represent the state

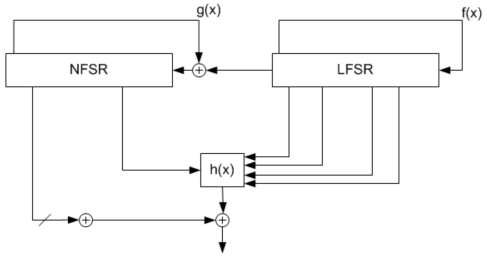

Fig. 3: The Cipher

of the cipher. h(x) is a filter function. This filter function is chosen to be balanced, correlation immune of the first order and has algebric degree 3. The nonlinearity is the highest possible for these functions. The input taken both from the LFSR and from the NFSR. The function is defined as

$h(x) = x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4 + x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4$

where the variables x_0 ,x_1 , x_2 , x_3 and$x_4$correspond to the tap positions s_{i+3}, s_{i+25} ,s_{i+46},$s_{i+64}$and b_{i+63}respectively. The output function is taken as

$$z_i = \sum_{k \in A} b_{i+k} + h(s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, s_{i+63}) \qquad (3)$$

where $A = \{1,2,4,10,31,43,56\}$.

B. Key Initialization

Before any kind of key stream is generated the cipher is initialised with the key and the IV. Let the bits of key is denoted by k and The bits of IV is denoted by IV. First load the NFSR with the key bits, $b_i = k_i$, for $0 \leq i \leq 79$, then load the 64 bits with LFSR with the IV, $s_i = IV_i$ for $0 \leq i \leq 63$. The Remaining bits of LFSR is filled with ones, $s_i = 1$ for $64 \leq i \leq 79$. Because of this the LFSR cannot be initialised to the all zero state. Then the cipher is clocked 160 times without producing any running key. Instead the output function is feedback and xored with the input, both to the LFSR and to the NFSR.

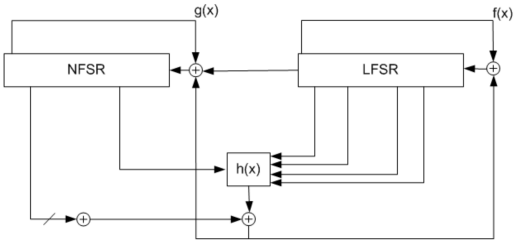

Fig. 4: The Key Intialization

C. Design Criteria

The main goal was to design an algorithm which is very simple to implement in hardware, requires only a small chip area, lower gate count , and limited memory also. The security requirements correspond to the computational complexity of 2^{80}, so it is necessary to build the cipher with 160-bit memory. in the making of grain cipher we have to maintaining its small size so we have to minimize the function that work together with memory. But they have enough in numbers also to maintains the security of a cipher. The LFSR in the cipher is of size 80 bits so it produces an output with period $2^{80} - 1$ with the probability of $\frac{2^{80}-1}{2^{80}} = 1 - 2^{-80}$ To make the NFSR state is balanced the input of NFSR is masked with the output of the LFSR.The filter function $h(x)$,is quite small with 5 variables and nonlinearity is equal to 12. If the cipher needs to be restart frequently with a new IV, initialization efficiency is a potential barrier.

D. Throughput rate:

Both shift registers namely LFSR and NFSR are regularly clocked so the cipher will output 1 bit/clock. It can be seen that the feedback shift registers NFSR and LFSR maximum shifts 16 bits per clock cycle [3] . Before initialization the LFSR contains the IV with 64 bits$\{iv_0, iv_1, ...iv_{63}.\}$ and 16 ones, $s_i = 1, 64 \leq i \leq 79$ as we seen in previous key intization part.For an initialization with two different IV , which differ by only one bit, the probability that a shift register bit is the same for both initializations should be close to 0.5.

To simplify this implementation, the last 15 bits of the shift registers, $s_i = 1$, $65 \leq i \leq 79$ are not used in the feedback functions or in the input of the filter function. This allows the speed to be easily are multiplied to 16 if enough hardware material is available. In the shift register when the speed is increased by a factor t each bit of shift registers is shifted t step instead of one.. Then the number of clockings used in the key initial-zation is $160/t$.

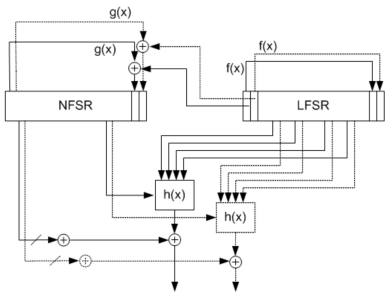

Fig. 5: The Cipher when speed is doubled.

E. Strength and limitations

Grain cipher is designed on specific properties. we can understand that it is not possible to have a design that is perfect for our all purposes i.e., processors of all words lengths, all hardware applications, all memory constraints etc. Grain is designed to be very small in hardware, using less number of logic gates and maintain high security. we can use Grain in general application software, when high speed in software is required . Because of this it does make

sense to compare Grain with other cipher.

The basic implementaion has rate 1 bit/clock cycle. The speed of a word oriented cipher is higher than a bit oriented cipher. Grain is a bit oriented due to the focus on small hardware complexity and this is compensated by the possibility to increase the speed at the cost of more hardware.

F. Code For Grain

As we have not tried any type of cryptanalysis previosly. so the implimentation of Grain is very Hard For Us. So Dr. Dibyoundu Roy Help us to understand the Implimentation of Grain V1. The provided code by Dr. Roy is Private and Not Shareable publically.

IV. Atom Cipher

Armknecht and Mikhalev [4] proposed the stream cipher Sprout with a Grain-like architecture, whose internal state was equal in size with its secret key and yet resistant against TMD attacks.Although Sprout had other weaknesses, it germinated a sequence of stream cipher designs like Lizard and Plantlet with short internal states.Both these designs have had cryptanalytic results reported against them. In this paper, we propose the stream cipher Atom that has an internal state of 159 bits and offers a security of 128 bits. Atom uses two key filters simultaneously to thwart certain cryptanalytic attacks that have been recently reported against keystream generators.

In addition, we found that the design is one of the smallest stream ciphers that offers this security level. Atom resists all the attacks that have been proposed against stream ciphers so far in literature. On the face of it, Atom also builds on the basic structure of the Grain family of stream ciphers.

A. Design Specification

Atom is composed of a linear feedback shift register (LFSR) and n nonlinear feedback shift register (NFSR). In Atom, the size of secret key k is 128 bits. Atom also have an initialization phase and a key stream phase.

1) Building Blocks: Atom's structure combined with LFSR and NFSR connected through a Xor gate. The length of NFSR and LFSR are 90 bits and 69 bits, respectively. At clock t = 0,1,..., denote the contents in NFSR and LFST by $B^t = (b^t_0, \ldots, b^t_{89})$ and $L^t = (l^t_0, \ldots, l^t_{68})$, respectively.

Output Function: The output function is a sum of linear terms, a quadratic bent function and another 9-variable function h.

$$O(B_t, L_t) = b^t_1 \oplus b^t_5 \oplus b^t_{11} \oplus b^t_{22} \oplus b^t_{36} \oplus b^t_{53} \oplus b^t_{72} \oplus b^t_{80} \oplus b^t_{84} \oplus l^t_5 l^t_{16} \oplus l^t_{13} l^t_{15} \oplus l^t_{30} l^t_{42} \oplus l^t_{22} l^t_{67} \oplus h(l^t_7, l^t_{33}, l^t_{38}, l^t_{50}, l^t_{59}, l^t_{62}, b^t_{85}, b^t_{41}, b^t_9)$$

During Initialization, the 69-bit LFSR is partitioned into 60 and 9 bits (6) with each part updated by its own update logic.During the Keystream generation phase, both these parts function as a single 69-bit LFSR.

Where $h(x) = h(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ is defined as [IV-A1] –

$$h(x) = x_0x_1x_2x_7x_8 \oplus x_0x_1x_2x_7 \oplus x_0x_1x_2x_8 \oplus x_0x_1x_2 \oplus x_0x_1x_3x_7x_8 \oplus x_0x_1x_3x_7 \oplus x_0x_1x_4x_7x_8 \oplus x_0x_1x_4x_7 \oplus x_0x_1x_4x_8 \oplus x_0x_1x_4 \oplus x_0x_1x_5x_7x_8 \oplus x_0x_1x_5x_7 \oplus x_0x_1x_6x_7x_8 \oplus x_0x_1x_6x_8 \oplus x_0x_1x_7x_8 \oplus x_0x_1x_8 \oplus x_0x_2x_3x_7x_8 \oplus x_0x_2x_3x_7 \oplus x_0x_2x_3x_8 \oplus x_0x_2x_3 \oplus x_0x_2x_4x_7x_8 \oplus x_0x_2x_4x_8 \oplus x_0x_2x_5x_7x_8 \oplus x_0x_2x_5x_7 \oplus x_0x_2x_5x_8 \oplus x_0x_2x_5 \oplus x_0x_2x_6x_7x_8 \oplus x_0x_2x_6x_8 \oplus x_0x_2x_7x_8 \oplus x_0x_2x_8 \oplus x_0x_3x_4x_7x_8 \oplus x_0x_3x_4x_7 \oplus x_0x_3x_5x_7x_8 \oplus x_0x_3x_5x_7 \oplus x_0x_3x_6x_7x_8 \oplus x_0x_3x_6x_7 \oplus x_0x_3x_8 \oplus x_0x_3 \oplus x_0x_4x_5x_7x_8 \oplus x_0x_4x_5x_7 \oplus x_0x_4x_6x_7x_8 \oplus x_0x_4x_6x_8 \oplus x_0x_4x_7 \oplus x_0x_4 \oplus x_0x_5x_6x_7x_8 \oplus x_0x_5x_6x_7 \oplus x_0x_5x_7x_8 \oplus x_0x_5x_7 \oplus x_0x_6x_7 \oplus x_0x_6x_8 \oplus x_0x_7x_8 \oplus x_1x_2x_3x_7x_8 \oplus x_1x_2x_4x_7x_8 \oplus x_1x_2x_4x_8 \oplus x_1x_2x_5x_7x_8 \oplus x_1x_2x_6x_7x_8 \oplus x_1x_2x_6x_8 \oplus x_1x_2x_7 \oplus x_1x_2x_8 \oplus x_1x_2 \oplus x_1x_3x_4x_7x_8 \oplus x_1x_3x_5x_7x_8 \oplus x_1x_3x_6x_7x_8 \oplus x_1x_3x_7 \oplus x_1x_4x_5x_7x_8 \oplus x_1x_4x_5x_8 \oplus x_1x_4x_6x_7x_8 \oplus x_1x_4x_7 \oplus x_1x_4 \oplus x_1x_5x_6x_7x_8 \oplus x_1x_5x_6x_7 \oplus x_1x_5x_7x_8 \oplus x_1x_5x_7 \oplus x_1x_5x_8 \oplus x_1x_6x_7 \oplus x_1x_8 \oplus x_1 \oplus x_2x_3x_4x_7x_8 \oplus x_2x_3x_5x_7x_8 \oplus x_2x_3x_6x_7x_8 \oplus x_2x_4x_5x_7x_8 \oplus x_2x_4x_5x_8 \oplus x_2x_4x_6x_7x_8 \oplus x_2x_4x_7x_8 \oplus x_2x_4x_8 \oplus x_2x_5x_6x_7x_8 \oplus x_2x_5x_6x_8 \oplus x_2x_5x_8 \oplus x_2x_6x_7x_8 \oplus x_2x_6x_8 \oplus x_2x_7x_8 \oplus x_2 \oplus x_3x_4x_5x_7x_8 \oplus x_3x_4x_5x_7 \oplus x_3x_4x_6x_7x_8 \oplus x_3x_4x_6x_7 \oplus x_3x_5x_6x_7x_8 \oplus x_3x_5x_7x_8 \oplus x_3x_6x_7x_8 \oplus x_3x_6x_7 \oplus x_3x_7 \oplus x_3 \oplus x_4x_5x_6x_7x_8 \oplus x_4x_5x_6x_8 \oplus x_4x_6x_7x_8 \oplus x_4x_6x_8 \oplus x_4x_7 \oplus x_5x_7x_8 \oplus x_5 \oplus x_6 \oplus x_7x_8 \oplus x_7 \oplus x_8 \oplus 1$$

Note:- h is a (9,5,3,240) function, where 9 Variable, 5 Algebraic degree, 3 Correlation Immunity, 240 non-linearty. It has one of highest non-linearities among all 9 variable Boolean Function. Bent Function are known to have highest non-linearity for even variable function and provide adequate protection.

2) NFSR: The defination of the update function $G(B^t)$ used in NFSR is specified as follows.

$$G(B^t) = b^t_0 \oplus b^t_{24} \oplus b^t_{49} \oplus b^t_{79} \oplus b^t_{84} \oplus b^t_3 b^t_{59} \oplus b^t_{10} b^t_{12} \oplus b^t_{15} b^t_{16} \oplus b^t_{25} b^t_{53} \oplus b^t_{35} b^t_{42} \oplus b^t_{55} b^t_{58} \oplus b^t_{60} b^t_{74} \oplus b^t_{20} b^t_{22} b^t_{23} \oplus b^t_{62} b^t_{68} b^t_{72} \oplus b^t_{77} b^t_{80} b^t_{81} b^t_{83}$$

3) LFSR: The update function $F(L^t)$ used in LFSR is defined as below: it is based on a primitive polynomial over GF(2) and hence always ensures a period of $2^{69} - 1$.

$$F(L^t) = l^t_0 \oplus l^t_5 \oplus l^t_{12} \oplus l^t_{22} \oplus l^t_{37} \oplus l^t_{45} \oplus l^t_{58} \tag{4}$$

B. Initialization Phase

Denote the 128-bit secret key by $k = (k_0, \ldots, k_{127})$ and the 128-bit initial value by $IV = (IV_0, \ldots, IV_{127})$. In addition, there will be extra 22-bit padding denoted by $PD = (PD_0, \ldots, PD_{21}) = 1^{22}$(all one string). The two registers are initialized based on Equations (5) and (6), respectively.

$$b^0_i = IV_i \quad (0 \le i \le 89) \tag{5}$$

$$l^0_i = \begin{cases} IV_{i+90} & (0 \le i \le 37) \\ PD_{i-38} & (38 \le i \le 59) \\ 0 & (60 \le i \le 68) \end{cases} \tag{6}$$

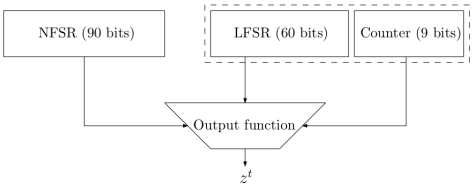

Fig. 6: The LFSR is divided into two part.

After the two registers are initialized, the state at clock t $(0 \leq t \leq 510)$ is updated as follows:

$$\begin{aligned} z^t &= O(B_t, L_t), \\ cnt &= l^t_{62}||l^t_{63}||l^t_{64}||l^t_{65}||l^t_{66}||l^t_{67}||l^t_{68}, \\ b^{t+1}_{89} &= G(B^t) \oplus l^t_0 \oplus k_{cnt} \oplus z^t \\ b^{t+1}_i &= b^t_{i+1} \ (0 \leq i \leq 88) \\ l^{t+1}_{59} &= F(L^t) \oplus z^t \\ l^{t+1}_i &= l^t_{i+1} \ (0 \leq i \leq 58) \\ l^{t+1}_{i+60} &= ((t+1) >> (8-i) \wedge 1) \ (0 \leq i \leq 8) \end{aligned}$$

In the initialization phase, LFSR's 60-bit part is updated using the LFSR logic and the 9 bit part operates as a decimal up- counter. The last 7 bit when interpreted as a decimal no. (cnt) alsoform the index of the key bit used in the NFSR update.

$$l^{t+1}_{i+60} = ((t+1) >> (8-i)) \text{ where } i \in [0,8]$$

$(t+1)^{th}$ cycle the LFSR bit 60 to 68 are updated with the 8 bits of decimal up- counter t+1. The 60 th LFSR location gets the MSB of the counter and 68 th location gets the LSB.

C. Key Generation

After the initialization phase,Illustration of Initialization phase NFSR$B^{511} = (b^{511}_0, ...b^{511}_{89})$ and LFSR $L^{511} = (l^{511}_0, ..., l^{68}_{511})$. The initialization phase had 511 rounds, the last 9 bits of the LFST at the beginning of the keystream generation phase will always be the all 1 vector. The first keystream used for plaintext encryption is z^{511}.

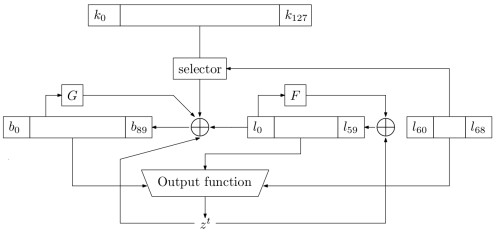

Fig. 7: Illustration of the initialization phase.

Note :- amount of keystream extractable from a single key-IV pair to 2 64 bits.

$$\begin{aligned} z^t &= O(B_t, L_t), \\ cnt &= l^t_{62}||l^t_{63}||l^t_{64}||l^t_{65}||l^t_{66}||l^t_{67}||l^t_{68}, \\ b^{t+1}_{89} &= G(B^t) \oplus l^t_0 \oplus k_{cnt} \oplus k_{t\%128} \\ b^{t+1}_i &= b^t_{i+1} \ (0 \leq i \leq 88) \\ l^{t+1}_{59} &= F(L^t) \oplus z^t \\ l^{t+1}_i &= l^t_{i+1} \ (0 \leq i \leq 67) \end{aligned}$$

The cnt sequence is derived from interpreting the last 7 bits of the LFSR as a decimal number. During the initialization phase, it is simply the i mod 128 sequence with I ranging from 0 to 510. The LFSR has a period of $2^{69}-1$, so does the cnt sequence. This garbles the order of key bits. it also provides immunity.

D. Code For Atom

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ROUND 510
#define lOOP 200
#define ull unsigned long long
#define ALLONE (0xffffffffffffffffUll)

ull b[90],l[69],c[9],z[lOOP];
ull KEY[128],IV[128];

//Generate 64 bit random string
ull myrand64(){
    ull temp = (ull) 0;
    int i;
    for (i = 0; i < 64; ++i) {
        temp = ((temp << (ull) 1) | !(drand48() < 0.5));

    }
    return temp;
}

void keyload(){

 for (int i = 0; i < 128; i++){
  KEY[i] = myrand64();
          IV[i] = myrand64();
 }
 for (int i = 0; i < 90; i++) b[i] = IV[i];

     for (int i = 0; i < 38; i++) l[i] = IV[i+90];
```

```
32
33      for (int i = 38; i < 60; i++) l[i] = ALLONE;
34
35  }
36  //filter function
37  ull h(ull x0, ull x1, ull x2, ull x3, ull x4, ull x5, ull x6, ull x7,ull x8){
38
39   ull a = (x0& x1& x2& x7& x8) ^ (x0& x1& x2& x7) ^ (x0& x1& x2& x8) ^ (x0& x1& x2)
40    ^ (x0& x1& x3& x7& x8) ^ (x0& x1& x3& x7)^(x0& x1& x4& x7& x8)^(x0& x1& x4& x7)
41    ^(x0& x1& x4& x8)^(x0& x1& x4)^(x0& x1& x5& x7& x8)^(x0& x1& x5& x7)^(x0& x1& x6& x7& x8)
42    ^(x0& x1& x6& x8)^(x0& x1& x7& x8)^(x0& x1& x8)^(x0& x2& x3& x7& x8)^(x0& x2& x3& x7)
43    ^(x0& x2& x3& x8)^(x0& x2& x3)^(x0& x2& x4& x7& x8)^(x0& x2& x4& x8)^(x0& x2& x5& x7& x8)
44    ^(x0& x2& x5& x7)^(x0& x2& x5& x8)^(x0& x2& x5)^(x0& x2& x6& x7& x8)^(x0& x2& x6& x8)
45    ^(x0& x2& x7& x8)^(x0& x2& x8)^(x0& x3& x4& x7& x8)^(x0& x3& x4& x7)^(x0& x3& x5& x7& x8)
46    ^(x0& x3& x5& x7)^(x0& x3& x6& x7& x8)^(x0& x3& x6& x7)^(x0& x3& x8)^(x0& x3)
47    ^(x0& x4& x5& x7& x8)^(x0& x4& x5& x7)^(x0& x4& x6& x7& x8)^(x0& x4& x6& x8)
48    ^(x0& x4& x7)^(x0& x4)^(x0& x5& x6& x7& x8)^(x0& x5& x6& x7)^(x0& x5& x7& x8)^(x0& x5& x7)
49    ^(x0& x6& x7)^(x0& x6& x8)^(x0& x7& x8)^(x1& x2& x3& x7& x8)^(x1& x2& x4& x7& x8)
50    ^(x1& x2& x4& x8)^(x1& x2& x5& x7& x8)^(x1& x2& x6& x7& x8)^(x1& x2& x6& x8)^(x1& x2& x7)
51    ^(x1& x2& x8)^(x1& x2)^(x1& x3& x4& x7& x8)^(x1& x3& x5& x7& x8)^(x1& x3& x6& x7& x8)
52    ^(x1& x3& x7)^(x1& x4& x5& x7& x8)^(x1& x4& x5& x8)^(x1& x4& x6& x7& x8)^(x1& x4& x7)
53    ^(x1& x4)^(x1& x5& x6& x7& x8)^(x1& x5& x6& x7)^(x1& x5& x7& x8)^(x1& x5& x7)^(x1& x5& x8)
54    ^(x1& x6& x7)^(x1& x8)^(x1)^(x2& x3& x4& x7& x8)^(x2& x3& x5& x7& x8)^(x2& x3& x6& x7& x8)
55    ^(x2& x4& x5& x7& x8)^(x2& x4& x5& x8)^(x2& x4& x6& x7& x8)^(x2& x4& x7& x8)^(x2& x4& x8)
56    ^(x2& x5& x6& x7& x8)^(x2& x5& x6& x8)^(x2& x5& x8)^(x2& x6& x7& x8)^(x2& x6& x8)
57    ^(x2& x7& x8)^(x2)^(x3& x4& x5& x7& x8)^(x3& x4& x5& x7)^(x3& x4& x6&x7&x8)^(x3&x4&x6&x7)
58    ^(x3& x5& x6& x7& x8)^(x3& x5& x7& x8)^(x3& x6& x7& x8)^(x3& x6& x7)^(x3& x7)^(x3)
59    ^(x4& x5& x6& x7& x8)^(x4& x5& x6& x8)^(x4& x6& x7& x8)^(x4& x6& x8)^(x4& x7)^(x5& x7& x8)
60    ^(x5)^(x6)^(x7& x8)^(x7)^(x8)^(1);
61   return a;
62  }
63  //intialization
64  void intialization(){
65   int i,cnt=0;
66   ull f=0,g=0,z=0;
67
68   for(i=0;i<=ROUND;i++){
69    for (int j = 0; j < 9; j++) {
70                c[8-j] = (i >> j) & 1;
71                l[60 + j] = c[j];
72             }
73           cnt = l[68]+ 2* l[67] + 4* l[66] + 8* l[65] + 16* l[64] + 32* l[63] + 64* l[62];
74
75           //printf("%d",cnt);
76
77           f = l[0] ^ l[5] ^ l[12] ^ l[22] ^ l[28] ^ l[37] ^ l[45] ^ l[58];
78
79            g = b[0]^b[24]^b[49]^b[79]^b[84]^(b[3]&b[59])^(b[10]&b[12])^(b[15]&b[16])
80                ^(b[25]&b[53])^(b[35]&b[42])^(b[55]&b[58])^(b[60]&b[74])^(b[20]&b[22]&b[23])
81                ^ (b[62]&b[68]&b[72])^(b[77]&b[80]&b[81]&b[83])^l[0]^KEY[i % 128];
82
83            z = b[1]^b[5]^b[11]^b[22]^b[36]^b[53]^b[72]^b[80]^b[84]^(l[5] & l[16])
84                 ^ (l[13]&l[15])^(l[30]&l[42])^(c[7]&l[22])^h(l[7],l[33],l[38],l[50],
85                  l[59],c[2],b[85],b[41],b[9]);
86
87            for (int j = 0; j < 59; j++) l[j] = l[j + 1];
88
89            l[59] = f ^ z;
90
91            for (int j = 0; j < 89; j++) b[j] = b[j + 1];
92
93            b[89] = g ^ z ^ l[0]^ KEY[cnt];
94
95   i=511;
96       for (int j = 0; j < 9; j++){
97       c[8-j] = (i >> j) & 1;
98    l[60 + j] = c[j];
99   }
100
101      }
102 }
103 ull keystream() {
104     ull f = 0, g = 0, z = 0;
105     int cnt = 0;
```

```
    for (int i = 0; i < 200; i++) {

        cnt = l[68] + 2* l[67] + 4* l[66] + 8* l[65] + 16* l[64] + 32* l[63] + 64* l[62];

        f = l[0] ^ l[5] ^ l[12] ^ l[22] ^ l[28] ^ l[37] ^ l[45] ^ l[58];

        g=b[0]^b[24]^b[49]^b[79]^b[84]^(b[3]&b[59])^(b[10]&b[12])^(b[15]&b[16])
          ^(b[25]&b[53])^(b[35]&b[42])^(b[55]&b[58])^(b[60]&b[74])^(b[20]&b[22]&b[23])
          ^(b[62]&b[68]&b[72])^(b[77]&b[80]&b[81]&b[83])^l[0]^KEY[cnt]^KEY[i%128];

        z=b[1]^b[5]^b[11]^b[22]^b[36]^b[53]^b[72]^b[80]^b[84]^(l[5]&l[16])
           ^(l[13]&l[15])^(l[30]&l[42])^(l[67]&l[22])^h(l[7],l[33],l[38],l[50],
            l[59],l[62],b[85],b[41],b[9]);

        for (int j = 0; j < 68; j++) l[j] = l[j + 1];
        l[68]=f;

        for (int j = 0; j < 89; j++) b[j] = b[j + 1];

        b[89] = g ;

        return z;
    }
}
int main(){

    srand48(time(NULL));
    keyload();
    intialization();
    for(int i=0;i<10;i++){
    ull z=keystream();
    printf("%llu\n",z);
    }

 }
```

Code 1: Atom implimentation in c

E. Code Explaination

The Code Of Atom is very difficult to impliment. we are thank full for Dr. Dibyondu Roy for Helping us to impliment this code.

V. Cube Finding Algoritham For Grain V1

Cube Finding Algoritham is an algorithm based on careful selection of Cube variables which can test the non-randomness of a cipher (or a Boolean function). The concept of Cube Finding Algoritham was first introduced by Aumasson et al. [5] in 2009. The main idea behind designing a Cube Finding Algoritham on a cipher (or a Boolean function) is to select the Cube variables in such a way that if there is any non-randomness in the cipher (or a Boolean function) the non-randomness must be reflected in the corresponding superpoly. By using this kind of testing procedure one can check several properties of a function such as the presence of any monomial in the function, presence of neutral variables in the function, upper bound of the degree of the function, and balancedness of the function, etc.

REFERENCES

[1] L. Jiao, Y. Hao, and D. Feng, "Stream cipher designs: a review," 2020.

[2] M. Hell, T. Johansson, and W. Meier, "Grain: a stream cipher for constrained environments," *International journal of wireless and mobile computing*, vol. 2, no. 1, pp. 86–93, 2007.

[3] M. Feldhofer, "Comparison of low-power implementations of trivium and grain," in *The State of the Art of Stream Ciphers, Workshop Record*, 2007, pp. 236–246.

[4] S. Banik, A. Caforio, T. Isobe, F. Liu, W. Meier, K. Sakamoto, and S. Sarkar, "Atom: A stream cipher with double key filter," *IACR Transactions on Symmetric Cryptology*, pp. 5–36, 2021.

[5] J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, "Cube testers and key recovery attacks on reduced-round md6 and trivium," in *Fast Software Encryption*, O. Dunkelman, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–22.