CSE 470 HW Report

Abdullah Çelik
171044002

Research

Lightweight Cryptography Primitives

Lightweight cryptography is primarily targeting the highly constrained devices that are in the low-end of the device spectrum, such as embedded systems and RFID and Sensor Networks. While lightweight cryptography primarily targets the low end of the device spectrum, it is important to note that it may be necessary to implement lightweight algorithms at the high end of the spectrum as well for the sake of other receiver devices can directly interact with the data without a need of any other decoder hardware.

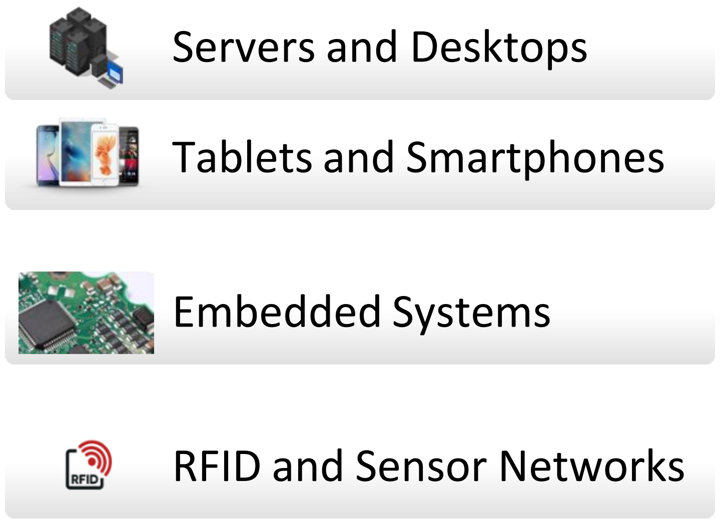


Figure 1: Device Spectrum from high-end to low-end

Lightweight Block Ciphers

Overview

Some lightweight block ciphers are proposed to achieve performance advantages over AES, particularly AES-128. Some of them were designed by simplifying conventional, well-analyzed block ciphers to improve their efficiency (like DESLX) and some of them were dedicated block ciphers that were designed from scratch (like SPECK).
The performance benefits of lightweight block ciphers over conventional block ciphers are achieved using lightweight design choices. Such as:

- Smaller Block Sizes
  AES-128 uses 128-bit block size. To save memory, lightweight block ciphers can use smaller blocks. But reducing the block size reduces the limit of maximum plaintext blocks to be encrypted.
- Smaller Key Sizes
- Simpler Rounds
  Components and the operations are simpler in the lightweight block ciphers. 4-bit S-Boxes are preferred over 8-bit S-Boxes because of the constraints. When the rounds are simpler, iteration count may be bigger to achieve security
- Simpler Key Schedules

Complex key schedules increases the required memory, latency and power consumption. Most lightweight block ciphers use simple key schedules (or not use any) that can generate round keys easily.

- Minimal Implementations

We can separate existing lightweight block ciphers in two groups. Substitution-Permutation Network and Fiestel Networks.

Substitution-Permutation Network (SPN)

This structure comes from the idea of Shannon to provide both confusion and diffusion using two distinct operations. Confusion means that each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two and hiding the relationship between the ciphertext and the key. Diffusion means that if we change a single bit of the plaintext, then about half of the bits in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then about half of the plaintext bits should change. Confusion is performed by a layer named S-Box (Substitution Box). Diffusion is performed by a layer named P-Box (Permutation Box). An example for this structure is Rijndael, the current AES. Since it has been standardized, other algorithms are influenced by this. We can group these algorithms as AES-like. But there are other algorithms that not similar to AES. Here is a list of lightweight block ciphers that uses SPN.

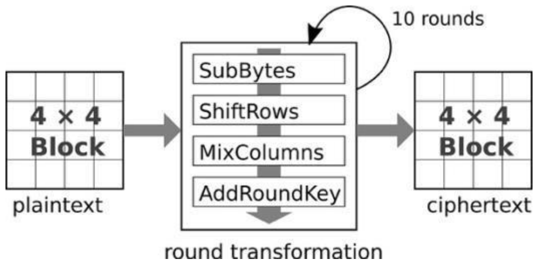
AES-Like

These are the algorithms having a structure derived from AES. Some examples for AES-Like SDN ciphers:

AES

The AES consist of 3 versions of the Rijndael cipher, AES-128, AES-196 and AES-256. The numbers correspond to the key size. The internal state is always 128 bits and organized as a 4x4 matrix of bytes. AES-128 is used for lightweight cryptography. Encryption consists of these steps:

- SubBytes: S-Box layer
- ShiftRows: P-Box(diffusion) layer for rows
- MixColumns: P-Box(diffusion) layer for columns
- AddRoundKey



I didn't find it necessary to explain the steps since AES is well-known.

Klein

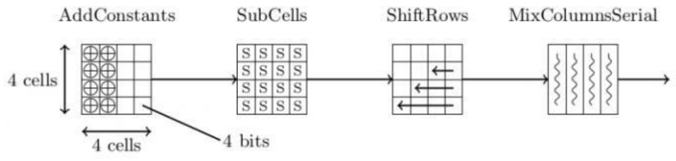
Klein consist of 3 version, Klein64, Klein80 and Klein96. The numbers correspond to the key size. All versions have 64-bit fixed block size. Klein64 is recommended for hash and mac and other versions are recommended for encryption in any operation modes. The state is organized as array of 4-bit nibbles(total 16 nibbles). Encryption consists of these steps:

- AddRoundKey: Each round key is generated with a Fiestel network. At each step all state is xored with round key.
- SubNibbles: All 16 nibbles are substituted using the same 4-bit S-Box for all nibbles.
- RotateNibbles: All nibbles rotated 2 bytes left per round.
- MixNibbles: Nibbles are divided into 2 tuples, and there are proceeded the same as MixColumns step of Rijndael

LED

Have 64-bit block size. The state is organized as 4x4 matrix of 4-bit nibbles. It uses 64-bit or 128-bit keys. One step consists of 4-round. One round consists of these steps:

- AddConstants: State is xored with a constant.
- SubCells: All 16 nibbles are substituted using the same 4-bit S-Box for all nibbles.
- ShiftRows: Same as Rijndael's ShiftRows
- MixColumnsSerial: Same as Rijndael's MixColumns except it uses special matrix called MDS matrix



Before each step round key is xored with the state. For 64-bit key round key is the same in all rounds. For 128-bit key round key is first half of the key in odd rounds and last half of the key in even rounds.

Others

Hardware Targeting Ciphers: Midori, SKINNY
Software Targeting Ciphers: Mysterion, Zorro

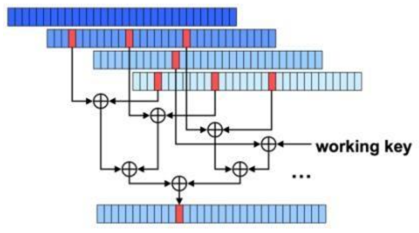
Bit-Sliced S-Boxes

Bit-sliced S-box is a non-linear layer consisting in the parallel application of several small non-linear functions which is supposed to be implemented using basic bitwise operations such as XOR, AND and OR. Some examples:

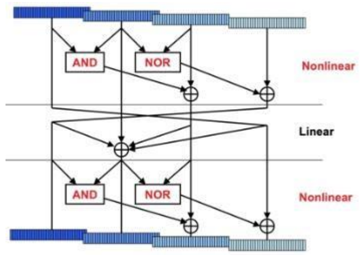
Noekeon

Has 128-bit block size and 128-bit key size. The state is organized as 4 32-bit words except for the S-Box layer called Gamma. Encryption consists of 16 rounds. The same round key is used in every round. One round consists of these steps:

- AddConstant: All words are xored with a constant
- Theta: Diffusion and key addition, illustrated in figure
- Permutation1: Shift words left by 1, 5, 2, 0 bits respectively.
- Gamma: Non-linearity, illustrated in figure
- Permutation2: Shift words right by 1, 5, 2, 0 bits respectively.



Theta



Gamma

Rectangle

Has 64-bit block size. The state is organized as 4x16 matrix. It is a 25 round SPN cipher. Each round consist of these steps:
AddRoundKey: State is xored with round key
SubColumn: Substitute each column using the same 4-bit S-Box. For example the first column consist of state[15], state[31], state[47], state[63] bits.
ShiftRow: Left rotate all rows by 0, 1, 12, 13 bits respectively.

Fiestel Networks

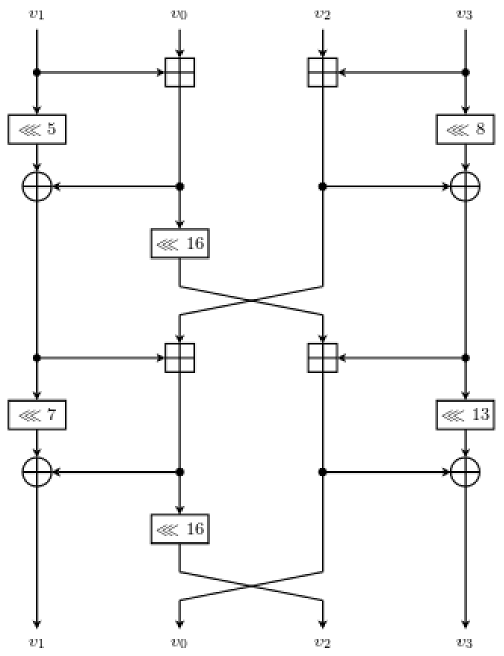
A fiestel network works by splitting data blocks into two equal pieces and applying encryption in multiple rounds. Each round implements permutation and combinations derived from primary function or key.

2.1- ARX-Based

ARX-based ciphers are only use modular Addition, Rotation and XOR. They are usually faster and smaller than S-Box-Based algorithms in software. But in hardware they tend to have more latency. Some examples for ARX-Based cipher that uses Fiestel Networks:

Chaskey Cipher

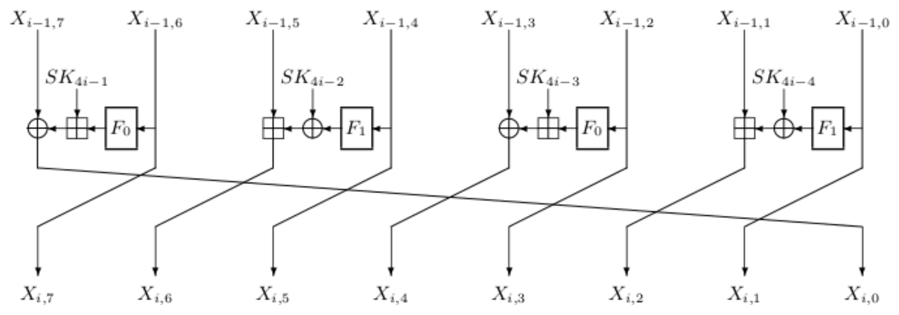
It is a 128-bit block cipher. Used in Mac algorithm called Chaskey. It has no key schedule. It is a very simple cipher and a round is represented as:



The cipher originally consists of 8 rounds but for more security there is a version called Chaskey-LTS which has 16 round and Chaskey-12 which has 12 rounds.

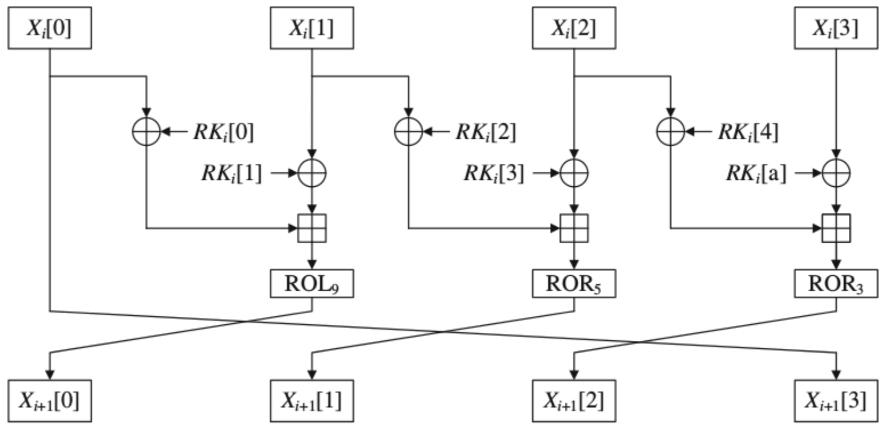
HIGHT

At every round key schedule generates an 8-byte whitening keys by selecting some bytes of the master key in a complex way. The round function is represented as:



LEA

It is a 128-bit block cipher. The key schedule is also ARX-based. For each round a contant added to the key state and then rotated. The round function is represented as:



The key is added in both datapath going in each addition.

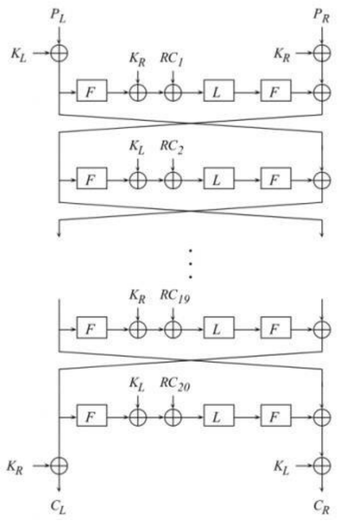
2.2- Two Branched

These ciphers operate with the blocks of size 2n where the Fiestel function maps n bits to nbits. Some examples:

DESLX

This is a modified version of DES. The round function uses a single S-Box rather than 8 different S-Boxes and omits the initial and final permutations to improve the size of the hardware implementation.

ITUbee



Uses 20 rounds of Fiestel structure. The whole encryption is represented in the figure.
L is a multiplication by a 5x5 matrix, operating on bytes.
F(x) is equals to S(L(S(x))) where S is a substition layer uses the S-Box of AES.
There is a key whitening at the beginning and the end of the encryption.

Others

Hardware Targeting Ciphers: GOST Revisited, KASUMI,
Software Targeting Ciphers: RoadRunneR
Hardware and Software Targeting Ciphers: MISTY, LBlock, SEA

2.3- Generalized Fiestel Networks(GFN)

CLEFIA, Piccolo and TWINE

2.4- Other Designs

KTANTAN and KATAN

Lightweight Hash Functions

Since conventional hash functions have high power consumption and their internal state sizes are larger, they can be not suitable for low end of the device spectrum. Because of that there is a need of lightweight hash functions like PHOTON, Quark, SPONGENT and Lesamnta-LW.
The performance benefits of lightweight hash functions over conventional hash functions are achieved using lightweight design choices. Such as:

- Smaller Internal State and Output Size
  Large output sizes are required for collision resistance. But for the applications that are not require collision resistance, smaller internal state and output sizes can be used.
- Smaller Message Size
  For most of the target protocols for lightweight hash functions has small input size (like at most 256 bit). So, supporting 2^{64}-bit input is not required. Therefore, functions optimized for short inputs are more suitable.

Lightweight MACS

For typical applications, recommended tag sizes are at least 64 bits. For certain applications, occasionally accepting an inauthentic message may have small effect on the security (like VoIP), so shorter tags can be used. Chaskey, TuLP and LightMAC are some of the lightweight MAC examples.

Lightweight Stream Ciphers

For constrained environments stream ciphers are also promising primitives. Grain, Trivium and Mickey are the 3 finalist stream ciphers for hardware applications with restricted resources.

NIST Approved Cryptographic Primitives in Constrained Environments

Block Ciphers

There are two block ciphers that are approved by NIST: AES and Triple DES.
For lightweight cryptography, AES-128 is the most suitable variant of AES. And on certain 8-bit microcontrollers, AES performs very well. For applications where the performance of AES is acceptable, AES should be used.

Hash Functions

There are two FIPS documents that specifies hash standards, FIPS 180-4 and FIPS 202. FIPS 180-4 specifies SHA-1 and SHA-2 family, FIPS 202 specifies SHA-3 family. None of the variants of these families are suitable for constrained environments because of their large internal-state size requirements.

Authenticated Encryption Algorithms and MACs

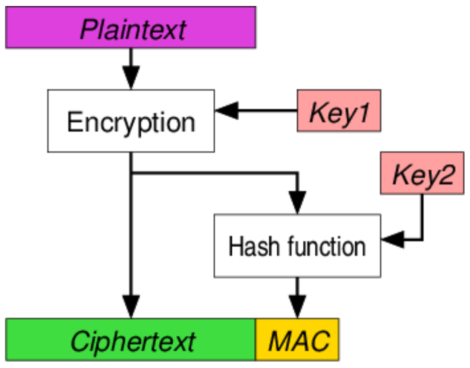
NIST approves CCM and GCM modes that provides authentication and encryption. NIST also approves standalone Macs, CMAC, GMAC and HMAC to be used for generating and verifying tags to provide authentication.

Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption is an encryption which simultaneously assure the confidentiality and authenticity of data.
AEAD is a variant of AE (Authenticated Encryption) that allows a recipient to check the integrity of both the encrypted and unencrypted information. AEAD binds the AD (Associated Data) with cipher, so in order to validate the message receiver needs to same AD too.
There are 3 main approaches for AEAD.

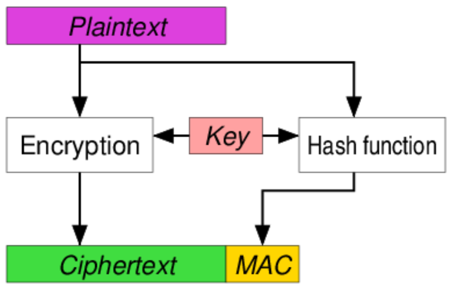
Encrypt-then-MAC (EtM)

This is the only method that can achieve maximum security in AE, but Hash function must be strongly unforgeable. The high-level schema is represented in the figure.



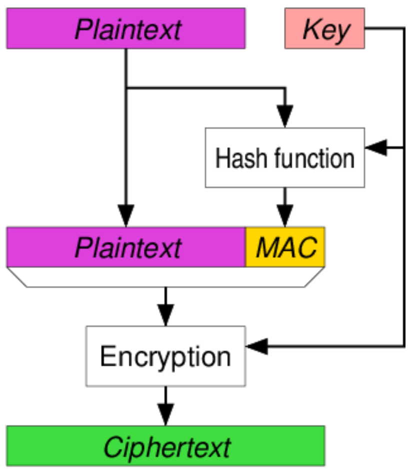
Encrypt-and-MAC (E&M)

The high-level schema is represented in the figure.



MAC-then-Encrypt (MtE)

It is used in SSL/TLS. The high-level schema is represented in the figure.



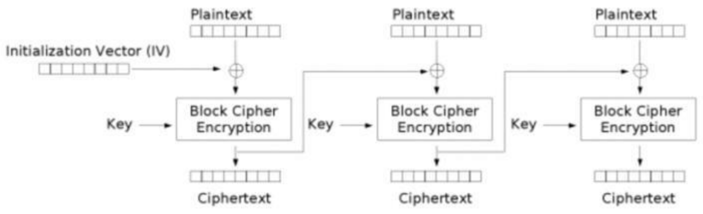
ISAP

ISAP is a nonce-based Encrypt-then-Mac AEAD scheme family. Encryption is done by XORing the message and a keystream, and the authentication/verification is based on hash-then-MAC paradigm.

CBC

Encryption

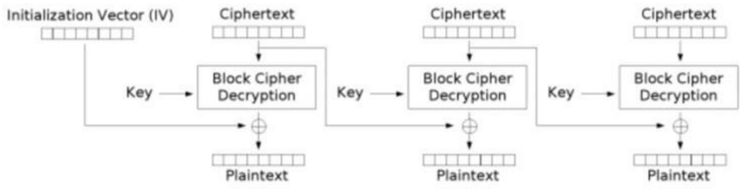
I use 2 buffers named input and message_buffer.
For first step I set input to IV. After that I xor the input with message_buffer. Message_buffer is almost all the time equals to plain text but in the last round remaining plain text's length may be smaller than the block size. For that case I pad the message_buffer using PKCS5 method (explained in here). After xoring the input, I call my encryption method (not the AEAD encrypt) and I store the result of this encryption to input buffer. Then I repeat these steps until all the plaintext is encrypted.



Cipher Block Chaining (CBC) mode encryption

Decryption

I use only 1 buffer named input. The reason that I don't use message buffer is because the cipher is always having BLOCK_SIZE * n length. So, there will be no padding during decryption. For first step I set input to IV. After that I call my decryption algorithm. The result is written to the plain text. After that I xor the plaintext with input buffer, then I set the input buffer to this round's ciphertext. Then I repeat these steps until all the ciphertext is decrypted. After decryption, since I pad the plaintext in encryption, I am removing the padding if it exists.

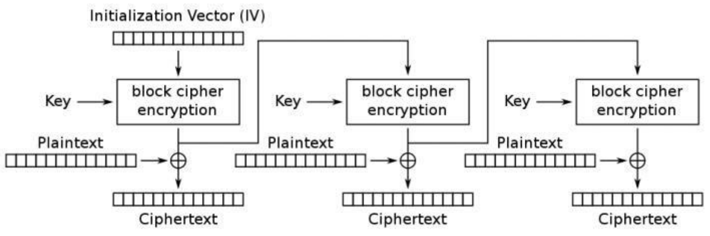


Cipher Block Chaining (CBC) mode decryption

OFB

Encryption

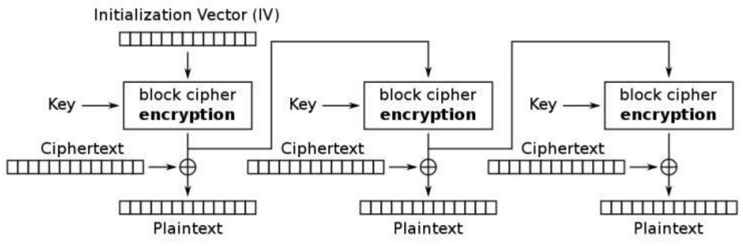
I use 2 buffers named input and message_buffer. For the first step I set input to IV. Then I encrypt the input buffer and save the result to ciphertext. then copy ciphertext to input. After that I take the plaintext to message_buffer and pad it if needed. Then xor the ciphertext with the message_buffer. Then I repeat these steps until all the plaintext is encrypted.



Output Feedback (OFB) mode encryption

Decryption

I use only one buffer name input because of the same reason I explained in the CBC decryption. The decryption of the OFB is almost the same as encryption. The only difference is there is no padding because the ciphertext must have BLOCK_SIZE*n length. After decryption I unpad the result with the same way in the CBC decryption.



Output Feedback (OFB) mode decryption

ISAP

Encrypt

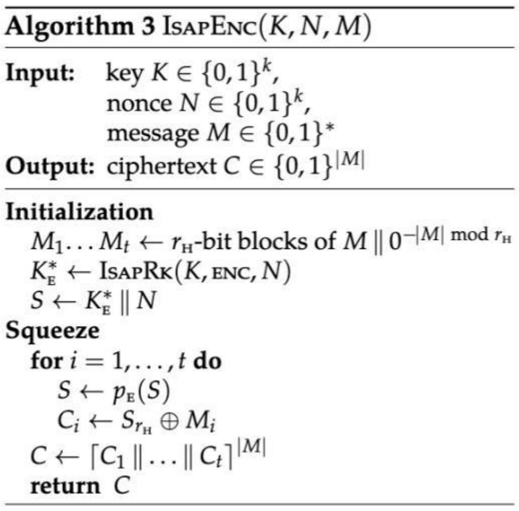
Since ISAP uses EtM(Encrypt then MAC) scheme, the encrypt function first encrypts the plaintext using isap_enc. Then creates a tag with the cipher text using isap_mac and appends the tag to the end of the cipher.

Decrypt

Decrypt method first calculates the tag without the tag part of the ciphertext then compares it with the tag part of the ciphertext. If they are not the same no decryption is done, and function returns an error. Otherwise, if they are the same, the authentication passes and decryption is done by using isap_enc since isap_enc is a symmetric block cipher and decryption is the same as encryption.

Isap_enc

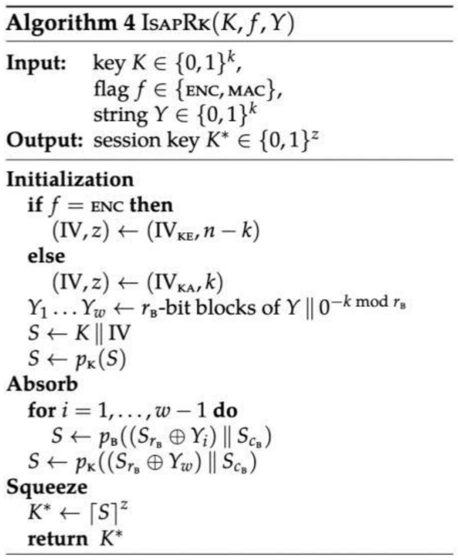
First a session key is generated using isap_rk and save it to state.
For each byte of the message, until the all bytes of the generated key is used, xor the message with the key and save it as cipher. If all bytes of the generated key is used, generate new key by permuting the current key.
Permuting is done by ascon-p.



```
Algorithm 3 IsapEnc$(K, N, M)$
Input:  key $K \in \{0,1\}^k$,
        nonce $N \in \{0,1\}^k$,
        message $M \in \{0,1\}^*$
Output: ciphertext $C \in \{0,1\}^{|M|}$
Initialization
  $M_1 \ldots M_t \leftarrow r_H$-bit blocks of $M \| 0^{-|M| \bmod r_H}$
  $K_E^* \leftarrow$ IsapRk$(K, \text{ENC}, N)$
  $S \leftarrow K_E^* \| N$
Squeeze
  for $i = 1, \ldots, t$ do
    $S \leftarrow p_E(S)$
    $C_i \leftarrow S_{r_H} \oplus M_i$
  $C \leftarrow \lceil C_1 \| \ldots \| C_t \rceil^{|M|}$
  return $C$
```

Isap_rk

For mac and enc, different IV is used. IV constants are specified in the specification. State is set to key || iv || 0 then permuted with a specific number of rounds. Then all string is absorbed and permuted by a specified way (I copied the way from the given algorithm in the specifications). Since Rb is 1 for all 4 versions of isap, concatenating S[0] ^Y with S[1:] is the same as doing S[0] ^= Y in C language.

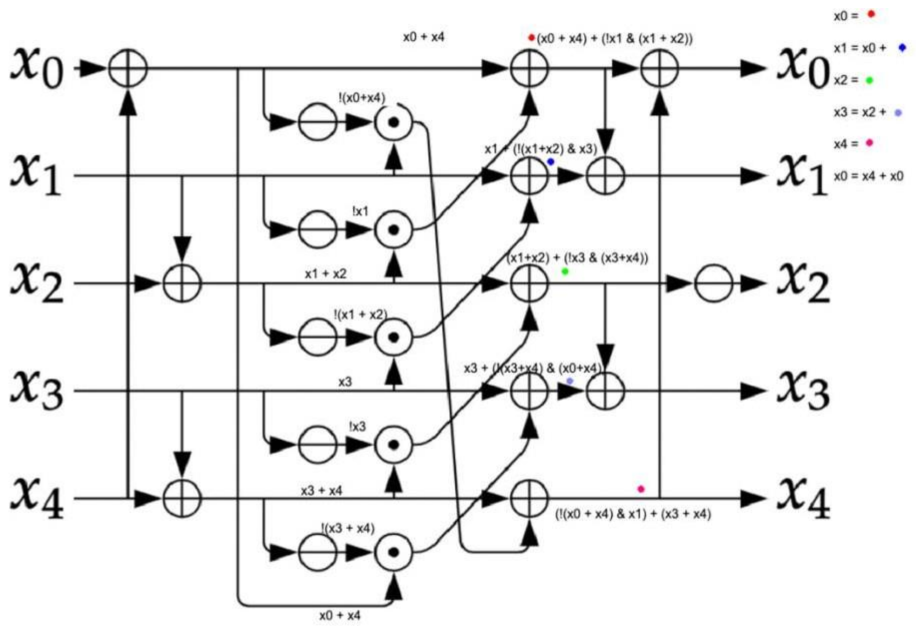


```
Algorithm 4 IsapRk$(K, f, Y)$
Input:  key $K \in \{0,1\}^k$,
        flag $f \in \{\text{ENC}, \text{MAC}\}$,
        string $Y \in \{0,1\}^k$
Output: session key $K^* \in \{0,1\}^z$
Initialization
  if $f = \text{ENC}$ then
    $(\text{IV}, z) \leftarrow (\text{IV}_{\text{KE}}, n - k)$
  else
    $(\text{IV}, z) \leftarrow (\text{IV}_{\text{KA}}, k)$
  $Y_1 \dots Y_w \leftarrow r_B$-bit blocks of $Y \| 0^{-k \bmod r_B}$
  $S \leftarrow K \| \text{IV}$
  $S \leftarrow p_K(S)$
Absorb
  for $i = 1, \dots, w-1$ do
    $S \leftarrow p_B((S_{r_B} \oplus Y_i) \| S_{c_B})$
  $S \leftarrow p_K((S_{r_B} \oplus Y_w) \| S_{c_B})$
Squeeze
  $K^* \leftarrow \lceil S \rceil^z$
  return $K^*$
```

Since all the other functions are well specified mathematical functions, I don't now what to explain about them. I implemented all the algorithms that are given in the specification file of the algorithm. My implementations are not optimized for a specific CPU architecture.

Permutation

ISAP is instantiated with using keccak or ascon-p permutation. I implement the ascon-p because it was simpler and clean for me. These are the calculations that I did for the S-Box of the Ascon. The schema is from the ascon specification.



And these are for the linear layer for ascon permutation. It was very straightforward to implement.

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$$

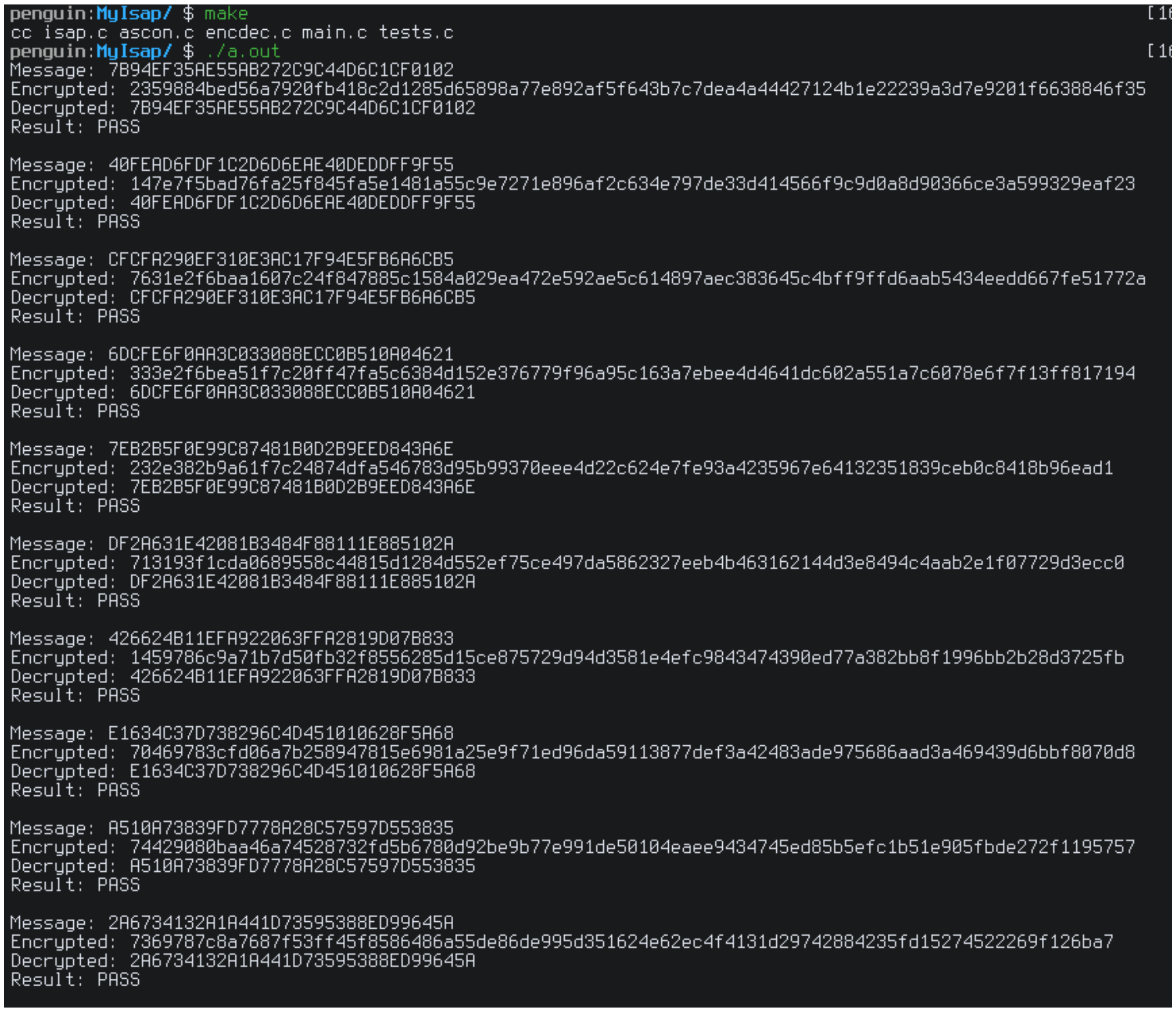
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$$

$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$$

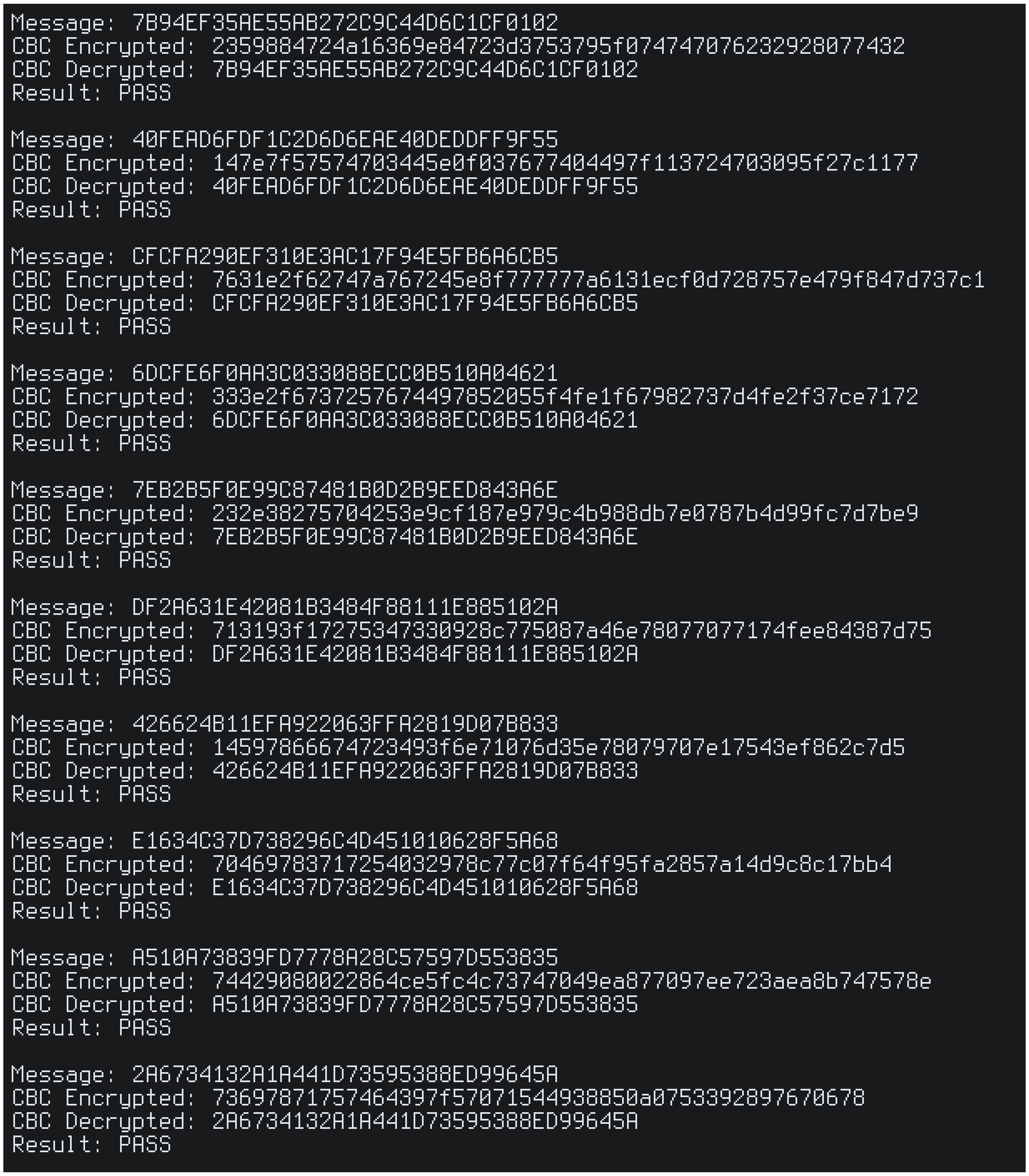
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$$

$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$$

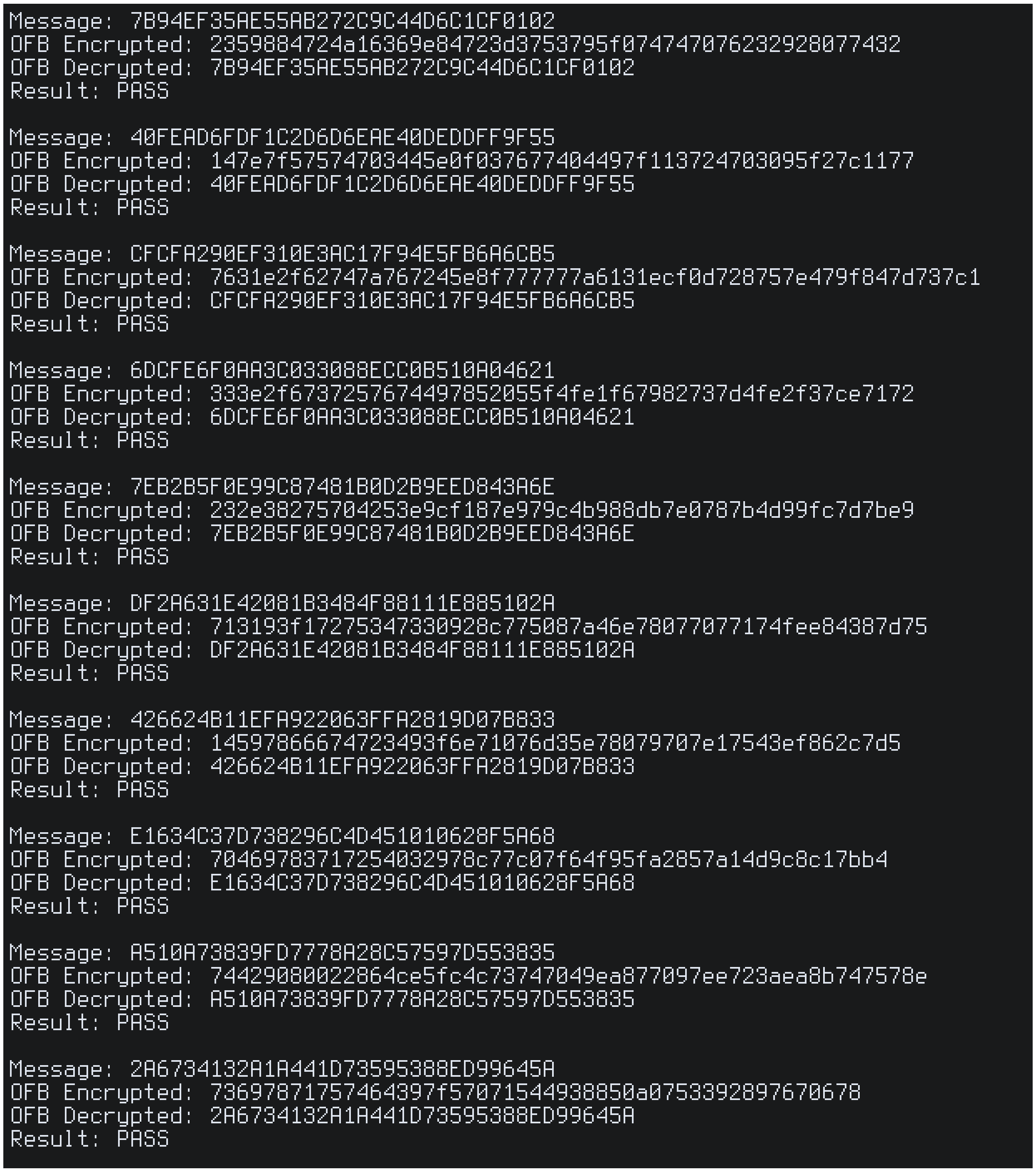
ISAP Test Results:



```
penguin:MyIsap/ $ make                                                                                    [1
cc isap.c ascon.c encdec.c main.c tests.c
penguin:MyIsap/ $ ./a.out                                                                                 [1
Message:  7B94EF35AE55AB272C9C44D6C1CF0102
Encrypted:  2359884bed56a7920fb418c2d1285d65898a77e892af5f643b7c7dea4a44427124b1e22239a3d7e9201f6638846f35
Decrypted:  7B94EF35AE55AB272C9C44D6C1CF0102
Result:  PASS

Message:  40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Encrypted:  147e7f5bad76fa25f845fa5e1481a55c9e7271e896af2c634e797de33d414566f9c9d0a8d90366ce3a599329eaf23
Decrypted:  40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Result:  PASS

Message:  CFCFA290EF310E3AC17F94E5FB6A6CB5
Encrypted:  7631e2f6baa1607c24f847885c1584a029ea472e592ae5c614897aec383645c4bff9ffd6aab5434eedd667fe51772a
Decrypted:  CFCFA290EF310E3AC17F94E5FB6A6CB5
Result:  PASS

Message:  6DCFE6F0AA3C033088ECC0B510A04621
Encrypted:  333e2f6bea51f7c20ff47fa5c6384d152e376779f96a95c163a7ebee4d4641dc602a551a7c6078e6f7f13ff817194
Decrypted:  6DCFE6F0AA3C033088ECC0B510A04621
Result:  PASS

Message:  7EB2B5F0E99C87481B0D2B9EED843A6E
Encrypted:  232e382b9a61f7c24874dfa546783d95b99370eee4d22c624e7fe93a4235967e64132351839ceb0c8418b96ead1
Decrypted:  7EB2B5F0E99C87481B0D2B9EED843A6E
Result:  PASS

Message:  DF2A631E42081B3484F88111E885102A
Encrypted:  713193f1cda0689558c44815d1284d552ef75ce497da5862327eeb4b463162144d3e8494c4aab2e1f07729d3ecc0
Decrypted:  DF2A631E42081B3484F88111E885102A
Result:  PASS

Message:  426624B11EFA922063FFA2819D07B833
Encrypted:  1459786c9a71b7d50fb32f8556285d15ce875729d94d3581e4efc9843474390ed77a382bb8f1996bb2b28d3725fb
Decrypted:  426624B11EFA922063FFA2819D07B833
Result:  PASS

Message:  E1634C37D738296C4D451010628F5A68
Encrypted:  70469783cfd06a7b258947815e6981a25e9f71ed96da59113877def3a42483ade975686aad3a469439d6bbf8070d8
Decrypted:  E1634C37D738296C4D451010628F5A68
Result:  PASS

Message:  A510A73839FD7778A28C57597D553835
Encrypted:  74429080baa46a74528732fd5b6780d92be9b77e991de50104eaee9434745ed85b5efc1b51e905fbde272f1195757
Decrypted:  A510A73839FD7778A28C57597D553835
Result:  PASS

Message:  2A6734132A1A441D73595388ED99645A
Encrypted:  7369787c8a7687f53ff45f8586486a55de86de995d351624e62ec4f4131d29742884235fd15274522269f126ba7
Decrypted:  2A6734132A1A441D73595388ED99645A
Result:  PASS
```



```
Message: 7B94EF35AE55AB272C9C44D6C1CF0102
CBC Encrypted: 2359884724a16369e84723d3753795f0747470762329280774 32
CBC Decrypted: 7B94EF35AE55AB272C9C44D6C1CF0102
Result: PASS

Message: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
CBC Encrypted: 147e7f57574703445e0f037677404497f113724703095f27c1177
CBC Decrypted: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Result: PASS

Message: CFCFA290EF310E3AC17F94E5FB6A6CB5
CBC Encrypted: 7631e2f62747a767245e8f777777a6131ecf0d728757e479f847d737c1
CBC Decrypted: CFCFA290EF310E3AC17F94E5FB6A6CB5
Result: PASS

Message: 6DCFE6F0AA3C033088ECC0B510A04621
CBC Encrypted: 333e2f67372576744978520 55f4fe1f67982737d4fe2f37ce7172
CBC Decrypted: 6DCFE6F0AA3C033088ECC0B510A04621
Result: PASS

Message: 7EB2B5F0E99C87481B0D2B9EED843A6E
CBC Encrypted: 232e38275704253e9cf187e979c4b988db7e0787b4d99fc7d7be9
CBC Decrypted: 7EB2B5F0E99C87481B0D2B9EED843A6E
Result: PASS

Message: DF2A631E42081B3484F88111E885102A
CBC Encrypted: 713193f17275347330928c775087a46e78077077174fee84387d75
CBC Decrypted: DF2A631E42081B3484F88111E885102A
Result: PASS

Message: 426624B11EFA922063FFA2819D07B833
CBC Encrypted: 14597866674723493f6e71076d35e78079707e17543ef862c7d5
CBC Decrypted: 426624B11EFA922063FFA2819D07B833
Result: PASS

Message: E1634C37D738296C4D451010628F5A68
CBC Encrypted: 70469783717254032978c77c07f64f95fa2857a14d9c8c17bb4
CBC Decrypted: E1634C37D738296C4D451010628F5A68
Result: PASS

Message: A510A73839FD7778A28C57597D553835
CBC Encrypted: 74429080022864ce5fc4c73747049ea877097ee723aea8b747578e
CBC Decrypted: A510A73839FD7778A28C57597D553835
Result: PASS

Message: 2A6734132A1A441D73595388ED99645A
CBC Encrypted: 7369787175746439 7f57071544938850a0753392897670678
CBC Decrypted: 2A6734132A1A441D73595388ED99645A
Result: PASS
```



```
Message: 7B94EF35AE55AB272C9C44D6C1CF0102
OFB Encrypted: 2359884724a16369e84723d3753795f074747076232928077432
OFB Decrypted: 7B94EF35AE55AB272C9C44D6C1CF0102
Result: PASS

Message: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
OFB Encrypted: 147e7f57574703445e0f037677404497f113724703095f27c1177
OFB Decrypted: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Result: PASS

Message: CFCFA290EF310E3AC17F94E5FB6A6CB5
OFB Encrypted: 7631e2f62747a767245e8f777777a6131ecf0d728757e479f847d737c1
OFB Decrypted: CFCFA290EF310E3AC17F94E5FB6A6CB5
Result: PASS

Message: 6DCFE6F0AA3C033088ECC0B510A04621
OFB Encrypted: 333e2f6737257674497852055f4fe1f67982737d4fe2f37ce7172
OFB Decrypted: 6DCFE6F0AA3C033088ECC0B510A04621
Result: PASS

Message: 7EB2B5F0E99C87481B0D2B9EED843A6E
OFB Encrypted: 232e38275704253e9cf187e979c4b988db7e0787b4d99fc7d7be9
OFB Decrypted: 7EB2B5F0E99C87481B0D2B9EED843A6E
Result: PASS

Message: DF2A631E42081B3484F88111E885102A
OFB Encrypted: 713193f17275347330928c775087a46e78077077174fee84387d75
OFB Decrypted: DF2A631E42081B3484F88111E885102A
Result: PASS

Message: 426624B11EFA922063FFA2819D07B833
OFB Encrypted: 14597866674723493f6e71076d35e78079707e17543ef862c7d5
OFB Decrypted: 426624B11EFA922063FFA2819D07B833
Result: PASS

Message: E1634C37D738296C4D451010628F5A68
OFB Encrypted: 70469783717254032978c77c07f64f95fa2857a14d9c8c17bb4
OFB Decrypted: E1634C37D738296C4D451010628F5A68
Result: PASS

Message: A510A73839FD7778A28C57597D553835
OFB Encrypted: 74429080022864ce5fc4c73747049ea877097ee723aea8b747578e
OFB Decrypted: A510A73839FD7778A28C57597D553835
Result: PASS

Message: 2A6734132A1A441D73595388ED99645A
OFB Encrypted: 73697871757464397f57071544938850a07533928976706 78
OFB Decrypted: 2A6734132A1A441D73595388ED99645A
Result: PASS
```

Grain-128AEAD

The Grain consists of two main blocks. The first is the output pre-producer, which is created using LFSR (Linear Feedback Shift Register) and NFSR (Non-linear Feedback Shift Register) structures. The second is the verifier producer, which uses a shift register and accumulator.

The output pre-producer provides a pseudo-random bit stream for use in encryption and authentication. The LFSR and NFSR blocks create a total of 256-bit pseudo-random state, with both LFSR and NFSR being 128 bits. The specific bits designated to fill the LFSR and NFSR blocks are explained in the relevant code as commented lines. These bits are placed again using decomposition.

The verifier block holds 128 bits of data, with a 64-bit shift register and accumulator. The shift register is composed of the last 64 bits of the output pre-producer.

The Grain-128AEAD NFSR block is implemented to be more effective against cryptographic attacks and to further complicate the relationship between the key, key stream, and state.

While the LFSR and NFSR are 128 bits, the hardware footprint is larger than ISAP. However, this choice is made to provide a more complex relationship, as mentioned in the previous paragraph.

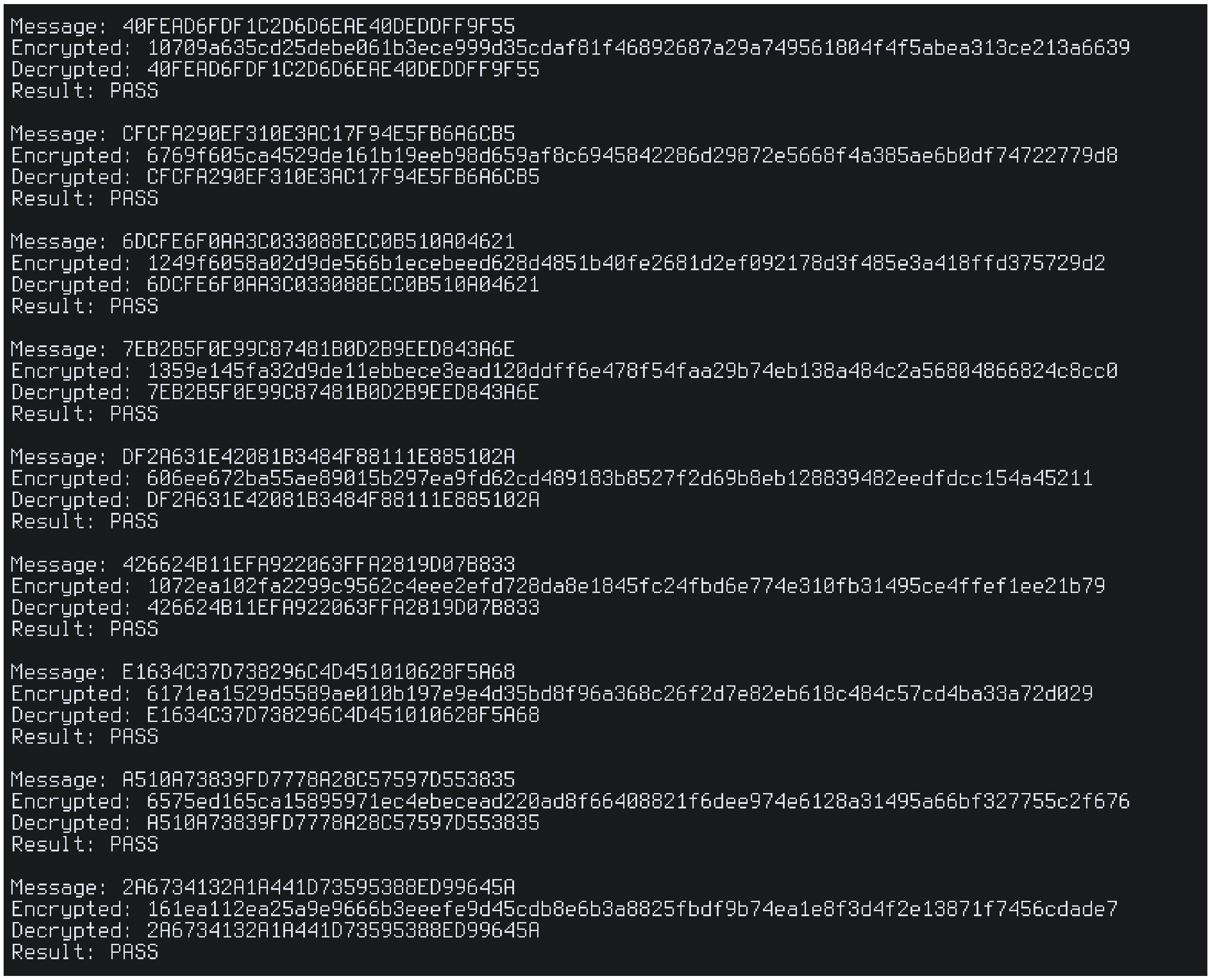
While ISAP uses a 16-bit MAC tag, Grain uses a 64-bit MAC tag.

Unlike ISAP, which can only be used with the encryption function, Grain requires the verification part.

Both ISAP and Grain use a 128-bit key.

Unlike ISAP, Grain does not keep the key in a fixed area, making it easier to update the key within the device. This expands its range of use.

Grain Test Results:



```
Message: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Encrypted: 10709a635cd25debe061b3ece999d35cdaf81f46892687a29a749561804f4f5abea313ce213a6639
Decrypted: 40FEAD6FDF1C2D6D6EAE40DEDDFF9F55
Result: PASS

Message: CFCFA290EF310E3AC17F94E5FB6A6CB5
Encrypted: 6769f605ca4529de161b19eeb98d659af8c6945842286d29872e5668f4a385ae6b0df74722779d8
Decrypted: CFCFA290EF310E3AC17F94E5FB6A6CB5
Result: PASS

Message: 6DCFE6F0AA3C033088ECC0B510A04621
Encrypted: 1249f6058a02d9de566b1ecebeed628d4851b40fe2681d2ef092178d3f485e3a418ffd375729d2
Decrypted: 6DCFE6F0AA3C033088ECC0B510A04621
Result: PASS

Message: 7EB2B5F0E99C87481B0D2B9EED843A6E
Encrypted: 1359e145fa32d9de11ebbece3ead120ddff6e478f54faa29b74eb138a484c2a56804866824c8cc0
Decrypted: 7EB2B5F0E99C87481B0D2B9EED843A6E
Result: PASS

Message: DF2A631E42081B3484F88111E885102A
Encrypted: 606ee672ba55ae89015b297ea9fd62cd489183b8527f2d69b8eb128839482eedfdcc154a45211
Decrypted: DF2A631E42081B3484F88111E885102A
Result: PASS

Message: 426624B11EFA922063FFA2819D07B833
Encrypted: 1072ea102fa2299c9562c4eee2efd728da8e1845fc24fbd6e774e310fb31495ce4ffef1ee21b79
Decrypted: 426624B11EFA922063FFA2819D07B833
Result: PASS

Message: E1634C37D738296C4D451010628F5A68
Encrypted: 6171ea1529d5589ae010b197e9e4d35bd8f96a368c26f2d7e82eb618c484c57cd4ba33a72d029
Decrypted: E1634C37D738296C4D451010628F5A68
Result: PASS

Message: A510A73839FD7778A28C57597D553835
Encrypted: 6575ed165ca15895971ec4ebecead220ad8f66408821f6dee974e6128a31495a66bf327755c2f676
Decrypted: A510A73839FD7778A28C57597D553835
Result: PASS

Message: 2A6734132A1A441D73595388ED99645A
Encrypted: 161ea112ea25a9e9666b3eeefe9d45cdb8e6b3a8825fbdf9b74ea1e8f3d4f2e13871f7456cdade7
Decrypted: 2A6734132A1A441D73595388ED99645A
Result: PASS
```

Sources

NISTIR 8114: Report on Lightweight Cryptography

Lightweight Block Ciphers

The KLEIN Block Cipher

The LED Block Cipher

Noekeon Slides

RECTANGLE: A Bit-slice Lightweight Block Cipher Suitable for Multiple Platforms