Indian Institute of Information Technology Vadodara (Gandhinagar Campus)

Design Project Report-2021

on

Implementation of Grain v1 and Atom for analysis

Submitted By

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

Under the supervision of
Dr. Dibyendu Roy

Supervisor's Signature

***Abstract*—In this project, we discussed the implementation of two lightweight stream ciphers – Grain-V1 and Atom. The Results of the analysis are following-**

1) **The design of Grain V1 targets hardware environments where gate count, power consumption, and memory are very limited. It is based on two shift registers(NFSR & LFSR), and a nonlinear output function. The cipher has the additional feature that the speed can be increased at the cost of extra hardware.**
2) **A new stream cipher, Atom has an internal state of 159 bits and offers security of 128 bits. The length of NFSR and LFSR are 90 bits and 69 bits, respectively. Atom uses two key filters simultaneously to oppose 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 built 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 [1]. The design of Grain V1 deals with less number of hardware required. The Cipher is very useful where gate count, power consumption and memory is very limited.

An RFID tag is an example of a product where the size of memory and power is finite [1]. These are microchips capable of transmitting an identifying sequence upon a request from a reader. For example - An RFID tag can have mortal consequences if the tag is used in electronic payments. Hence there is a need for cryptographic primitives implemented in these types of 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.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 [2] 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 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 fewer logic gates and maintaining high security [1]. 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 ciphers.

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 [2].

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 [3].

In this project, the stream cipher Atom has an internal state of 159 bits and offers the security of 128 bits. Atom uses two key filters simultaneously to oppose certain cryptanalytic attacks that have been recently reported against

keystream generators [3].

Atom is one of the smallest stream ciphers that offer a very high-security level. Atom resists all the attacks that have been proposed against stream ciphers so far in the literature. Atom also builds on the basic structure of the Grain family of stream ciphers [3]. By including the additional key filter in the architecture of Atom that makes it immune to all cryptanalytic advances proposed against stream ciphers in recent cryptographic attacks.

The project report is organized as follows. Section 2 provides a Literature survey of the prject. Section 3 shows present investigation about the project. section 4 shows Result and Discussion . Section 5 concludes the analysis of the report and future work.

II. Literature Survey

The design project is started with a basic understanding of different types of ciphers. To find a basic understanding of ciphers we took the help of the book named "Cryptography Theory and Practice" which was written by Douglas R. Stinson and Maura B. Peterson. After it, we started our project with two very recent ciphers named Grain V1 and Atom. To understand Grain, we choose "Grain - A Stream Cipher for Constrained Environments" which was published in 2004 by Martin Hell, Thomas Johansson, and Willi Meier. This Paper helped us with the main component of designing a cipher, How cipher works etc. It also helped us to understand which steps are required to analyze a cipher. After getting the required information about Grain we switched to Atom. To Understand the structure of Atom we choose "Atom: A Stream Cipher with Double Key Filter" which was published by Subhadeep Banik, Andrea Caforio, Takanori Isobe, Fukang Liu, Willi Meier, Kosei Sakamoto, and Santanu Sarkar. The Structure of Atom is like Grain that becomes a little bit easier to understand.

III. The Present Investigation

A. Steam cipher

A stream cipher [4] is a symmetric key cipher [5] in which the plaintext digits are combined with a pseudorandom cipher stream (keystream). In a stream cipher, each plaintext digit is individually encrypted with the corresponding digit of the keystream 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 [5]. The seed value(start state) 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 if the same start state (seed) is used twice [5].

1) Idea Behind the Stream Cipher: A surprisingly easy idea enters the picture: we can encrypt a message using a key by performing an XOR operation. Assume we have two parties, Alice and Bob. Alice is the sender & Bob is Receiver.

Fig. 1: Stream Cipher 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 [5].

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 Huge depending on the size of the message or document they are attempting to encrypt.

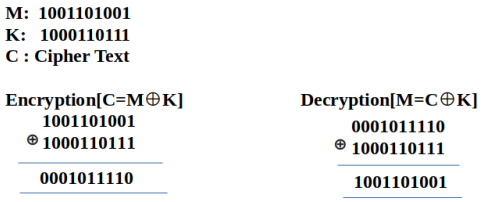

Fig. 2: Stream Cipher Demonstration

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.

B. Grain V1 Cipher

Martin Hell, Thomas Johansson, and Willi Meier submitted Grain [1] 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 consumption and memory is very limited.

Grain is a stream cipher primitive that is designed to be very easy and small to implement in 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 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 fewer logic gates and maintaining 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 ciphers.

Grain provides a higher security than several other well known ciphers intended to be used in hardware applications [2]. Well known examples of such ciphers are E0 used in Bluetooth and A5/1 used in GSM.

1) Design Specification: The cipher design consists of three main component, 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

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

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

$$s_{i+80} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i$$

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 uncertainty we also write the update function of the NFSR.

$$\begin{aligned} b_{i+80} = {} & s_i + b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + \\ & b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + b_{i+14} + \\ & b_{i+9} + b_i + b_{i+36}b_{i+60} + b_{i+37}b_{i+33} + \\ & b_{i+15}b_{i+9} + b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + \\ & b_{i+63}b_{i+45}b_{i+28}b_{i+9} + b_{i+60}b_{i+52}b_{i+37}b_{i+33} + \\ & b_{i+63}b_{i+60}b_{i+21}b_{i+15} + b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + \\ & b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + \\ & b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21} \end{aligned}$$

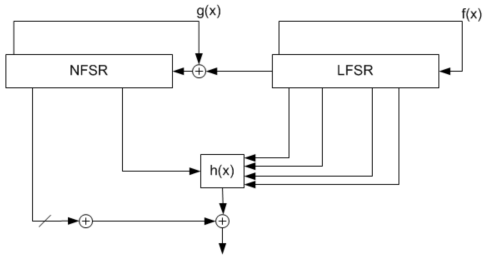

Fig. 3: The Cipher

The contents of the two shift registers represent the state of the cipher. h(x) is a filter function. This filter function has algebric order 3 and to be choosen to be balanced, correlation immune of the 1^{st} order. The non-linearity is the highest possible with these functions. The input is created by both LFSR and 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 andx_4 correspond to the top positions s_{i+3}, s_{i+25} , s_{i+46}, s_{i+64} and b_{i+63} respectively. The output function is taken as

$$z_i = \sum_{k \in A} b_{i+k} + h(s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, s_{i+63}) \quad (1)$$

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

2) Key Initialization: Before any kind of keystream is generated the cipher is initialized with the key and the IV. Let the bits of the key be denoted by k and The bits of IV are 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 initialized 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.

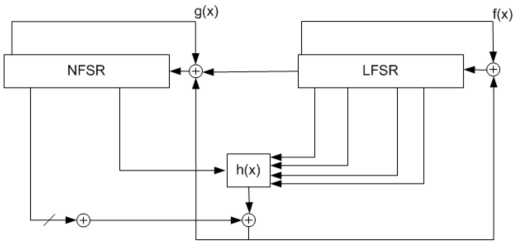

Fig. 4: The Key Intialization

3) 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 Nonlinear Feedback Shift Register 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.

4) 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 [6] . 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 $160/t$.

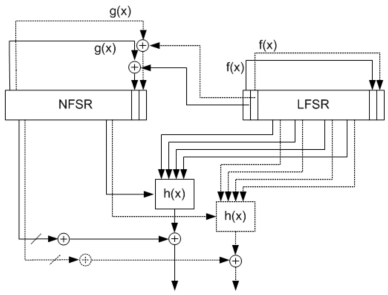

Fig. 5: The Cipher when speed is doubled.

5) 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.

C. Atom Cipher

Armknecht and Mikhalev [3] proposed the stream cipher "Sprout" with a Grain-like architecture, whose internal state was equal in size with its secret key but secure against TMD attacks. while Sprout had other weaknesses, it generated a sequence of stream cipher designs like "Lizard" and "Plantlet" with short internal states. Cryptanalytic results have been reported against both of these schemes.

In this report, Atom is a stream cipher with an internal state of 159 bits and security of 128 bits. Atom uses two key filters simultaneously to oppose certain cryptanalytic attacks that have been recently reported against keystream generators. The design of Atom is one of the smallest stream ciphers that offer this security level. Atom cipher resists all the attacks that have been proposed against stream ciphers so far. Atom also builds on the basic structure of the Grain family of stream ciphers.

1) Design Specification: Atom has two shift registers - linear feedback shift register (LFSR) nonlinear feedback shift register (NFSR) and the size of secret key k is 128 bits. Like most of the stream ciphers, Atom also has an initialization phase and a keystream phase.

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 $B^t = (b_0^t, \ldots, b_{89}^t)$ and $L^t = (l_0^t, \ldots, l_{68}^t)$, respectively.

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

$$O(B_t, L_t) = b_1^t \oplus b_5^t \oplus b_{11}^t \oplus b_{22}^t \oplus b_{36}^t \oplus b_{53}^t \oplus b_{72}^t \oplus b_{80}^t \oplus b_{84}^t \oplus l_5^t l_{16}^t \oplus l_{13}^t l_{15}^t \oplus l_{30}^t l_{42}^t \oplus l_{22}^t l_{67}^t \oplus h(l_7^t, l_{33}^t, l_{38}^t, l_{50}^t, l_{59}^t, l_{62}^t, b_{85}^t, b_{41}^t, b_9^t)$$

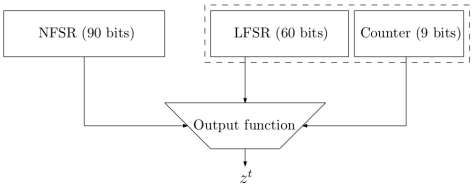

Fig. 6: The LFSR is divided into two part.

During Initialization, the 69-bit LFSR is divided into 60 and 9 bits (Figure 6) with each part updated by its own update logic. During the Keystream generation phase, both these parts works 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 –

$$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(x) has 9 Variable, 5 Algebraic degree, 3 Correlation Immunity, 240 non-linearty. Among the nine variable Boolean Functions, it has the highest non-linearity. Bent functions are recognized to have maximum non-linearity in even variable functions and also provide appropriate security.

NFSR: The Feedback function of $G(B^t)$ used in NFSR is defined as follows:

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

LFSR: The Feedback function $F(L^t)$ used in LFSR is specified as follow:

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

It is based on a primitive polynomial over GF(2) and hence always ensures a period of $2^{69}-1$ because the size of LFSR is 69bit so it has a period of 2^{69} and we can't consider all zero's as feedback so now it has a period of $2^{69}-1$.

2) Initialization Phase: $k = (k_0, k_1 ..., k_{127})$ denotes the 128-bit secret key, and $IV = (IV_0, IV_1 ..., IV_{127})$ denotes the 128-bit Intial Value. There will be an additional 22 bit padding, represented by $PD = (PD_0, PD_1, ..., PD_{21}) = 1^{22}$. All Paddings have one string as value. The two registers are initialized based on Equations (2) and (3), respectively.

$$b^0_i = IV_i \quad (0 \le i \le 89) \tag{2}$$

$$l^0_i = \begin{cases} IV_{i+90} & (0 \le i \le 37) \\ PD_{i-38} & (38 \le i \le 59) \\ 0 & (60 \le i \le 68) \end{cases} \tag{3}$$

After initialization of NFSR and LFSR , The State is updated for t $(0 \le t \le 510)$ as follows:

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

The 60-bit part of the LFSR is updated using the LFSR logic, whereas the 9-bit part is update as a decimal up-counter during the initialization phase. When interpreted as a decimal number (cnt), the last 7 bits form the index of the key bit utilized in the NFSR update.

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

The LFSR bits 60 to 68 are updated with the 8 bits of the decimal up-counter t+1 on the $(t+1)th$ cycle. The counter's MSB (Most Significant Bit) is assigned to the 60th LFSR location, while the LSB (Least Significant Bit) is assigned to the 68th.

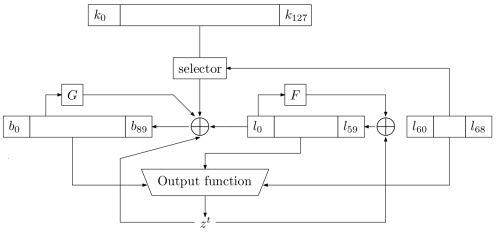

Fig. 7: Illustration of the initialization phase.

3) Key Generation: After the completion initialization phase, the NFSR state is $B^{511} = (b^{511}_0, ...b^{511}_{89})$ and the LFSR state is $L^{511} = (l^{511}_0, ..., l^{68}_{511})$. Total rounds in the initialization phase had 511 rounds. since at the beginning of the keystream generation phase, the last 9 bits of LFSR worked as a decimal up counter. and the last 9 bits of the LFSR at the keystream generation phase will always be all 1 vector.

we get the first keystream that is used for plain text encryption as an output function of t=511 and it is denoted as z^{511}. At the keystream phase, the state is clock $t \ge 511$ is updated as follows and the output will be keystream bit z^t.

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

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

As we know in the key initialization phase the cnt sequence is concatenated of the last 7 bits of the LFSR as a decimal number. During the initialization, The Additional key filter k counter is simply i mod 128. where i belong from 0 to 510. since the cnt sequence is fully dependent on LFSR so both have the same period of $2^{69}-1$. with the use of additional key filter $k_{t\%128}$ provides very high immunity against the cryptanalytic attack and also provides high period.

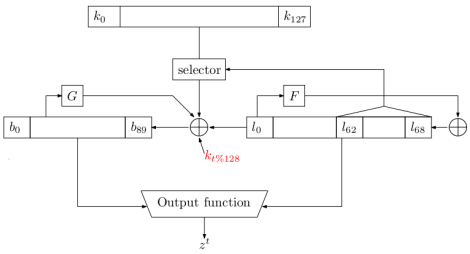

Fig. 8: Illustration of the keystream generation phase.

```
#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];

     for (int i = 38; i < 60; i++) l[i] = ALLONE;

}
//filter function
ull h(ull x0, ull x1, ull x2, ull x3, ull x4, ull x5, ull x6, ull x7,ull x8){

 ull a = (x0& x1& x2& x7& x8) ^ (x0& x1& x2& x7) ^ (x0& x1& x2& x8) ^ (x0& x1& x2)
 ^ (x0& x1& x3& x7& x8) ^ (x0& x1& x3& x7)^(x0& x1& x4& x7& x8)^(x0& x1& x4& x7)^
 (x0& x1& x4& x8)^(x0& x1& x4)^(x0& x1& x5& x7& x8)^(x0& x1& x5& x7)^(x0& x1& x6& x7& x8)
 ^(x0& x1& x6& x8)^(x0& x1& x7& x8)^(x0& x1& x8)^(x0& x2& x3& x7& x8)^(x0& x2& x3& x7)^
 (x0& x2& x3& x8)^(x0& x2& x3)^(x0& x2& x4& x7& x8)^(x0& x2& x4& x8)^(x0& x2& x5& x7& x8)
 ^(x0& x2& x5& x7)^(x0& x2& x5& x8)^(x0& x2& x5)^(x0& x2& x6& x7& x8)^(x0& x2& x6& x8)^
 (x0& x2& x7& x8)^(x0& x2& x8)^(x0& x3& x4& x7& x8)^(x0& x3& x4& x7)^(x0& x3& x5& x7& x8)
 ^(x0& x3& x5& x7)^(x0& x3& x6& x7& x8)^(x0& x3& x6& x7)^(x0& x3& x8)^(x0& x3)^
 (x0& x4& x5& x7& x8)^(x0& x4& x5& x7)^(x0& x4& x6& x7& x8)^(x0& x4& x6& x8)
 ^(x0& x4& x7)^(x0& x4)^(x0& x5& x6& x7& x8)^(x0& x5& x6& x7)^(x0& x5& x7& x8)^(x0& x5& x7)
 ^(x0& x6& x7)^(x0& x6& x8)^(x0& x7& x8)^(x1& x2& x3& x7& x8)^(x1& x2& x4& x7& x8)^
```

```
 (x1& x2& x4& x8)^(x1& x2& x5& x7& x8)^(x1& x2& x6& x7& x8)^(x1& x2& x6& x8)^(x1& x2& x7)
 ^(x1& x2& x8)^(x1& x2)^(x1& x3& x4& x7& x8)^(x1& x3& x5& x7& x8)^(x1& x3& x6& x7& x8)^
 (x1& x3& x7)^(x1& x4& x5& x7& x8)^(x1& x4& x5& x8)^(x1& x4& x6& x7& x8)^(x1& x4& x7)^
 (x1& x4)^(x1& x5& x6& x7& x8)^(x1& x5& x6& x7)^(x1& x5& x7& x8)^(x1& x5& x7)^(x1& x5& x8)
 ^(x1& x6& x7)^(x1& x8)^(x1)^(x2& x3& x4& x7& x8)^(x2& x3& x5& x7& x8)^(x2& x3& x6& x7& x8)
 ^(x2& x4& x5& x7& x8)^(x2& x4& x5& x8)^(x2& x4& x6& x7& x8)^(x2& x4& x7& x8)^(x2& x4& x8)
 ^(x2& x5& x6& x7& x8)^(x2& x5& x6& x8)^(x2& x5& x8)^(x2& x6& x7&x8)^(x2& x6&x8)^(x2& x7&x8)
 ^(x2)^(x3& x4& x5& x7& x8)^(x3& x4& x5& x7)^(x3& x4& x6& x7& x8)^(x3& x4& x6& x7)^
 (x3& x5& x6& x7& x8)^(x3& x5& x7& x8)^(x3& x6& x7& x8)^(x3& x6& x7)^(x3& x7)^(x3)^
 (x4& x5& x6& x7& x8)^(x4& x5& x6& x8)^(x4& x6& x7& x8)^(x4& x6& x8)^(x4& x7)^(x5& x7& x8)
 ^(x5)^(x6)^(x7& x8)^(x7)^(x8)^(1);
 return a;
}
//intialization
void intialization(){
 int i,cnt=0;
 ull f=0,g=0,z=0;

 for(i=0;i<=ROUND;i++){
  for (int j = 0; j < 9; j++) {
             c[8-j] = (i >> j) & 1;
             l[60 + j] = c[j];
          }
         cnt = l[68] + 2 *l[67] + 4 *l[66] + 8 *l[65] + 16 *l[64] + 32 *l[63] + 64 *l[62];

         //printf("%d",cnt);

         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[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]) ^ (c[7] & l[22])
                ^ h(l[7], l[33], l[38], l[50],l[59], c[2], b[85], b[41], b[9]);

          for (int j = 0; j < 59; j++) l[j] = l[j + 1];

          l[59] = f ^ z;

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

          b[89] = g ^ z ^ l[0]^ KEY[cnt];

 i=511;
     for (int j = 0; j < 9; j++){
     c[8-j] = (i >> j) & 1;
  l[60 + j] = c[j];
 }

     }
}
ull keystream() {
    ull f = 0, g = 0, z = 0;
    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]) ^ (c[7] & l[22])
            ^ h(l[7], l[33], l[38], l[50],l[59], c[2], 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();

    ull z = keystream();
    printf("%llu\n",z);

 }
```

Code 1: Atom implimentation in c

5) Code Explaination: To impliment Atom in c we have take help from some Github Repositaries and Grain V1 implimentation. To impliment the Atom cipher we define all the variables in unsigned long long. as we have discussed in the design part that l[69] is for LFSR bits and b[90] is for NFSR. further the LFSR is divided into two parts 60+9. ant the 9 bits are used to calculated for cnt. then KEY, IV are defined as size of 128.

Then we define a function for a random 64 bit generator. and this function returns a 64 bit random value.

Then we define a another function to load the All the keys. and their we assign the random numbers to KEY, IV.

Then we define a filter function according to the research paper. Then we define initialization function and assigned values as per the research paper. cnt is a concatanation of 7 bits. l[59] is assigned with $f \oplus z$ and b[89] is assigned with $g \oplus z \oplus l[0] \oplus KEY[cnt]$.

Then we define a keystream generation function as per the paper. and lastly we controll these functions with main function.A screenshot is attached with the report see [9] .

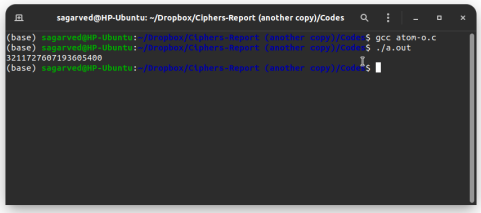

Fig. 9: Screenshot of Atom Implimentation.

IV. Result And Discussion

The project "Implementation of Grain v1 and Atom for analysis" has been understood and implemented. We understand cipher design, Specification, shift register, and how the cipher work in the Key initialization and keystream generation phase. After implementing of atom cipher code we get a 64-bit random keystream that use to encrypt plain text.

V. Conclusion and Future Work

In conclusion, this Project achieved a very detailed point of view on Designing and implementing Grain and Atom ciphers. We have tried two very recent cryptographic ciphers - Grain-V1 and Atom.

Grain has been introduced for very small hardware implementation that has very strong immunity against different cipher attacks. Atom also has Grain like structure and an additional key filter that seems to protect against most advanced cryptoanalytic attacks.

In the Future, we would like to perform very intense research on such types of ciphers with more resources and more knowledge. we definitely assume that with the proper knowledge about cryptographic attacks we can find more interesting results.

Acknowlegement

We are thankful to IIIT Vadodara for providing us with this valuable opportunity to learn different new things and All the Necessary Resources for the project.

All in all, we would like to thank everyone involved in this project and helped us with their suggestions to make this project better.

References

[1] 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.

[2] "Grain (cipher) - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Grain_(cipher)

[3] 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*, vol. 2021, pp. 5–36, 3 2021. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/8832

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

[5] "what is a stream cipher?" [Online]. Available: https://searchsecurity.techtarget.com/definition/stream-cipher

[6] 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.