An introduction to Symmetric Key Cipher Algorithms

BY CALEB STEINMETZ

What is the purpose of this talk?

What this is:

- Understand basic topics in cryptography
- Explore the difference between block and stream ciphers
- A basic understand on how these algorithms function
- View the implementation of multiple cipher algorithms in software

What this is NOT:

- A deep mathematical proof on how these ciphers function
- A guide on how to design new cipher algorithms
- A guide on how to crack ciphers
- How to safely store data and keys

Confusion and Diffusion

Confusion: Each bit in the cipher text should depend on multiple parts of the key. This should be done to obscure the connection between the cipher text and the key. The more complex the connection between the key and the cipher text, the higher the confusion.

An algorithm with poor confusion would be a simple XOR with a static key. The issue with this is that each bit in the cipher text only corresponds to one bit of the key (which is not a complex connection). Furthermore, cracking this cipher is quite easy as the key imprints on large strings of zeros.

Example: Low Confusion

Assume the key is XORed with each set of 4 bits within the plain text
Plain Text = 0110 0000 1011 0000 0000 0000 0000 0000
Key = 0011
Cipher Text = 0101 0011 1011 0011 0011 0011 0011 0011

Diffusion: Diffusion is the amount that the output changes if we modify a single bit in the input. For example, an algorithm with a high diffusion rate would expect that for each bit in the plain text modified, around half of the bits in the cipher text will change. Similarly, if we change one bit of the cipher text then we would expect approximately half of the plain text bits to change.

Example: High Diffusion

Before bit swap		**After bit swap**
Plain Text: 1010 0011	→	Plain Text: 1010 0010
Cipher Text: 0011 0011		Cipher Text: 0100 0111

Example: Low Diffusion

Before bit swap		**After bit swap**
Plain Text: 1010 0011	→	Plain Text: 1010 0010
Cipher Text: 0011 0011		Cipher Text: 1011 0011

Symmetric vs Asymmetric Key Encryption

Symmetric key encryption uses a single key that is used to encrypt and decrypt messages. A common issue with this approach is that sharing the key is difficult. If other parties obtain this key, they can encrypt and decrypt messages.

Asymmetric key encryption uses a pair of public and private keys to encrypt and decrypt messages. The public key can be shared to others, but the private key must be hidden to ensure encrypted messages cannot be decrypted by other parties.

<u>For the remainder of this talk we will focus on Symmetric Key Encryption algorithms. But what kind of ciphers use Symmetric Key Encryption? Block and Stream ciphers!</u>

A less formal example…

http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

What is the difference between Block and Stream Ciphers?

Block:

Block ciphers convert plain text into cipher text using a standardized block size. For example, a 128-bit block size would require the input plain text to be divisible by 128-bit blocks. If the plain text isn't divisible by this value, then the plain test should be padded until it is divisible by the block size with no remainder.

Simply put, we break the plain text into segments that equal the block size so that we can encrypt entire blocks of plain text at a time.

Examples of Block Ciphers:

Speck Cipher (Created by the NSA)

Simon Cipher (Created by the NSA)

AES

Stream:

Stream ciphers convert plain text into cipher text one byte at a time. This means that no padding is required for this type of cipher. Stream ciphers tend to be simpler than block ciphers.

Examples of Stream Ciphers:

Salsa20 (Created by Daniel J. Bernstein)

Linear-Feedback Shift Register (LFSR)

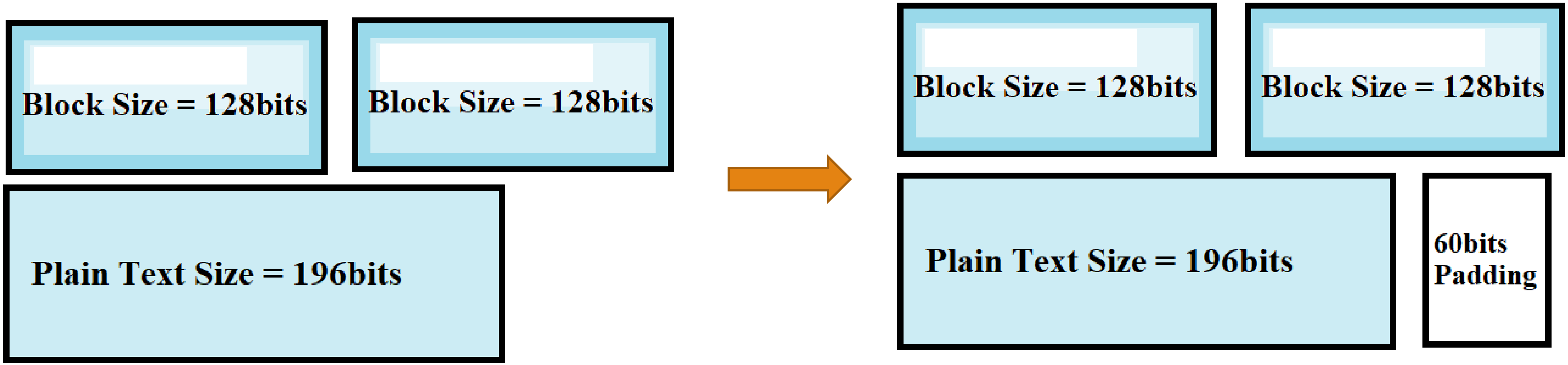
SEAL

Example: How to calculate padding for Block Cipher plain text

<u>**Assume 128bit Block Size and 196bit Plain Text Size**</u>

$$\left\lceil \frac{\text{Plain Text Size}}{\text{Block Size}} \right\rceil = \text{Blocks} \qquad \left\lceil \frac{196}{128} \right\rceil = 2 \text{ Blocks}$$

Blocks * Block Size - Plain Text Size = Padding Size
(2 Blocks * 128bits) - 196bits = 60bits



A few words of caution: Padding is a naïve approach to resolve plain text size issues with block ciphers. A common flaw in padding is that with sufficiently small plain text messages, the encrypted output can quickly inflate in size. For example, assume our block cipher uses 64bit blocks and our message come in 32bit chunks. This would cause the output of our cipher to be twice the size of the initial message. Another common issue is that padding makes block cipher more vulnerable to tampering attacks. Other more complex solution exist that can avoid the downfalls of padding such as ciphertext steal or residual block termination (but that is a topic for another time ☺).

Speck Cipher (Block Cipher)

The Speck Cipher is a family of lightweight block ciphers created by the NSA and released to the public in 2013. The Speck Cipher has been optimized for software whereas its sister algorithm the Simon Cipher is optimized for hardware implementations. While I personally have experience implementing both algorithms, I will only go over the Speck Cipher today as the Simon Cipher requires in-depth knowledge on register transfer level (RTL) code such VHDL or Verilog to demonstrate properly.

Block size (bits)	Key size (bits)	Rounds
2×16 = 32	4×16 = 64	22
2×24 = 48	3×24 = 72	22
	4×24 = 96	23
2×32 = 64	3×32 = 96	26
	4×32 = 128	27
2×48 = 96	2×48 = 96	28
	3×48 = 144	29
2×64 = 128	2×64 = 128	32
	3×64 = 192	33
	4×64 = 256	34

The chart to the left shows what block, key, and round sizes correspond to each other. For the example today I will be going over the version with 128-bit block size, 128-bit key size, and 32 rounds.

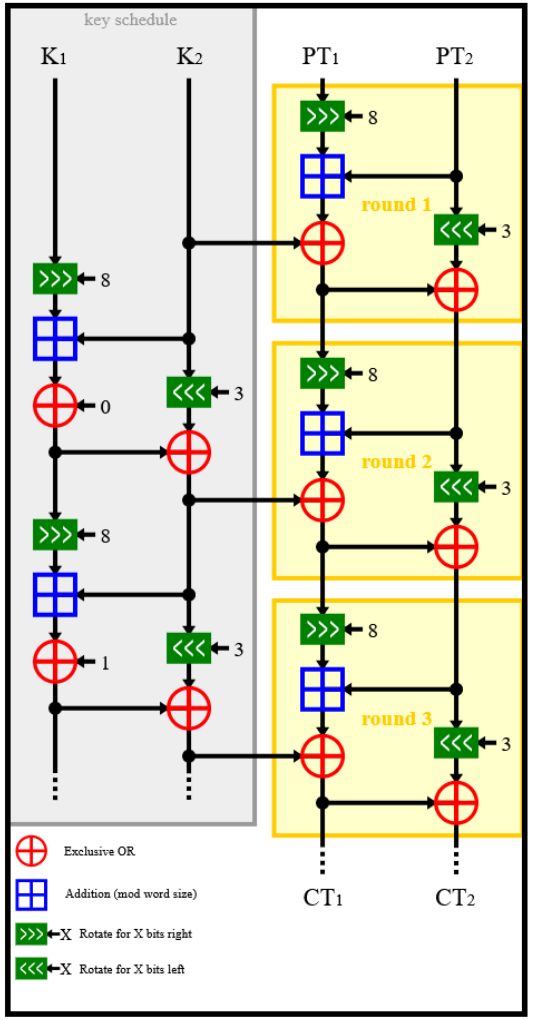
https://en.wikipedia.org/wiki/Speck_(cipher)

Speck Cipher Encryption and Decryption

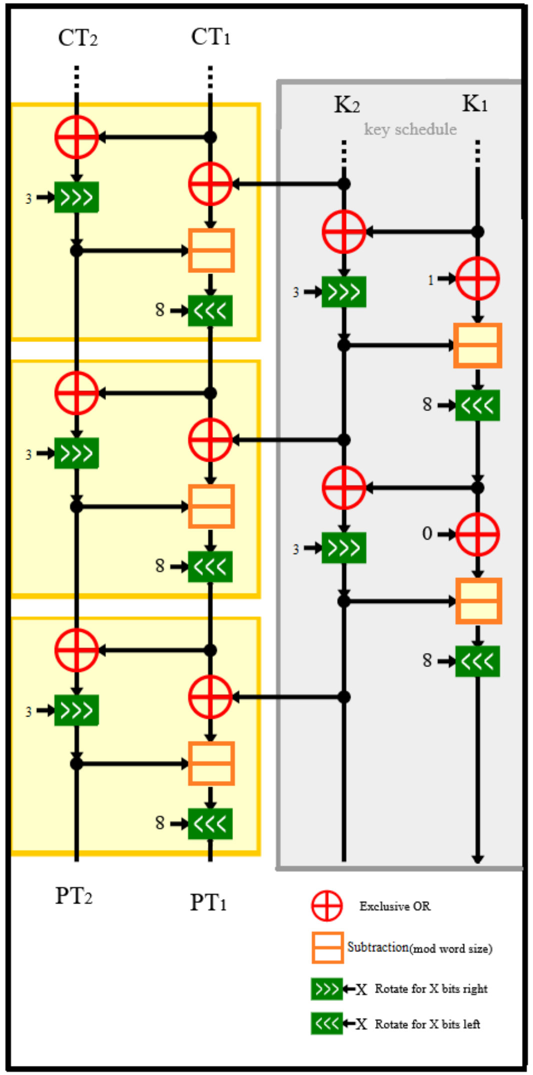
For a working example in C++ with both encryption and decryption use the GitHub link below. Please note that this example uses padding which is generally bad practice for block ciphers.

https://github.com/caleb1000/speck_cipher

Encryption



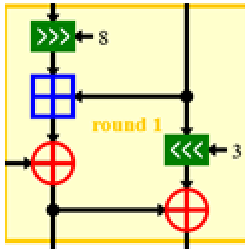
Decryption



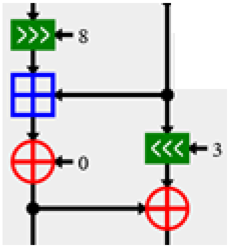

```
:~/speck$ ./speck "I hope you all are having fun!"
Original data ascii= I hope you all are having fun!
Original data Hex= 0x4920686f706520796f7520616c6c20617265206861766696e672066756e210000
Encrypted data Hex= 0xba084e7d169f6acd52e83f3b1f414158ea9c0c9fd76b952b3cdb7eb0558af382
Decrypted data Hex= 0x4920686f706520796f7520616c6c20617265206861766696e672066756e210000
Decrypted data Ascii= I hope you all are having fun!
```

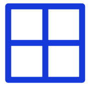
Bitwise operations:

Cipher Round



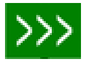
Key Rotation



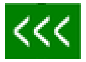


Addition: Adds two unsigned values together and outputs the sum

Subtraction: Subtracts an unsigned value from another unsigned value and outputs the result



Bitwise Rotate Right: Rotates an input value N number of bits to the right, looping the values shifted right to the front of the value.
(Note: this is different than simply shifting the number N times right)



Bitwise Rotate Left: Rotates an input value N number of bits to the left, looping the values shifted left to the end of the value.
(Note: this is different than simply shifting the number N times left)

<u>Example: Bitwise Rotate Right</u>

Value = 4 (binary: 0100)
N = 3
Step 0 (binary value): Value = 0100
Step 1 (shift right 1): Value = 0010
Step 2 (shift right 1): Value = 0001
Step 3 (shift right 1): Value = 1000

Speck Cipher Encryption

Below we have a function called encrypt that takes in a plain text block with a key which outputs the cipher text to the given array ct.

```
#include <stdint.h>

#define ROR(x, r) ((x >> r) | (x << (64 - r)))
#define ROL(x, r) ((x << r) | (x >> (64 - r)))
#define R(x, y, k) (x = ROR(x, 8), x += y, x ^= k, y = ROL(y, 3), y ^= x)
#define ROUNDS 32

void encrypt(uint64_t ct[2],
             uint64_t const pt[2],
             uint64_t const K[2])
{
   uint64_t y = pt[0], x = pt[1], b = K[0], a = K[1];

   R(x, y, b);
   for (int i = 0; i < ROUNDS - 1; i++) {
      R(a, b, i);
      R(x, y, b);
   }

   ct[0] = y;
   ct[1] = x;
}
```

Rotate right

Rotate left

Cipher round on plain text

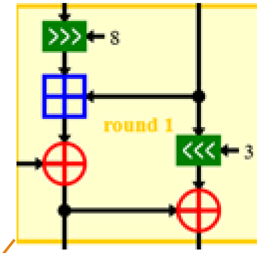
Key rotation

Cipher round on plain text

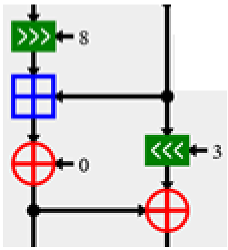
Upper 64-bits of cipher text

Lower 64-bits of cipher text

Cipher Round



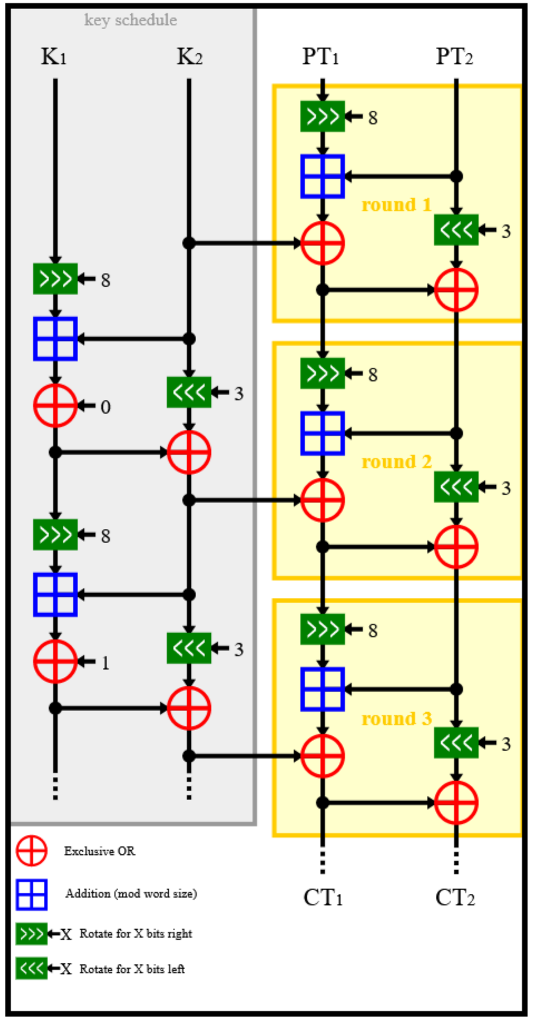
Key Rotation



Notice we can use the same function for both cipher rounds and key rotations!

The number of Key Rotations is equal to the number of Cipher Rounds – 1. For this example, we have 32 Rounds so we will have 31 Key rotations!

Encryption



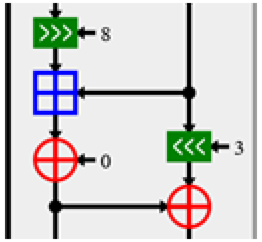
https://en.wikipedia.org/wiki/Speck_(cipher)

Speck Cipher Decryption

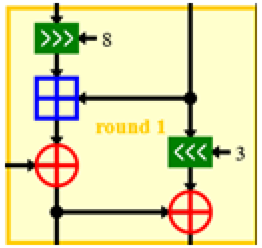
```
#define ROR(x, r) ((x >> r) | (x << (64 - r)))
#define ROL(x, r) ((x << r) | (x >> (64 - r)))
#define UR(x, y, k) (y ^= x, y = ROR(y, 3), x ^=k, x -= y, x = ROL(x,8))
#define ROUNDS 32

crypted_data decrypt(uint64_t const pt[2],
                     uint64_t const K[2])
{
   uint64_t y = pt[0], x = pt[1], b = K[0], a = K[1];

   for (int i = ROUNDS-2; i >= 0; i--) {
      UR(x, y, b); Cipher Round
      UR(a, b, i); Key Rotation
   }
   UR(x, y, b);  Cipher Round

   crypted_data result;
   result.key[0] = b;
   result.key[1] = a;
   result.data[0] = y;
   result.data[1] = x;
   return result;
}
```

For decryption we must invert the functions we used to encrypt the data. For example, the inverse of doing an addition operation is subtraction. Interestingly, the inverse of XOR is itself.

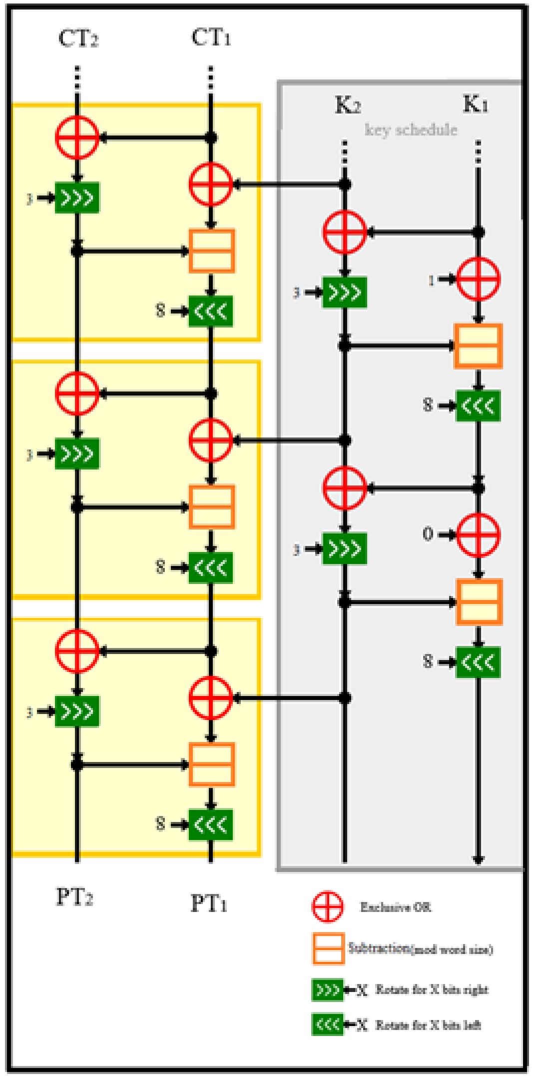
Key Rotation



Cipher Round



Decryption



LFSR [Linear-Feedback Shift Register] (Stream Cipher)

This cipher is known as a <u>Synchronous Stream Cipher</u>, meaning that to encrypt/decrypt the sender and receiver much transmit error free data that is in proper order. If the target bytes are out of order, then attempts to decrypt the message will fail.

Essentially, the current byte rely on the previous byte for the proper encryption/decryption. This means that even a small error (such as a single bit flip) can cause the remainder of the encrypted message to be indecipherable.

Can we get to an example please?…

LFSR Cipher Algorithm Pseudo Code

Using the following algorithm, we loop 8 times for each byte in the plain text (for example if the plain text is 4 bytes long, we will loop a total of 32 times). After every 8 loops we use the lowest byte of the state register S and XOR it with the current byte from the plain text. We continue this step until all bytes of the plain text have been encrypted.

Assume F is the value of the feedback, PT is the plain text, and S is the current value of the shift register, the next state of the cipher is calculated by the following steps:

```
for ( int x = 0 ; x<8 ; x++){
        //for each byte of the plain text we need to loop through this 8 times (steps)
        if ( 0x01 & S == 0){
                //If the current shift register's lowest bit is 0, simply shift the current shift register right one bit
                S = S>>1;
        }
        else{
        // If the current shift register's lowest bit is 1 simply shift the current shift register right by one bit and XOR it with the value of F
                S = (S>>1) ^ F
        }
}
uint8_t encrypted_byte = (S & 0xFF) ^  PT[index]; // XOR the current plain text byte with the value gotten from the loop. Repeat for every byte in the plain text
```

Example: Encrypt First Byte

Assume F = 0xDEADBEEF and the initial value of S = 0x12345678
Assume plain text is "caci" (In Hex -> 0x63, 0x61, 0x63, 0x69)

F = Feedback Register, PT =Plain Text, S = Shift Register

Step 1

Initial S = 00010010001101000101011001111000

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00001001000110100010101100111100

Step 2

Current S = 00001001000110100010101100111100

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00000100100011010001010110011110

Step 3

Current S = 00000100100011010001010110011110

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00000010010001101000101011001111

Step 4

Current S = 00000010010001101000101011001111

Because the lowest bit of S is one, we shift S right by one bit and XOR it with F

00000001001000110100010101100111 XOR 11011110101011011011111011101111

New S = 11011111100011101111101110001000

Step 5

Current S = 11011111100011101111101110001000

Because the lowest bit of S is zero, we shift S right by one bit

New S = 01101111110001110111110111000100

Step 6

Current S = 01101111110001110111110111000100

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00110111111000111011111011100010

Step 7

Current S = 00110111111000111011111011100010

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00011011111100011101111101110001

Step 8

Current S = 00011011111100011101111101110001

Because the lowest bit of S is one, we shift S right by one bit and XOR it with F

00001101111110001110111110111000 XOR 11011110101011011011111011101111

Final S = 11010011010101010101000101010111

<u>Assume F = 0xDEADBEEF and the initial value of S = 0x12345678</u>
<u>Assume plain text is “caci” (In Hex -> 0x63, 0x61, 0x63, 0x69)</u>

F = Feedback Register, PT =Plain Text, S = Shift Register

<u>After eight steps we get the following S value</u>

Final S = 11010011010101010101000101010111
Final S in Hex = 0xD3555157
Lowest S byte: S(0) = 0x57
Plain text = 0x63616369
First plain text byte: PT(0) = 0x63
S(0) XOR PT(0) = 0x34

The value 0x34 is the encrypted first byte of the cipher text message. This whole process is repeated for every byte of the plain text, in this case it is going to be repeated 3 more times.

The final encrypted value of the cipher text is 0x349162F0

```cpp
const uint32_t F = 0xDEADBEEF;
const uint32_t maskBit = 0x01;
const uint32_t maskByte = 0xFF;

unsigned char *cipher(unsigned char * data, int dataLength, uint32_t initialValue, bool ascii)
{
    unsigned char * result;
    result = (unsigned char*)malloc(dataLength*sizeof(unsigned char));
    uint32_t S = initialValue;

    // loops each character
    for(int x = 0; x < dataLength; x++)
    {

        for(int y = 0; y < 8; y++)
        {
            // loops 8 times to create key
            if(S & maskBit)
            {
                // lowest bit of S is 1
                S = (S >> 1) ^ F;
            }
            else
            {
                // lowest bit of S is 0
                S = (S >> 1);
            }
        }
        // get first key byte of S
        uint32_t lowestByte = S & maskByte;
        // xor key byte with character
        uint32_t outputChar = lowestByte ^ data[x];
        result[x] = outputChar;
        if(ascii){
            printf("%c",outputChar);
        }
        else{
            printf("%02x",outputChar);
        }
    }
    printf("\n");
    return result;

}
```

C++ Example of a Linear-Feedback Shift Register Algorithm

On the left is an example of a simple LFSR algorithm. Even with this meager amount of code we can encrypt and decrypt messages of any size. Because of this, Linear-Feedback Shift Registers are often used in ASICs (Application-Specific Integrated Circuit) where simplicity allows for cheaper hardware designs and lower circuit complexity. Furthermore, this cipher is the basis of many other stream ciphers.

Decryption for this algorithm is incredibly simple. Unlike block ciphers, we can simply use the same function for both encryption and decryption. By feeding the algorithm the cipher text and the same initial seed we can decrypt the plain text.

https://github.com/caleb1000/jpeg_secret_message/blob/main/cipher.cpp

Example: Decrypt First Byte

Assume F = 0xDEADBEEF and the initial value of S = 0x12345678
Assume cipher text is 0x349162F0

F = Feedback Register, CT = Cipher Text, S = Shift Register

Step 1

Initial S = 00010010001101000101011001111000

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00001001000110100010101100111100

Step 2

Current S = 00001001000110100010101100111100

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00000100100011010001010110011110

Step 3

Current S = 00000100100011010001010110011110

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00000010010001101000101011001111

Step 4

Current S = 00000010010001101000101011001111

Because the lowest bit of S is one, we shift S right by one bit and XOR it with F

00000001001000110100010101100111 XOR 11011110101011011011111011101111

New S = 11011111100011101111101110001000

Step 5

Current S = 11011111100011101111101110001000

Because the lowest bit of S is zero, we shift S right by one bit

New S = 01101111110001110111110111000100

Step 6

Current S = 01101111110001110111110111000100

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00110111111000111011111011100010

Step 7

Current S = 00110111111000111011111011100010

Because the lowest bit of S is zero, we shift S right by one bit

New S = 00011011111100011101111101110001

Step 8

Current S = 00011011111100011101111101110001

Because the lowest bit of S is one, we shift S right by one bit and XOR it with F

00001101111110001110111110111000 XOR 11011110101011011011111011101111

Final S = 11010011010101010101000101010111

Assume F = 0xDEADBEEF and the initial value of S = 0x12345678
Assume cipher text is 0x349162F0

F = Feedback Register, CT =Cipher Text, S = Shift Register

After eight steps we get the following S value

Final S = 11010011010101010101000101010111
Final S in Hex = 0xD3555157
Lowest S byte: S(0) = 0x57
Cipher text = 0x349162F0
First cipher text byte: CT(0) = 0x34
S(0) XOR CT(0) = 0x63

The value 0x63 is the decrypted first byte of the plain text message. This whole process is repeated for every byte of the plain text, in this case it is going to be repeated 3 more times.

The final decrypted value of the plain text is 0x63616369
Which in ASCII is “caci”!