[CS304] Introduction to Cryptography and Network Security

Course Instructor: Dr. Dibyendu Roy — Winter 2022-2023
Scribed by: Chitranshi Srivastava (202051055) — Lecture 15, 16, 17 and 18(Week 8 and 9)

1 One Time Padding

As we have discussed earlier, so we know encryption and decryption in OTP are given as:
Encryption: $C = M \oplus K$
Decryption: $M = C \oplus K$
We have already discussed that OTP provides perfect secrecy under the following conditions:

1. The secret key K cannot be used to encrypt two messages, that is, the key can not be reused.
2. The length of key must be greater than or equal to the length of message.
3. Key K is uniformly selected from the key space.

However, OTP is not practical because if we have a method to share the secret key (length equal to the message), then we can share the message using that mechanism. Here, we want to design an algorithm as efficient as OTP but which can be used for practical purposes. Also, it may not provide perfect secrecy, but it is not possible to prove or disprove that it provides perfect secrecy.
Let us a define a function F as:

$$F(K, IV) = Z_i \text{ where } Z_i \in \{0, 1\}$$

K is secret key, IV is initialization vector (a public parameter). Function F produces n bits $Z_0, Z_1, ..., Z_{n-1}$. The function F is efficiently computable.
Plaintext: $m_0, m_1, ..., m_{n-1}$
Z: $Z_0, Z_1, ..., Z_{n-1}$
Encryption: $C_0 = m_0 \oplus Z_0, C_1 = m_1 \oplus Z_1, ..., C_{n-1} = m_{n-1} \oplus Z_{n-1}$

The function F, known as Pseudo Random Bit Generation Function has certain properties. These properties are listed below:

1. The output i.e. $Z_0, Z_1, ..., Z_{n-1}$ is a random looking string. Let us try to understand the meaning of random looking string. Consider an unbiased coin, toss it for n times, output 0 if head comes, and 1 if tails comes up. Let's say the string generated by coin tossing is $x_0, x_1, ..., x_{n-1}$. Now if you are given the string $Z_0, Z_1, ..., Z_{n-1}$, you cannot distinguish, whether the string is obtained from a coin tossing experiment or by a pseudo random bit generator with key and IV as input. However, the string $Z_0, Z_1, ..., Z_{n-1}$ can be reproduced by passing the same inputs to the function F. This means a string that seems to be random but can be reproduced again by passing same inputs to F is known as random looking string.
2. If we give same input (K, IV) to F in different time, it will generate same Z_i.

$$\begin{tikzcd}
\underline{A} & & \underline{B} \\
K & & K \\
F(K,IV) = Z_i & & F(K,IV) = Z_i \\
C_i = m_i \oplus Z_i \arrow[rr, "{C_i, IV}"] & & C_i \oplus Z_i = m_i
\end{tikzcd}$$

3. If key K is selected randomly and is kept secret, then the outputs $Z_0, Z_1, ..., Z_{n-1}$ will be indistinguishable from bit string generated by using a random bit generator (coin tossing).

$$F(K, IV) \rightarrow Z_0, Z_1, ..., Z_{n-1} \text{ and Coin Tossing} \rightarrow x_0, x_1, ..., x_{n-1} \text{ are indistinguishable.}$$

4. The length of the output bits $Z_0, Z_1, ..., Z_{n-1}$, that is, n is very much larger than the length of K. There are efficient functions that can produce $2^{80} - 1$ length output for a key of length 80 bits. Using this property, a small key can encrypt very large messages. However, if there is repetition in the output bits, then difference can be calculated between corresponding message bits just as in OTP. Block ciphers like AES and DES will take a very long time to encrypt a message of $2^{80} - 1$ bits. DES will perform around 2^{64} encryption to encrypt the whole message. Hence, this technique is much more efficient.

5. If we modify at least one bit of K or of IV, then there will be unpredictable change in output of Z_i.

$$F(K, IV_1) = Z_i^{(1)} \ 0 \le i \le n-1$$
$$F(K, IV_2) = Z_i^{(2)} \ 0 \le i \le n-1$$

   $Z_i^{(1)}$ and $Z_i^{(2)}$ are uncorrelated. This property can be used to encrypt two different messages using same key. We can use the same key but change the IV to get different Z_i.

$$\begin{tikzcd}
\underline{A} & & \underline{B} \\
K & & K \\
F(K,IV_1) = Z_i^{(1)} & & F(K,IV_1) = Z_i^{(1)} \\
C_i^{(1)} = m_i^{(1)} \oplus Z_i^{(1)} \arrow[rr, "{C_i^{(1)}, IV_1}"] & & C_i^{(1)} \oplus Z_i^{(1)} = m_i^{(1)} \\
F(K,IV_2) = Z_i^{(2)} & & F(K,IV_2) = Z_i^{(2)} \\
C_i^{(2)} = m_i^{(2)} \oplus Z_i^{(2)} \arrow[rr, "{C_i^{(2)}, IV_2}"] & & C_i^{(2)} \oplus Z_i^{(2)} = m_i^{(2)}
\end{tikzcd}$$

Hence, the advantages of this technique are that very long messages can be encrypted using a small key and the same key can be used to encrypt different messages.

2 Stream Cipher

A stream cipher consists of two things, a pseudo random bit generator and an encryption and decryption technique. The PBGR takes few inputs and generates keystream bits Z_i.

PBGR: $F(K, ...) = Z_i$ (keystream bits)
Encryption: $C_i = m_i \oplus Z_i$
Decryption: $m_i = C_i \oplus Z_i$

We are using XOR here as as encryption technique, however it can be any other efficient computation too.

2.1 Synchronous Stream Cipher

A synchronous stream cipher is one in which the key stream is generated independently of plaintext and ciphertext bits. It has the following functions:

- State Update function $\implies S_{i+1} = f(S_i, K)$
- Keystream Generator function $\implies Z_i = g(S_i, K)$
- Ciphertext Generation Function $\implies C_i = h(Z_i, m_i)$

Here S_0 is initial state and may be determined from K and IV. In synchronous stream cipher the functions f and g does not take plaintext and ciphertext input.

2.2 Asynchronous or Self-Synchronizing Stream Ciphers

An asynchronous stream cipher is one in which the keystream bits are generated as a function of key and a fixed number of previous cipher text bits.

- State Update Function: $\sigma_{i+1} = f(\sigma, K, IV)$ where $\sigma = (C_{i-t}, C_{i-t+1}, ..., C_{i-1})$
- Keystream Generation Function: $Z_i = g(\sigma_i, K)$
- Ciphertext Generation Function: $C_i = h(Z_i, m_i)$

Here, $\sigma_0 = (C_{-t}, C_{-t+1}, ..., C_{-1})$ is the non-secret initial state.

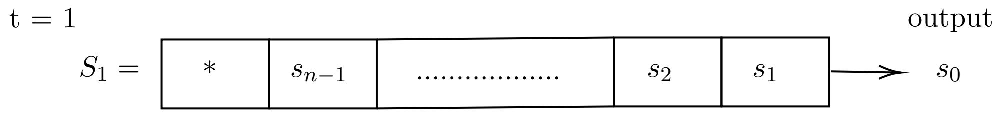
2.3 Linear Feedback Shift Register

This is one of the most used stream cipher and has been used for every communication (voice calls) till 4G. It contains a n-bit register. The states are denoted by S, and the bits are denoted by s. There is a clock and at each clocking number the state of the register updates and the, an output (keystream bit) is generated using which encryption of message can be done.
A register of length n means that it is a n-bit LFSR, or equivalently it has a state of length n.
Let us say at clocking number t = 0, the state of the register is S_0. For each $0 \leq i \leq n-1$, $s_i \in \{0, 1\}$.

t = 0, t $\rightarrow$ clocking number

$$S_0 = \begin{array}{|c|c|c|c|c|}\hline s_{n-1} & s_{n-2} & .... & s_1 & s_0 \\ \hline\end{array}$$

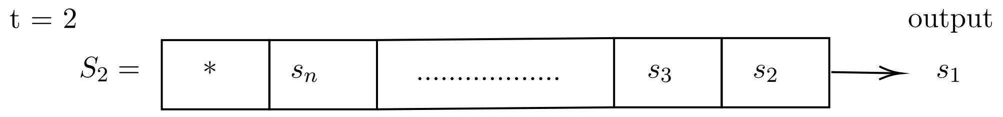
At each clocking number, a shift by one bit takes place (either right or left, depends on the design). Here, we will take right shift for understanding.



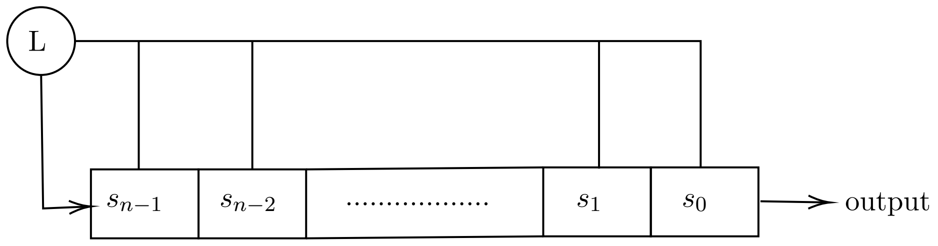
The rightmost bit s_0 moves out and is the output (the keystream bit). However, the leftmost bit s_n becomes empty. s_n is known as feedback bit and is calculated as:

$$s_n = L(s_0, s_1, ..., s_{n-1}) = L(S_0)$$

After one more clocking, the state can be represented as:



Again, the feedback bit $S_{n+1} = L(S_1)$. We will look at the function L in some time. The LFSR with right shift operation can be represented with the following circuit.



LFSR with Right Shift

The function L is a linear function on the bits of the previous state.

$$L : \{0,1\}^n \rightarrow \{0,1\}$$
$$L(s_0, s_1, ..., s_{n-1}) = s_n$$

A linear function can be represented as:

$$L_a = a_0 \cdot s_0 \oplus a_1 \cdot s_1 \oplus ..... \oplus a_{n-1} \cdot s_{n-1} \text{ where } a_i \in \{0,1\}$$

Suppose, an arbitrary function L be defined as:

$$L = a_0 \cdot s_0 \oplus a_1 \cdot s_1 \oplus ..... \oplus a_{n-1} \cdot s_{n-1} \oplus a_n \text{ where } a_i \in \{0,1\}$$

In this function, if $a_n = 0$ then $L = L_a$, a linear function. Otherwise, if $a_n = 1$ the $L \neq L_a$. In fact, such a function is known as Affine function.

A function can be proved to be linear using the following property.

$$L(X) \oplus L(Y) = L(X \oplus Y)$$
$$\implies L(X) \oplus L(Y) \oplus L(X \oplus Y) = 0$$

Example: Find if the following functions are linear or not. Solve it considering 2-bit inputs.

1. $L_1(x, y) = x \oplus y$

2. $L_2(x, y) = 1 \oplus x \oplus y$

Solution:

1. Let's compute $L_1(x) \oplus L_1(y) \oplus L_1(x \oplus y)$

$$L_1(x) \oplus L_1(y) \oplus L_1(x \oplus y) = (x_1 \oplus x_2) \oplus (y_1 \oplus y_2) \oplus ((x_1 \oplus y_1) \oplus (x_2 \oplus y_2))$$
$$L_1(x) \oplus L_1(y) \oplus L_1(x \oplus y) = 0$$

   Therefore, L_1 is a linear function.

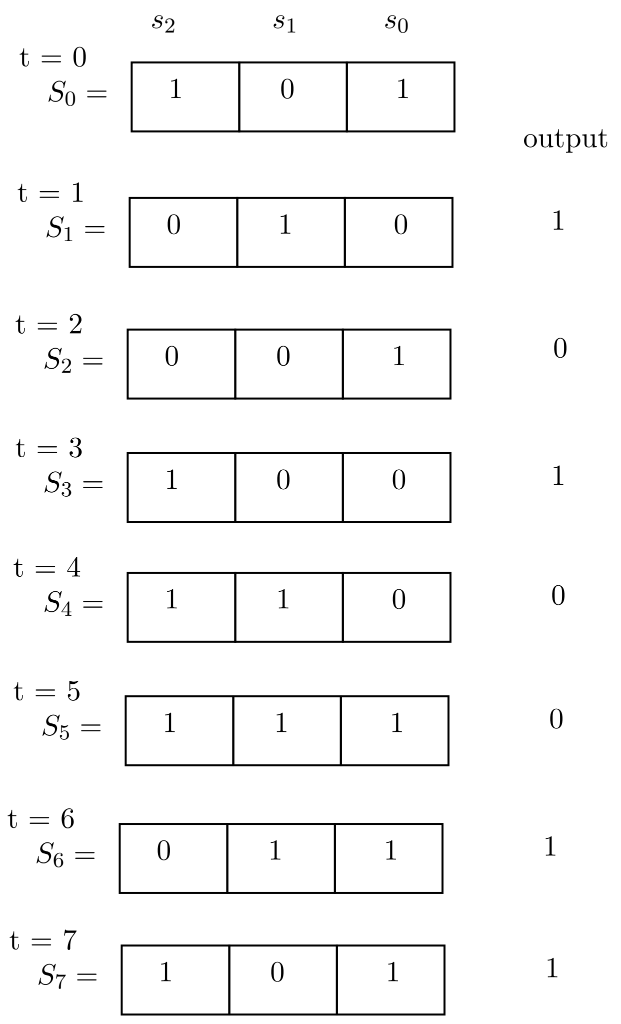
2. $L_2(x) \oplus L_2(y) \oplus L_2(x \oplus y) = (1 \oplus x_1 \oplus x_2) \oplus (1 \oplus y_1 \oplus y_2) \oplus (1 \oplus (x_1 \oplus y_1) \oplus (x_2 \oplus y_2))$

$$L_2(x) \oplus L_2(y) \oplus L_2(x \oplus y) = 1$$

   Therefore, L_2 is not a linear function.

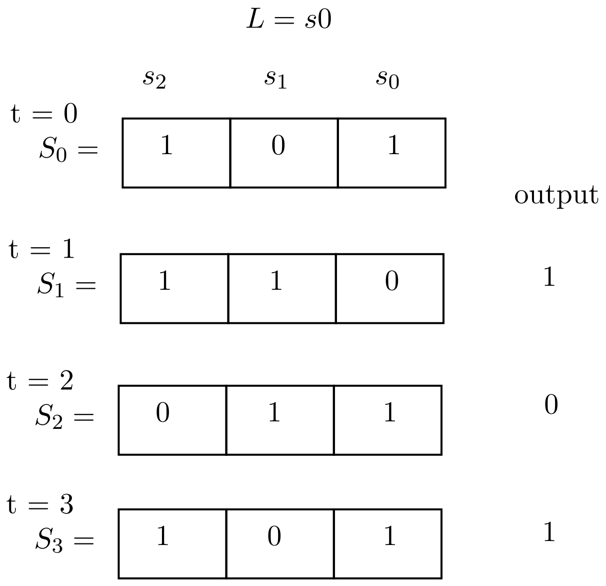
LFSR has a linear function, whose output depends on previous state, therefore it provides feedback. There is shift operation on the bits stored in register. Hence, the name is Linear Feedback Shift Register.
Now, let us see an example of a 3-bit LFSR.

$$L = s_0 \oplus s_2$$



The L function for this LFSR is $L = s_0 \oplus s_2$.
In any LFSR, if you have reached the initial state again, the the output bits will be repeated. In the above example it can be seen that from start state t = 0 to the state t = 7, all the non-zero states are obtained.
The output bits in the above example are repeated after $t = 2^3 - 1$ states because at t = 7, the initial state is reached. Hence, maximum output length that can be achieved in this LFSR without repetition is 7. Hence, using LFSR, only $2^n - 1$ maximum number of non-zero states can be generated.
Also, if we take the 0 state i.e. all bits in the register are 0, it will remain in the zero state forever. Hence, in any LFSR, if input state is 0, then it will remain zero. Hence, LFSR has a fixed point (0-state).

Let us look at another example of a 3-bit LFSR with a different L function. This time the L function is $L = s_0$.



Here, we can see the initial state is reached again only at t = 3. For LFSR, the linear functionis very important in deciding the period after which the bits will repeat.

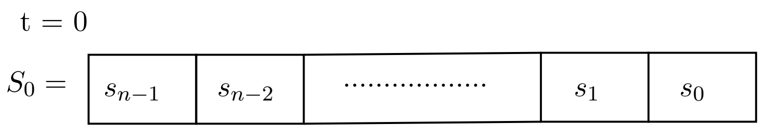
2.3.1 Period of an LFSR

Consider S_0 to be a non-zero state. S_0 is repeated after m clocking of LFSR, then m will be the period of LFSR. An LFSR with n-bits can have a maximum period of $2^n - 1$.

If there is an LFSR where different non-zero states are repeating at certain different number of clocking. Let us say x_1 number of states repeat after P_1 clocking, x_2 number of states repeat after P_2 clocking and so on x_n number of states repeat after P_N clocking. Here, every non-zero state is present in at least and only one x_i. Therefore, if we take any non-zero state, then it will repeat after certain number of clocking which will belong to the set $\{P_1, P_2, ..., P_N\}$. Therefore, the period of LFSR will be:

$$\text{Period of LFSR} = LCM(P_1, P_2, ..., P_N)$$

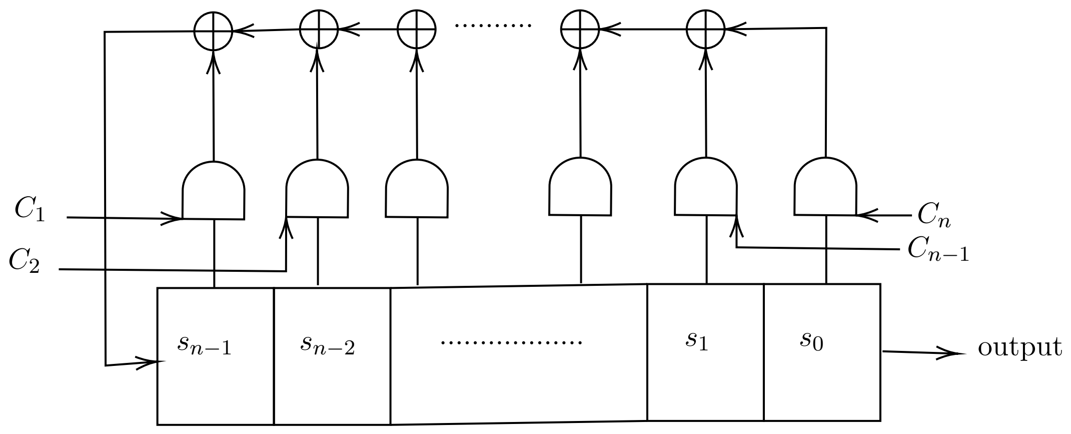
Consider an n-bit LFSR,



We know that after clocking, s_n will be,

$$s_n = L(s_0, s_1, ..., s_{n-1})$$
$$s_n = c_1 \cdot s_{n-1} \oplus c_2 \cdot s_{n-2} \oplus ..... \oplus c_n \cdot s_0 \text{ where } c_i \in \{0, 1\}$$

This is known as Algebraic Normal Form of writing s_n. Therefore, we can see that to implement LFSR in hardware, we only need AND and XOR gates. The circuit for LFSR is given below. The value of s_i will be xored or not depends on the value of c_{n-i}.



Corresponding to every LFSR, we have a Linear Feedback Function (L). Corresponding to LFF, we can construct a polynomial $f(x)$.

$$L = c_1 \cdot s_{n-1} \oplus c_2 \cdot s_{n-2} \oplus ..... \oplus c_n \cdot s_0$$
$$f(x) = 1 + c_1 \cdot x + c_2 \cdot x^2 + ... + c_n \cdot x^n$$

The polynomial $f(x)$ is known as connection polynomial of LFSR. If anyone of the linear feedback function or the connection poylnomial is known, the other can be easily constructed. Since, $c_i \in \{0, 1\}$ for $1 \leq i \leq n$, therefore, $f(x) \in F_2[x]$. Therefore,

$$\text{n-bit LFSR} \iff \text{Linear Feedback Function} \iff \text{one polynomial in } F_2[x] \text{ of degree} \leq n$$

We know that is S_0 repeats after $2^n - 1$ clocking, then it is a full period LFSR. Now, consider a connection polynomial of degree n in $F_2[x]$.

1. If the connection polynomial is primitive polynomial, then the LFSR will have full period.

   If we recall, during AES we studied that if $G(x)$ is a primitive polynomial, then $(F_2[x]/\langle G(x)\rangle, +, *)$ is a field where $F_2[x]/\langle G(x)\rangle$ contains all polynomials with degree less than degree of G(x). Similarly, here we have a n degree connection polynomial. If it is primitive, then we can construct all polynomials of degree less than n. That is, we can construct $2^n - 1$ polynomials. Therefore, we can generate all the possible non-zero states of LFSR.

2. If connecting polynomial is irreducible (and not primitive), $F_2[x]/\langle G(x)\rangle$, then the period of LFSR will divide $2^n - 1$.

3. If connecting polynomial is reducible, then different state will have different cycle length (different period).

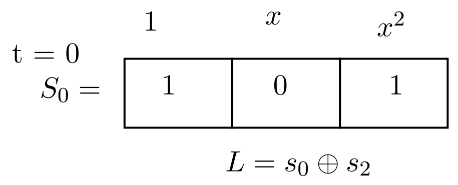
Example: Consider the example of the 3-bit LFSR taken earlier where $L = s_0 \oplus s_2$.
Solution: The connecting polynomial for the LFSR is:

$$f(x) = 1 + x + x^3$$

Now, checking if $f(x)$ is primitive or not. Let's try to generate all possible non-zero polynomial from $f(x)$.

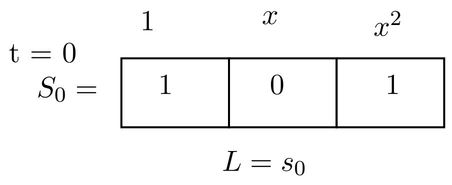
$$\{1, x, x^2, x^3 = x + 1, x^2 + x, x^3 + x^2 = 1 + x + x^2, x^3 + x^2 + x = x^2 + 1, x^3 + x = 1\}$$

Since, all polynomials of degree less than 3 are generated by using $\langle x\rangle$. Hence, $x^3 + x + 1$ is a primitive polynomial. Hence, the given 3-bit LFSR is full periodic.
Another but not so good way to find if LFSR is fully periodic or not is to construct a polynomial g(x) from the initial state and then keep multiplying it by x, again and again until we get g(x) again. g(x) is constructed by taking those powers of x for which the bit is 1 in the initial state. Use the connecting polynomial in case degree of g(x) gets greater than degree of connecting polynomial.



Therefore, the polynomial $g(x) = 1 + x^2$ and connecting polynomial $f(x) = x^3 + x + 1$.
$1 \times g(x) = 1 + x^2$
$x \times g(x) = x + x^3 = 1$
$x^2 \times g(x) = x$
$x^3 \times g(x) = x^2$
$x^4 \times g(x) = x^3 = x + 1$
$x^5 \times g(x) = x^2 + x$
$x^6 \times g(x) = x^3 + x^2 = x^2 + x + 1$
$x^7 \times g(x) = x^3 + x^2 + x = x + 1 + x^2 + x = 1 + x^2 = g(x)$
Since, after multiplication from x^7, g(x) is repeated again and $7 = 2^3 - 1$. Therefore, the given LFSR is fully periodic LFSR.
Question Suppose there is a non-zero state, and we are getting a period of $2^n - 1$. Is it guaranteed that period will be $2^n - 1$?
Solution: Yes, because all the states are generated in between these $2^n - 1$ states that are generated between the repetition of the given non-zero state. Hence, each state is repeated after $2^n - 1$ states. Hence, beginning with any state, the state will repeat after $2^n - 1$ states only. Hence, period will be $2^n - 1$.

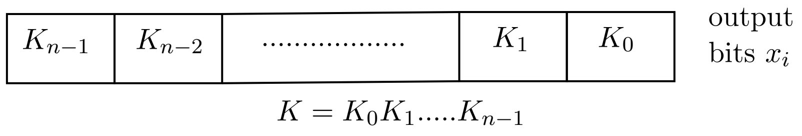
Example:



Therefore, the polynomial $g(x) = 1 + x^2$ and connecting polynomial $f(x) = x^3 + 1$.
$1 \times g(x) = 1 + x^2$
$x \times g(x) = x + x^3 = x + 1$
$x^2 \times g(x) = x^2 + x$
$x^3 \times g(x) = x^3 + x^2 = 1 + x^2 = g(x)$

Since, after multiplication from x^3, g(x) is repeated again and $7 \neq 2^3 - 1$. Therefore, the given LFSR is not fully periodic LFSR.

Let us now talk about the security of LFSR. Usually, we keep the secret key in the memory of LFSR, that is, in the register.



We will keep secret key or some other public parameter in the register and we will generate some bits.

$$\text{output bits } x_i \rightarrow \text{keystream bits } Z_i$$
$$m_i \oplus Z_i = C_i \rightarrow \text{ciphertext bits}$$

Every time we will be clocking the LFSR, we will get one bit as output and will have feedback also. Now, if we consider Known Plaintext Attack Model, that is, we know certain plaintext bits and corresponding ciphertext bits and our aim is to find the secret key. Therefore, we know m_i and c_i. Clearly,

$$Z_i = m_i \oplus C_i$$

Therefore, we know Z_i corresponding to the know m_i and C_i. Now, first output bit according to LFSR id K_0, the second output bit is K_1 and so on. That is, if we know the first n bits of the message and corresponding ciphertext, we can get entire secret key.

Even if we don't know the first n keystream bits, we can form a linear system of equations, because whatever bit we will get at leftmost position during clocking will be a linear function on the initial state only. For example, in the first clocking,

$$K_n = L(K_0, K_1, ..., K_{n-1})$$

and in the second clocking,

$$K_{n+1} = L(K_1, K_2, ..., K_n)$$

but K_n in again a linear combination of initial state only. Eventually, every bit coming to leftmost position is linear function on initial state only. Therfore, even if we know any key stream bits, that is any message bit and corresponding ciphertext bit, then,

$$Z_i = m_i \oplus C_i$$

and Z_i will always be a linear function of $K_0, K_1, ..., K_{n-1}$. If we know first n bits, Z_i is a linear function with only one term (K_0 or K_1 or K_{n-1}). So, if we know Z_i from known plaintext attack for any message bits. It is not required to know consecutive first n-bits. If we know any mesage bits and corresponding ciphertext bits, we can form a system of linear equations and solve that to get the secret key. Hence, simple LFSR does not provide good security.

Let us say you know one bit of message and corresponding ciphertext, that means you have only one linear equation. Let us say some i^{th} bit (that is, not necessarily the first bit). Is is possible to reduce the search complexity of the secret key using that equation. The exhaustive search complexity of recovering the secret key is 2^n because this many keys are possible.

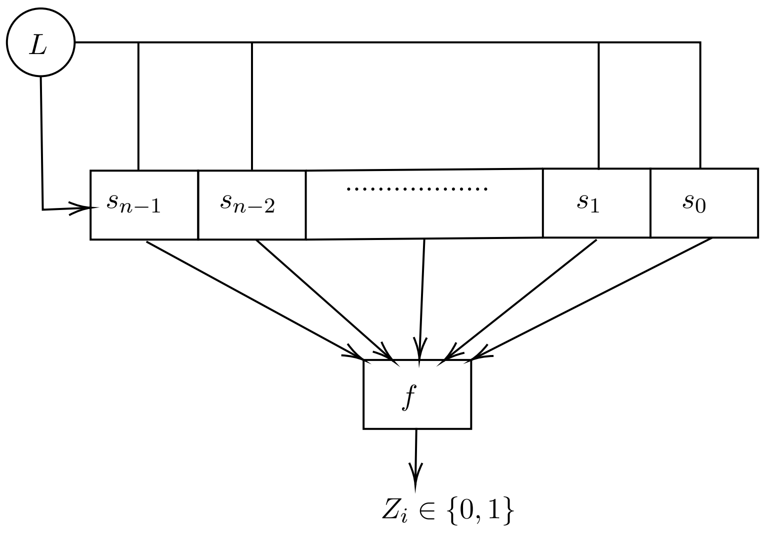
Let us say there is one $Z_i = 0$ and the linear equation formed is,

$$Z_i = L(K_0, K_1, ..., K_{n-1})$$
$$0 = Z_i = K_2 \oplus K_6 \oplus K_{n-2}$$

Using the linear equation, is it possible to reduce the complexity of searching the key? If we have correct guess for say K_6 and K_{n-2}, then K_2 is always determined. The search complexity, thus, is reduced to half.
Therefore, depending upon how many equations you are getting and the property of the system of equations, the search complexity will be reduced definitely. Therefore, knowing only 1 bit creates huge impact on the search complexity.
Say we have one system of linear equations with infinite solutions, so we will get all possible solutions here. So, if we have n variables but we have only few equations (say 5 or 6), still these equations will reduce the search complexity by a good margin. Therefore, using Known Plaintext Attack it is easy to break LFSR.

2.3.2 LFSR with Non-Linear Filter Function

Here, we will consider an l-bit boolean function f which takes l-bits as input and produces one bit as output. We will take l-bits out of the n-bits of the state of LFSR as input to f and use the output of f as $Z_i \in \{0, 1\}$. The function f here is a non-linear function.



$$f : \{0,1\}^l \rightarrow \{0,1\}$$
$$n \geq l$$
$$C_i = m_i \oplus Z_i$$

The state update function of LFSR will be same - there will be linear feedback function (L) and shifting as earlier. Therefore, if we select l fixed position from the n positions in the register, at each clocking the value at these position will update. If the function f is good enough, the output will still be random looking. In fact, this will also have full period if f is a good function and the connection polynomial corresponding to L is primitive.

The advantage here is that even if we know m_i and corresponding c_i from Known Plaintext Attack model, and consequently we know Z_i. The Z_i is now a non-linear function of state bits of LFSR. Solving a non-linear system of equation might be computationally difficult.

The state update function of LFSR is, say α. Therefore,

$$S_{t+1} = \alpha(S_t)$$
$$Z_{t+1} = f(S_{t+1})$$

Let us look at the LFSR state at clocking time t.

$$S_t = (s^t_{n-1}, s^t_{n-2}, ..., s^t_0)$$

The state of LFSR at clocking time (t+1) will be,

$$S_{t+1} = (s^{t+1}_{n-1}, s^{t+1}_{n-2}, ..., s^{t+1}_0)$$

Suppose the shifting to be right shift, therefore,

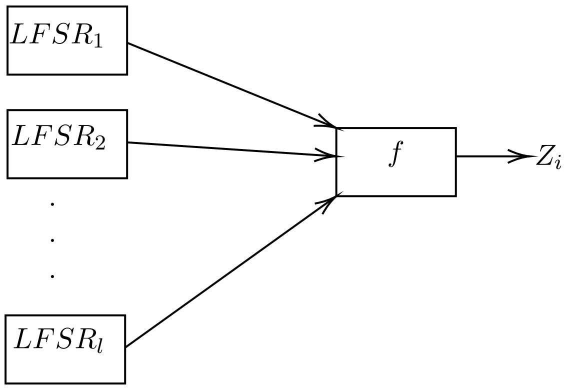
$$s^{t+1}_0 = s^t_1, s^{t+1}_1 = s^t_2, ..., s^{t+1}_{n-2} = s^t_{n-1}, s^{t+1}_{n-1} = L(s^t_{n-1}, s^t_{n-2}, ..., s^t_0)$$

The state update can be represented as a matrix multiplication in the following way,

$$S^{t+1} = \begin{bmatrix} s^{t+1}_0 \\ s^{t+1}_1 \\ \vdots \\ S^{t+1}_{n-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ c_n & c_{n-1} & c_{n-2} & c_{n-3} & \dots & c_1 \end{bmatrix} \begin{bmatrix} s^t_0 \\ s^t_1 \\ \vdots \\ S^t_{n-1} \end{bmatrix}$$
$$L = c_n \cdot s_0 \oplus c_{n-1} \cdot s_1 \oplus \ldots \oplus c_1 \cdot s_{n-1}$$

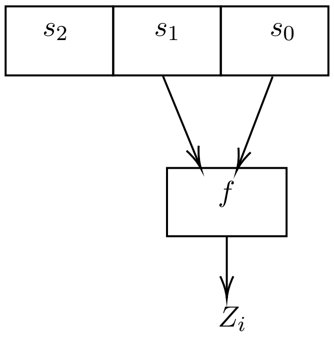
2.3.3 LFSR with Combiner Function

We have a similar function f as we discussed above. However, here we have l number of LFSR's. The output of these l LFSR's, that is, l-bits becomes the input for the combiner function f whose output is treated as Z_i. The function f is non-linear.



$$C_i = m_i \oplus Z_i$$

Example: Consider the following 3-LFSR with non-liner filter function f.



$$f : \{0,1\}^2 \rightarrow \{0,1\}$$
$$f(x_0, x_1) = x_0 \cdot x_1$$

Here, if we draw the truth table of f, it will look like,

x_0	x_1	f
0	0	0
0	1	0
1	0	0
1	1	1

We can see that $Pr[Z = 0] = \frac{3}{4}$. Here, f is not a good function because $Pr[Z = 0] > Pr[Z = 1]$, that is, it is highly biased. We need to design f such that the output is unpredictable. Here, we can predict that output will be zero with a higher probability. If we have a stream cipher with this model and we are able to find out the f function. We can eventually break the stream cipher. Therefore, f must be selected carefully.

2.4 Non-Linear Feedback Shift Register

After a few years, it was observed that LFSRs are good ciphers which are still secure but most of the ciphers can be broken using various adverse cryptanalysis techniques.

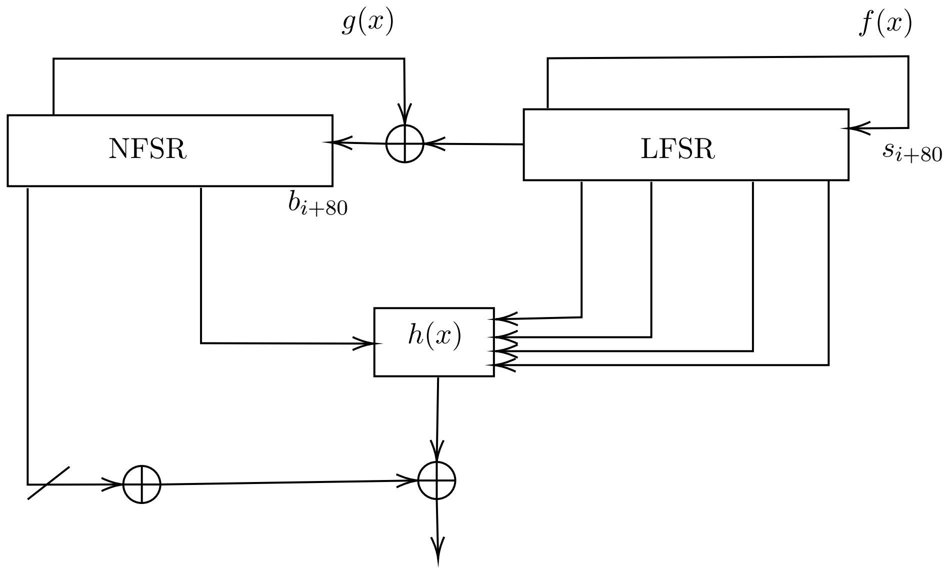
To prevent all this, there is a concept of stream ciphers known as Non-Linear Feedback Shift Register (NFSR's). From the name, it can be anticipated that the feedback function will be non-linear in NFSR. In fact, in modern scenarios, all stream ciphers are based on NFSR.

In NFSR, the mechanism is similar to LFSR, but the feedback is non-linear.
Note: The problem in NFSR is that if we consider a Non-Linear Feedback Function, there is no methodical proof which can determine the period of NFSR. For LFSR, if the connection polynomial is primitive, then the LFSR is fully periodic. In case of NFSR, we don't have any such proof or method.

2.4.1 Grain

Grain is a standard stream cipher which with an 80-bit key and a 64-bit IV. In block ciphers, there is only one standard algorithm, that is, the AES. There are other good block cipher algorithms that are as secure as AES, but they are not efficient. However, in stream ciphers, multiple ciphers are standardized. In fact, in most mobile and hardware devices, you will find some non-standardized stream cipher. In most 4G mobile phones, stream ciphers are based on ZUC stream cipher which is not a standardized stream cipher.

The design of Grain is based on NFSR with LFSR and a non-linear combiner function. Although the design is a little complicated, but its implementation is extremely efficient. Let us look into the design of Grain.



The NFSR and LFSR are each 80-bit. The connection polynomial corresponding to LFSR is primitive and hence LFSR is fully periodic. The function $g(x)$ is non-linear feedback function of NFSR and the function $f(x)$ is linear feedback function of LFSR. If we say that cipher is clocked, it means that both NFSR and LFSR are clocked. Both LFSR and NFSR are shifted towards left.

In every clocking, it takes 1 bit from the NFSR and 4 bits from LFSR and pass it to $h(x)$. The position of these bits are fixed. $h(x)$ is also a non-linear filter function. It will produce 1-bit output. This bit will be xored by the 7-bits from the NFSR (the line with a little slash). The 7-bits from NFSR are taken from fixed position and are xored and then the output is zored with the output of $h(x)$.

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 (connection) polynomial of the LFSR, $f(x)$ is a primitive polynomial of degree 80. It is defined as:

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

Therefore, the state update function of LFSR is:

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

The feedback polynomial of the NFSR, $g(x)$, is defined as:

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

The state update function of NFSR is:

$$b_{i+80} = s_i \oplus b_{i+62} \oplus b_{i+60} \oplus b_{i+52} \oplus b_{i+45} \oplus b_{i+37} \oplus b_{i+33} \oplus b_{i+28} \oplus b_{i+21} \oplus b_{i+14} \oplus b_{i+9} \oplus b_i \oplus b_{i+63}b_{i+60} \oplus b_{i+37}b_{i+33} \oplus b_{i+15}b_{i+9} \oplus b_{i+60}b_{i+52}b_{i+45} \oplus b_{i+33}b_{i+28}b_{i+21} \oplus b_{i+63}b_{i+45}b_{i+28}b_{i+9} \oplus b_{i+60}b_{i+52}b_{i+37}b_{i+33} \oplus b_{i+63}b_{i+60}b_{i+21}b_{i+15} \oplus b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} \oplus b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} \oplus b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21}$$

The feedback bit in NFSR is not just the non-linear feedback function of NFSR. The output of the non-linear feedback function is xored with the output bit (not the feedback bit) of LFSR, that is, s_i.

The contents of the two shift registers represent the state of the cipher. From this state, 5 variables are taken as input to a boolean function, $h(x)$. The function $h(x)$ is a degree 3 function defined as:

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

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:

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

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

The NFSR stream cipher is a good cipher but we don't have any proof for finding its period. But because of LFSR, we can claim that its period will be at least $2^{80} - 1$.

In most of the standardized stream ciphers, there are two phases:

- **Key IV Initialization Phase:** In Key IV (or Key) Initialization Phase the cipher initializes its state. In this phase, the cipher is initialized automatically. That is, certain bits will be put in the state and it will update the state in a peculiar way and generate a random looking state.
- **Keystream Generation Phase:** In Keystream Generation Phase, the cipher produces output bits (keystream bits, Z_i) which are used for encryption.

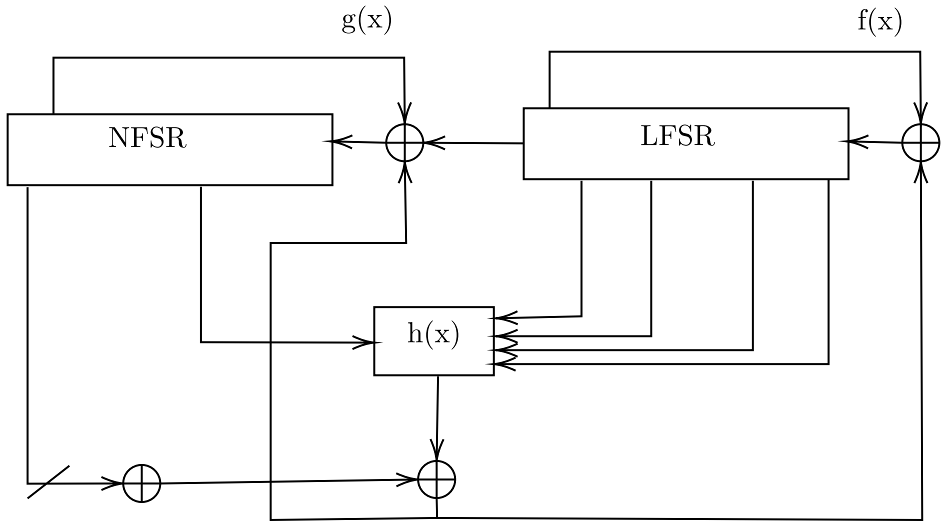
2.4.1.1 Key IV Initialization Phase of Grain

Before any keystream is generated the cipher must be initialized with the key and the IV. Let the key be K and Initialization Vector be IV. The bits of K and IV can be represented as:

$$K_i, 0 \leq i \leq 79$$
$$IV_i, 0 \leq i \leq 63$$

The initialization of the key is done as follows. First, we load the NFSR with the key bits, $b_i = k_i, 0 \leq i \leq 79$, then load the first 64 bits of the LFSR with the IV, $s_i = IV_i, 0 \leq i \leq 63$. The remaining bits of the LFSR are filled with ones, $s_i = 1, 64 \leq i \leq 79$. We know that if LFSR is in zero state, then it remains in zero state. Hence, to avoid LFSR being in zero state, the remaining 16 bits are filled with 1. Now, since we know the LFSR is fully periodic, hence it will never be in zero state.

At each clocking, we know the cipher will produce a single bit output, which in the Keystream Generation Phase is Z_i. However, in Key IV Initialization phase, this output is fed back and xored with the input, both to the LFSR and NFSR.



Hence, the key and the IV are mixed in a systematic way. If the key is random, it is expected that after certain number of clocking, we will have random looking NFSR and LFSR states. The Key and the IV are mixed because the computed Z_i is used in computing the feedback bit of both LFSR and NFSR. Therefore, if the key is secret, it is expected that after running certain number of clocking, the states of LFSR and NFSR can not be guessed. In fact, the cipher is clocked for 160 times after loading the key in NFSR and IV in LFSR. Therefore, after 160 clocking, the states of NFSR and LFSR will be random looking.

After Key IV Initialization Phase is completed, the output bits Z_i are used for encryption. This phase is known as the Keystream Generation Phase. As we have discussed during Stream ciphers, that we have a function $f(K, IV)$ which outputs the keystream bits. This is the f function we were talking about.

If we clock the cipher for, let's say, r number of rounds, where $r < 160$, then we will be calling it as Grain V1 with reduced rounds. Let us say we clock it for 100 times (and not 160 times), then we call it as Grain V1 with 100 rounds. Grain cipher is secure after around 115 rounds, that is, it can be broken till around 115 rounds. Like AES, AES-128 can be broken till 7^{th} round, but it is secure till 10^{th} round.

Grain is one of the most efficient cipher. The implementation of Grain is extremely easy. In fact, we can run this cipher in parallel mode in a very quick way.

2.4.2 RC4: Rivest Cipher 4

RC4 is a stream cipher and was the most used cipher till around 2011. The 4 in RC4 stands for the 4 lines of code we need to write to implement RC4. Consequently, RC4 is extremely efficient. Even after using such a simple implementation of RC4, there was no such attack on RC4 till 2011. It generates bits for the encryption and also we can use RC for generating permutations. Let us look into the design of RC4.

RC4 contains as S-Box S, typically an array of size $N = 2^n$, where each entry is a n-bit integer. Typically, $n = 8$, hence, $N = 256$.

$$S = (S[0], S[1], ..., S[N-1])$$

S is initialised as the identity permutation, that is:

$$S[i] = i, 0 \leq i \leq (N-1)$$

A secret key κ of size l bytes (typically $5 \leq l \leq 16$). An array K is used to hold the secret key, where each entry is an n-bit integer.

$$K = (K[0], K[1], ..., K[N-1])$$

The key is repeated in the K array at key length boundaries. For example, if the key length is 40 bits, then $K[0]$ to $K[4]$ are filled by the secret key and then this pattern is repeated to fill up the entire array K. Mathematically, it is:

$$K[y] = \kappa[y \ mod \ l], \text{ for } 0 \leq y \leq N-1$$

There are two components or phases in RC4. One is Key Scheduling Algorithm (KSA, initializes the state) and the other is Pseudo Random Generation Algorithm (PRGA, keystream generation phase).

```
**Initialization:**
**for** $i = 0$ *to* $N-1$ **do**
    $S_i = i$;
**end**
**Scrambling:**
$j = 0$;
**for** $i = 0$ *to* $N-1$ **do**
    $j = (j + S_i + K_i) \bmod N$;
    Swap($S[i], S[j]$);
**end**
```

Algorithm 1: RC4 KSA

The algorithm for Key Scheduling is given above. We are not getting any output during Key Scheduling. We are just permuting S array using the key array K. Suppose, during scrambling the first value of j comes out to be 2. Initially, $S[0] = 0$, but now $S[0]$ and $S[2]$ will be swapped. Hence, we are permuting the S array using the j index. If we don't know the key, we will have a random looking permutation S after scrambling.

```
**Initialization:**
$i = j = 0$
**Output Keystream Generation Loop:**
$i = (i + 1);$ mod $N$
$j = (j + S[i])$ mod $N$
$Swap(S[i], S[j])$
$t = (S[i] + S[j])$ mod $N$
$z = S[t]$
```

Algorithm 2: RC4 PRGA

It is worth noting here that the output is pseudo random bytes z (not bits). The keystream output z is xored with the next byte of plaintext to generate the ciphertext at the sender end. The receiver having the same secret key is able to generate the same z. The receiver then xor the generated keystream bits with the ciphertext to get the plaintext back.

3 Public Key Cryptography

Public-key cryptography, or asymmetric cryptography, is the field of cryptographic systems that use pairs of related keys. Each key pair consists of a public key and a corresponding private key. Key pairs are generated with cryptographic algorithms based on mathematical problems termed one-way functions.

3.1 Diffie and Hellman Key Exchange Algorithm

Suppose we are having a symmetric key encryption setup, that is, Alice and Bob have same key K.

$$\begin{tikzcd}
\underline{\text{Alice}} & \underline{\text{Bob}} \\
\text{K} & \text{K} \\
C = Enc(M, K) \arrow[r, "C"] & M = Dec(C, K)
\end{tikzcd}$$

Alice can encrypt a message and send it to Bob and Bob can decrypt the ciphertext. The problem here is that the secret key has to be same with Alice and Bob. Otherwise, the decryption will never give the correct plaintext. The problem lies in sharing the secret key. Before 1976, there was only one mechanism to share this secret key; which is, you have to meet the other party secretly. In 1976, Diffie and Hellman proposed a Key Exchange mechanism and with their results, the domain of Public Key Cryptography began. It was published in IEEE Transactions on Information Theory.

Let us first recall the concept of Group and a cyclic group before going further in the Diffie and Hellman Key Exchange Algorithm. A set G along with an binary operation, (G, *) is called to be Group if it satisfies the following properties:

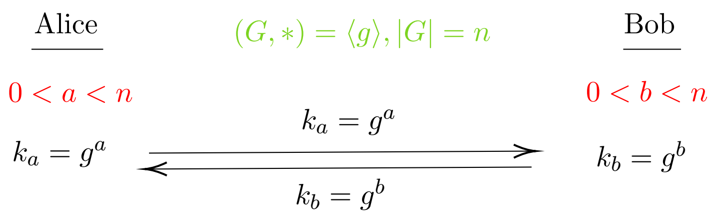
1. The operation * is closed under G, that is, if $a, b \in G$, then $a * b \in G$.
2. $*$ is associative on G, that is, $a * (b * c) = (a * b) * c \ \forall \ a, b, c \in G$
3. There is an element $e \in G$ called the Identity Element, such that $a * e = a = e * a \ \forall \ a \in G$.
4. For each $a \in G$, there exists an element $a^{-1} \in G$, called the inverse of a, such that $a * a^{-1} = e = a^{-1} * a \ \forall \ a \in G$.

(G, *) is a cyclic group if every element in G can be generated using only one element $g \in G$, that is,

$$\forall a \in G \ \exists \ g \in G, \text{ such that } a = g^i, \text{ where } i \in Z.$$

The element g is called the generator of G, and is denoted as $G = \langle g \rangle$.

Now, according to Diffie and Hellman Key Exchange Algorithm, Alice and Bob will agree on one cyclic group (G, *) over a public communication. It means that via a public communication, they will agree that they both will use the cyclic group $(G, *)$ whose generator is g.



The part that is written with green colour in the figure above is public where as the part written in red is secret to the person on whose side it is written. Alice selects a number a such that $0 < a < n$ and keeps it secret to herself. Similarly, Bob selects a number b such that $0 < b < n$ and keeps it secret to himself. Alice will compute $k_a = g^a$ and send it to Bob and Bob will compute $k_a = g^a$ and send it to Alice over the public channel. That is, Alice is making k_a public and Bob is making k_b public. Now, Alice and Bob have the following data with them.

Alice	Bob
a	b
g	g
$k_b = g^b$	$k_a = g^a$

Now, since, g^a and g^b belongs to same cyclic group. Given g^b, if we have a, then we can compute $(g^b)^a$. Therefore, Alice will compute $(g^b)^a$ and Bob will compute $(g^a)^b$. Therefore, Alice now has g^{ba} and Bob has g^{ab}. Since, a and b are integers, therefore,

$$a \cdot b = b \cdot a$$
$$\implies g^{ba} = g^{ab}$$

Hence, Alice and Bob have same element g^{ab}. They can use this element to be their secret key and start the communication using the symmetric key encryption algorithms. The same key generated by Alice and Bob, that is, g^{ab} is called the Shared Secret Key.

We can see that Alice and Bob are exchanging some data over the public channel and they are able to establish a secret key. Since, a and b are only known to Alice and Bob respectively, that is, they are not shared publicly, hence, they are known as secret keys. However, $k_a = g^a$ and $k_b = g^b$ are shared by Alice and Bob to each other over a public channel. Hence, they are made public and are known as public keys. Both the parties have two keys, one public key and one secret key. Note that there may be some technique to compute $a(b)$ from $g^a(g^b)$, but since Alice (Bob) is not sharing $a(b)$, we are assuming that it is secret key for Alice(Bob).

Now, if we don't know the secret key of Alice (a), even if we know g^a and g^b, we will not be able to compute g^{ab}, which is the key used for communication between Alice and Bob. That means, without knowing the secret key of Alice or Bob, we will not get the shared secret key used for communication between Alice or Bob.

Now, we have g^x as public key and x as secret key. If we have a very good cyclic group (G, *), based on the properties of the group, finding x from g^x is computationally difficult. It is possible theoretically, but the amount of time it requires is exponential. This hard problem is known as Discrete Log Problem. Since, finding x from g^x for a good group is computationally hard and we are using this group for establishing the shared secret key, then the key establishment mechanism will be secure. Therefore, the security of Diffie-Hellman Key Exchange Algorithm relies on the fact that Discrete Log Problem is hard for certain groups.

One obvious way to compute x from g^x, as we know G, g and g^x, is to compute g^i for $1 \leq i \leq n-1$ and return i if we get $g^i = g^x$.

```
for $i = 2$ to $N - 1$ do
    if $g^i == g^x$ then
        t = i;
        break;
    end
end
```

Algorithm 3: Brute Force Algorithm to find x from g^x

The complexity here will be equal to the size of the set G, i.e $|G|$. Let us say our group contains $2^{512} - 1$ elements, then getting x using this mechanism is impossible. The loop in this case will never terminate in practical time. Therefore, for the Discrete Log Problem to be hard on a group (G, *), the group must follow certain properties. These properties are mention below:

1. Size of the set G, i.e $|G|$, should be very large.
2. The group operation $*$ must be chosen carefully. Even if the group size is very large, you can still study the properties of the group operation and can find x from g^x very easily for certain group operators. For example, consider the cyclic group $G = (Z_p, +_p)$, p is a very large prime number and $G = \langle g \rangle$. If we try to compute g^i, it will be equal to:

$$g^i = g +_p g +_p ..... +_p g$$
$$g^i = i \cdot g$$

So, if we have generator of G, i.e g, g^i can be easily computed. Now, if we know g^i, finding i will be very easy and we will not have to search for i exhaustively. Multiplying both the sides with g^{-1} will give us:

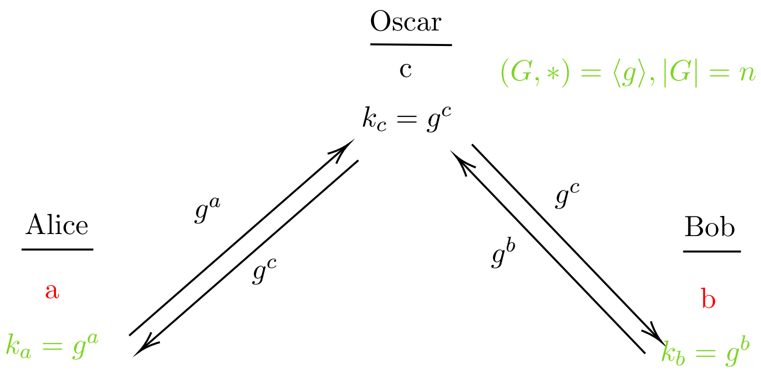
$$i = g^{-1} \cdot g^i \text{ mod p}$$

We know g^i and we can compute g^{-1} in polynomial time using the Extended Euclidean Algorithm, which will definitely hold true because gcd(i, p) = 1 as p is prime. Hence, $+_p$ is not a good group operation.

3.2 Man in the Middle Attack on Diffie-Hellman Key Exchange Algorithm

Man in the Middle means there is a person who is listening to your communication and he has control over the data. Suppose you are writing a letter to your friend. You write the letter, put it in an envelope and visit the nearby post office to send it to your friend. Now, the responsibility of the post office is to transfer the letter to your friend. If the middle man, i.e. the post office, is corrupted, there will be problems and your message can be read or can be changed by the post office. In Whatsapp, you exchange certain data with its server, the server, in turn, send it to the intended person. If the Whatsapp server is corrupted, your data can be leaked.

Let's try to understand the Man in the Middle Attack on the Diffie-Hellman Key Exchange Algorithm.



Here, Oscar can capture the communication between Alice and Bob. Suppose if Alice has sent some message to Bob. Oscar can capture this message and can send a different message to Bob. Suppose Alice has sent g^a to Bob. Oscar will compute g^c as g and G are public. Oscar will intercept the g^a sent by Alice to Bob and send g^c to Bob. Since, Bob doesn't have any mechanism for authenticating that the message is coming from Alice, Bob will believe that Alice has sent her public key to him. However, this is not the case, Bob has actually recieved g^c (not g^a).
Bob will share his public key g^b to Alice. Again, Oscar will intercept this message and send g^c to Alice. Alice, again, doesn't have any mechanism to check if it is coming from Bob or not, so she will accept it.
With the data that Alice has, she can compute:

$$(g^c)^a = g^{ac}$$

With the data that Bob has, he can compute:

$$(g^c)^b = g^{bc}$$

and with the data that Oscar have, he can compute:

$$(g^a)^c = g^{ac}$$
$$(g^b)^c = g^{bc}$$

Now, Oscar has two shared secret keys, one is same with Alice and the other is same with Bob. Alice and Bob will begin the communication as they are unaware about the middle man. Suppose Alice has sent a message to Bob.

$$\begin{tikzcd}
& \begin{array}{c}\underline{\text{Oscar}}\\ Dec(C_1, g^{ac}) = M \\ C_2 = Enc(M, g^{bc})\end{array} \arrow[dr, "C_2"] & \\
\begin{array}{c}\underline{\text{Alice}}\\ C_1 = Enc(M, g^{ac})\end{array} \arrow[ur, "C_1"] & & \begin{array}{c}\underline{\text{Bob}}\\ Dec(C_2, g^{bc}) = M\end{array}
\end{tikzcd}$$

Alice will encrypt the message using g^{ac} and send it. Oscar will receive this message and decrypt it using g^{ac}. Oscar will get the original message sent by Alice. Now, Oscar will encrypt this message using g^{bc} and send it to Bob. Bob on receiving the message will decrypt it using g^{bc} and will receive the original message sent by Alice. Both Alice and Bob will be unaware of the fact that Oscar is receiving all the communication between them. This is known as Man in the Middle Attack on Diffie-Hellman Key Exchange Algorithm.

In Whatsapp or Telegram, Diffie-Hellman Key Exchange is used. This exchange is performed by the middle person, i.e the Whatsapp or Telegram. If they want they can do anything with the data. However, there are certain mechanisms which they implement to provide security.

If we wish to compute the shared secret key g^{ab}, we have to compute g^a and g^b where $|G|$ is large and also a and b should be large. In fact, if you will use any pseudo-random selector from 1 to (n-1), it is highly guaranteed that it will be in the middle of 1 and (n-1). Hence, if $|G| = 2^{512}$, then a and b will be in the order of 256 bits. The problem here is how to compute g^a and g^b efficiently. It is not possible to compute g^a and g^b by running a loop and multiplying by g in each iteration. This is practically impossible. Therefore, to compute g^a and g^b, we use the Square and Multiply Algorithm. According to the algorithm, suppose we want to calculate x^c. The binary representation of c is $c_{l-1} \dots c_1, c_0$. Then,

$c = \Sigma_{i=0}^{l-1} c_i \cdot 2^i$

$x^c = x^{\Sigma_{i=0}^{l-1} c_i \cdot 2^i} = \Pi_{i=0}^{l-1} x^{c_i \cdot 2^i}$

$x^c = x^{c_0 \cdot 2^0} \dots x^{c_{l-1} \cdot 2^{l-1}}$

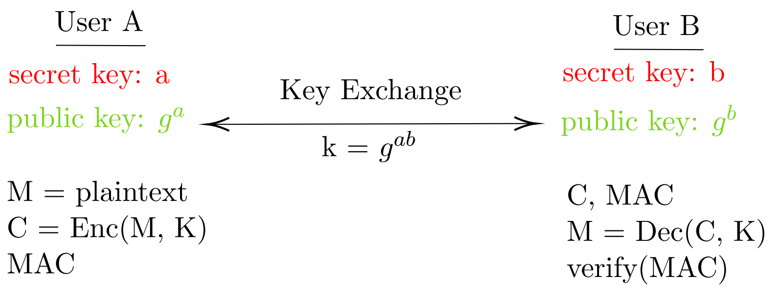
These are just $log(c)$ multiplications, hence, we can compute x^c in logarithmic time of c.

Let us take and example and calculate 3^5. Therefore, x = 3 and c = 5 = $(101)_2$. Initially Z = 1, i = 2 (since l = 3). First iteration of for loop, $Z = Z^2 = 1$, since $c_2 = 1$, $Z = Z * x = 3$. Second iteration of for loop, Z = 3, i = 1. $Z = Z^2 = 9$, since $c_1 = 0$, Z will not be multiplied by x. Last iteration of for loop, Z = 9, i = 0. $Z = Z^2 = 81$, since $c_0 = 1$, $Z = Z * x = 81 * 3 = 243$. Z = 243 = 3^5 is returned.

```
Input: x and c
Z = 1;
for $i = l - 1$ to $N0$ do
    $Z = Z^2$;
    if $c_i == 1$ then
        Z = Z * x;
    end
end
return Z;
```

Algorithm 4: Square and Multiply Algorithm to find x^c

Using the Square and Multiply Algorithm g^a and g^b can be computed efficiently in $log(a)$ and $log(b)$ time complexity respectively.



Now, if we have a Man in the Middle Attack, then we can have a communication setup where the MAC can also be incorporated. The MAC authenticates the source of the message as well as the correctness of the message. So, now we can authenticate the source, hence, the Man in the Middle Attack will not be possible now.

3.3 RSA (Rivest Shamir Adleman) Encryption

It is the first public key encryption algorithm. Before beginning any discussion on RSA encryption, let's first recall a few concepts.

- The Euler's Totient Function $\phi(n)$ denotes the number of integers less than n that are co-prime to n, i.e number of x such that gcd(x, n) = 1 where $1 \leq x \leq n-1$
- Let there be a set $S = \{x \text{ mod m}\}$ such that $|S| = m$.

$$S = \{r_1, r_2, \dots, r_m\}$$

All the elements in the set S are unique (usually they are from 0 to m-1). Assume an integer a such that $gcd(a, m) = 1$. Let's say that there is another set S_1 such that,

$$S_1 = \{ar_1 \bmod m, ar_2 \text{ mod m}, \dots, ar_m \bmod m\}$$

Since, $\{r_1, r_2, \dots, r_m\}$ are different elements and gcd(a, m) = 1, it can be concluded that $\{ar_1 \bmod m, ar_2 \text{ mod m}, \dots, ar_m \bmod m\}$ will also be m unique elements. This can be proves using contradiction. Suppose, if $ar_i = ar_j$ for $r_i \neq r_j$. Therefore,

$$ar_i \equiv ar_j \text{ mod } m$$

Since, gcd(a, m) = 1, therfore, $1 = ab + ms$(from Bezout's Identity). Therefore, there exists an integer b such that $ab \equiv 1$ mod m. The value of b is known as multiplicative inverse of a and it can be found using Extended Euclidean Algorithm. Therefore, on multiplying the above equation by b on both sides gives us,

$$b \cdot a \cdot r_i \equiv b \cdot a \cdot r_j \text{ mod } m$$

$$r_i \equiv r_j \text{ mod m } (\because ab \equiv 1 \text{ mod m})$$

Hence, it is a contradiction to our initial assumption that $r_i \neq r_j$. Hence, elements in the set S_1 will be unique iff gcd(a, m) = 1.