Digital System Design

Cipher Specification for Assignment 1

March, 2025

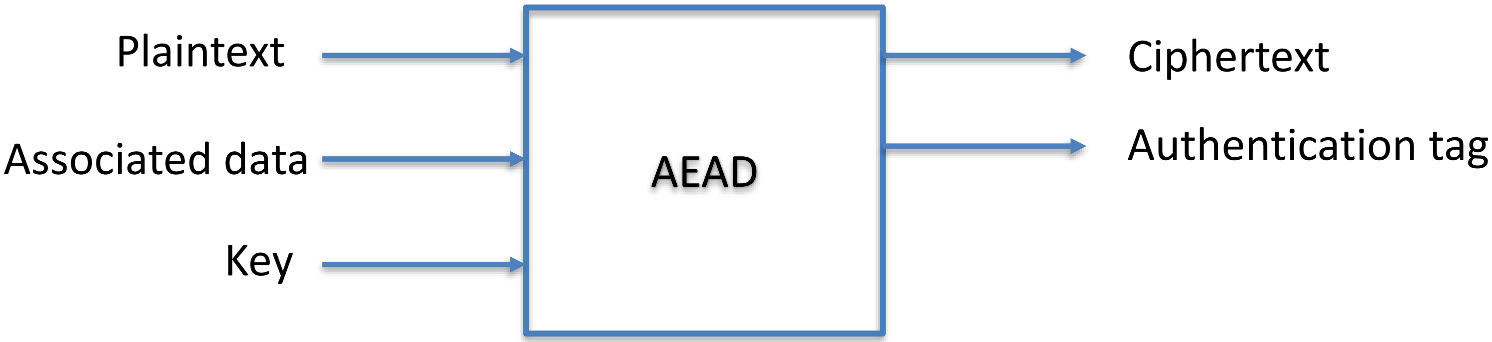
Sujoy Sinha Roy
sujoy.sinharoy@iaik.tugraz.at
Graz University of Technology

SCIENCE
PASSION
TECHNOLOGY

Authenticated Encryption with Associated Data (AEAD)

AEAD is a category of operating modes of block ciphers that ensure

1. authenticity
2. integrity
3. and confidentiality .



Two AEAD schemes for Assignment 1

Depending on your group, you will implement encryption of any one scheme

1. "Elephant" NIST Lightweight Cryptography Standardization.
   Full specification: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf

1. "PHOTON-Beetle" NIST Lightweight Cryptography Standardization.
   Full specification: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/photon-beetle-spec-round2.pdf

More information and source code: https://csrc.nist.gov/projects/lightweight-cryptography/round-2-candidates

Commonly used symbols

Symbols	Use
$a \oplus b$	Bitwise XOR between binary strings a and b
$a \parallel b$	Concatenation of binary strings a and b.
$a_1, a_2, \ldots \xleftarrow{r} a$	Splits a into r-bit sub strings a_1, a_2, etc.
a<<i	Left shift a by i positions with 0 filling in the right
a>>i	Right shift a by i positions with 0 filling in the left
a<<<i	Left circular shift of a by i positions
a>>>i	Right circular shift of a by i positions

Splitting of message into blocks

1. Message M is a binary string of any length.
2. It will be split into *n*-bit blocks.
3. If length(M) is not a multiple of *n*, then pad 0s at the end.

Footnote *: Depending on scheme, it 0s are added either to left or right. For a given scheme, you should check the specification and reference implementation.

Splitting of message into blocks

1. Message M is a binary string of any length.
2. It will be split into *n*-bit blocks.
3. If length(M) is not a multiple of *n*, then pad 0s at the end*.

Toy example: Let M = 10110100100011110101011
and n=4 bit.

Footnote *: Depending on scheme, it 0s are added either to left or right. For a given scheme, you should check the specification and reference implementation.

Splitting of message into blocks

1. Message M is a binary string of any length.
2. It will be split into *n*-bit blocks.
3. If length(M) is not a multiple of *n*, then pad 0s at the end.

Toy example: Let M = 1011010010001111010101011
and n=4 bit.

Length of M is 25. Hence pad three 0s to make the length 28.

M_0 M_1 M_2 M_3 M_4 M_5 M_6

M after **right** padding = 1011-0100-1000-1111-0101-0101-1000

Number of blocks = 28/4 = 7.

Footnote *: Depending on scheme, it 0s are added either to left or right. For a given scheme, you should check the specification and reference implementation.

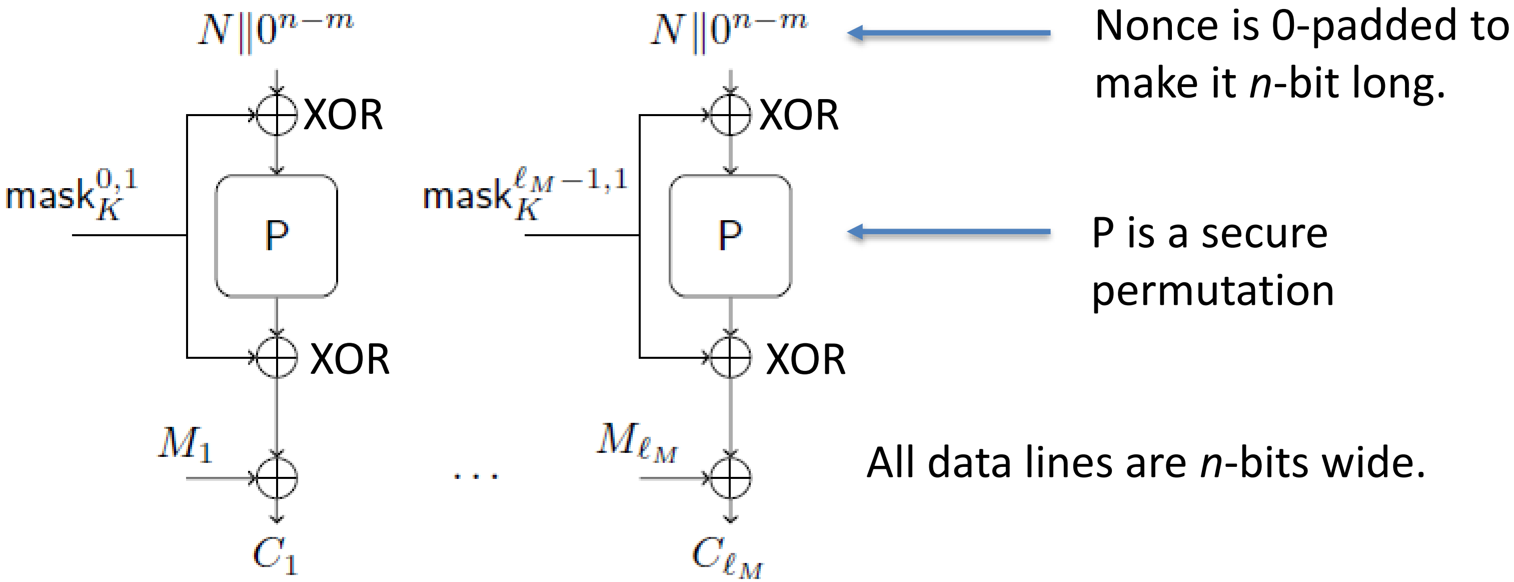
Encryption of Elephant
(Dumbo variant will be implemented in Assignment 1)

Notice

I will present the concept of the cipher. For exact parameters and orientation of bits, please follow the specification and reference implementation.

Ciphertext generation in Elephant

1. Encryption uses a random *m*-bit nonce *N* where $m \leq n$, where *n* is block length.
2. From the encryption key K, masks are generated using $\mathsf{mask}_K^{a,b} = \mathsf{mask}(K, a, b)$
3. Message blocks are encrypted one-by-one as shown below.



This example encrypts l_M blocks M_i and outputs l_M ciphertext blocks C_i

Permutation *P* in Elephant

- Elephant has three security levels.
- We will use the 160-bit permutation in Assignment 1.

instance	Key size k	m	n	t	P	φ_1	expected security strength	limit on online complexity
Dumbo	128	96	160	64	Spongent-π[160]	(3)	2^{112}	$2^{50}/(n/8)$
Jumbo	128	96	176	64	Spongent-π[176]	(4)	2^{127}	$2^{50}/(n/8)$
Delirium	128	96	200	128	Keccak-f[200]	(5)	2^{127}	$2^{74}/(n/8)$

$$\text{Spongent-}\pi[160]\colon \{0,1\}^{160} \rightarrow \{0,1\}^{160}$$

It maps 160-bit input into 160-bit output.

Permutation Spongent-π[160]

- This permutation is applied on the 160-bit state X.
- The state X is a byte-array of 20 words.

  BYTE state[20]

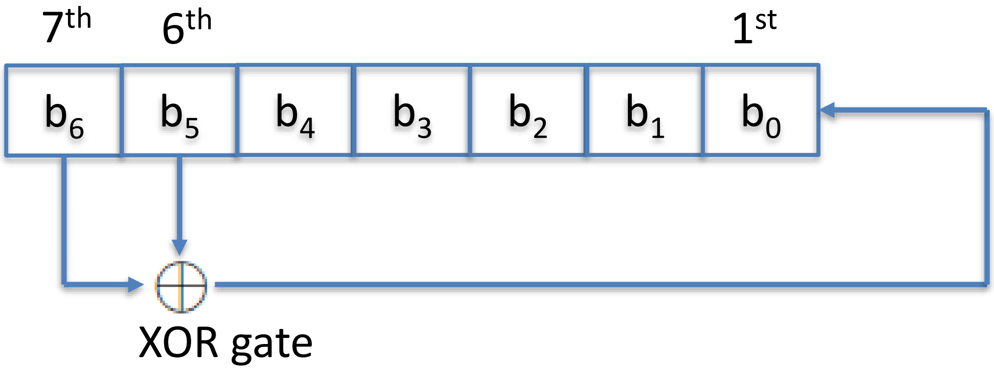
- The permutation performs three operations in a loop on state bytes of X.

P( ): Input X
for $i = 1, \ldots, 80$ **do**
  $X \leftarrow$ XOR most and least significant bytes of X with $\mathrm{ICounter}_{160}(i)$
  $X \leftarrow \mathrm{sBoxLayer}_{160}(X)$
  $X \leftarrow \mathrm{pLayer}_{160}(X)$
return X

ICounter$_{160}$(i)

- This function is a 7-bit Linear Feedback Shift Register (LFSR) initialized with "1110101" (Check spec/ref for ordering of bits)

- When the input is 'i', there are i number of shifts



- After one left shift, new bits of the LFSR is

$$\{b_6, b_5, \ldots, b_1, b_0\} \leftarrow \{b_5, b_4, \ldots, b_1, b_6 \wedge b_5\}$$

Footnote: Slide shows idea only. Check reference implementation for exact information.

Permutation Spongent-π[160]: Operation with ICounter

for i = 1, ..., 80 **do**

   $X \leftarrow \boxed{\text{XOR most and least significant bytes of } X \text{ with ICounter}_{160}(i)}$

   $X \leftarrow \text{sBoxLayer}_{160}(X)$

   $X \leftarrow \text{pLayer}_{160}(X)$

Step1:

$\text{ICounter}_{160}(i)$

↓

Byte IV = $\{0, b_6, b_5, \ldots, b_1, b_0\}$

↓

rev(IV)

↓

Byte INV_IV = $\{b_0, b_1, b_2, \ldots, b_6, 0\}$

Check ref for bit ordering

Permutation Spongent-π[160]: Operation with lCounter

for $i = 1, \ldots, 80$ **do**

$X \leftarrow$ $\boxed{\text{XOR most and least significant bytes of } X \text{ with lCounter}_{160}(i)}$

$X \leftarrow \text{sBoxLayer}_{160}(X)$

$X \leftarrow \text{pLayer}_{160}(X)$

Step2:

Update the least and most significant state bytes of X as:

state[0] = state[0] ^ IV;

State[19] = state[19] ^ INV_IV;

Permutation Spongent-π[160]: Operation with sBoxLayer

for i = 1, …, 80 **do**
    $X \leftarrow$ XOR most and least significant bytes of X with $\text{lCounter}_{160}(i)$
    $X \leftarrow \boxed{\text{sBoxLayer}_{160}(X)}$
    $X \leftarrow \text{pLayer}_{160}(X)$

1. The 160-bit state X is segmented into 4 bit chunks. There are 40 chunks.

2. Each 4-bit chunk is replaced by the mapping sBox()

chunk	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
sBox(chunk)	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

Permutation Spongent-π[160]: Operation with pLayer

for $i = 1, \ldots, 80$ **do**

$X \leftarrow$ XOR most and least significant bytes of X with $\mathrm{ICounter}_{160}(i)$

$X \leftarrow \mathrm{sBoxLayer}_{160}(X)$

$X \leftarrow \boxed{\mathrm{pLayer}_{160}(X)}$ This permutes the bits of X

pLayer_{160}: this function moves the j-th bit of its input to bit position $P_{160}(j)$, where

$$P_{160}(j) = \begin{cases} 40 \cdot j \bmod 159\,, & \text{if } j \in \{0, \ldots, 158\}\,, \\ 159\,, & \text{if } j = 159\,. \end{cases}$$

Permutation Spongent-π[160]: Operation with pLayer

for i = 1, ..., 80 **do**

  $X \leftarrow$ XOR most and least significant bytes of X with $\text{lCounter}_{160}(i)$

  $X \leftarrow \text{sBoxLayer}_{160}(X)$

  $X \leftarrow \boxed{\text{pLayer}_{160}(X)}$ This permutes the bits of X

pLayer_{160}: this function moves the j-th bit of its input to bit position $P_{160}(j)$, where

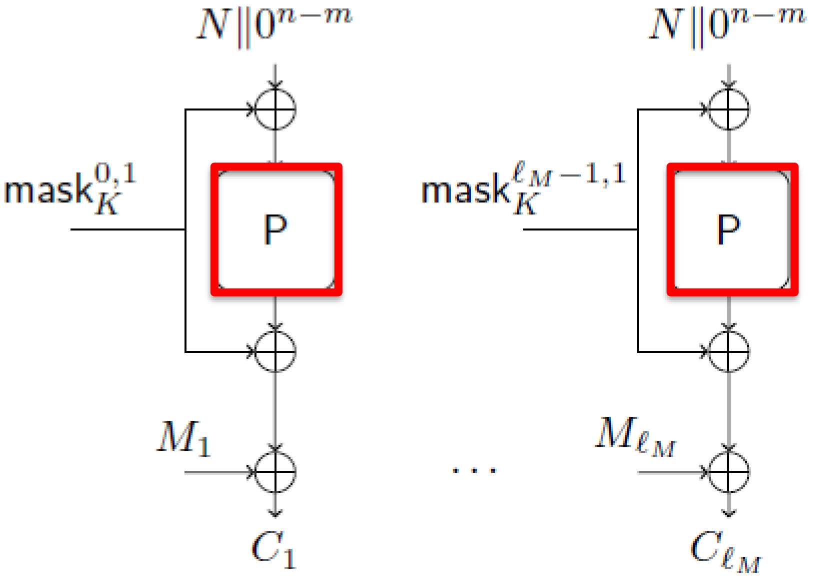
$$P_{160}(j) = \begin{cases} 40 \cdot j \bmod 159\,, & \text{if } j \in \{0, \ldots, 158\}\,, \\ 159\,, & \text{if } j = 159\,. \end{cases}$$

Example: Bit X_0 moves to position 0.

Bit X_1 moves to position 40.

Bit X_5 moves to position 200 mod 159 = 41.

Ciphertext generation in Elephant



We have seen how
P = Spongent-π[160] works

Next: We will see how $\mathsf{mask}_K^{a,b} = \mathsf{mask}(K, a, b)$ works.

Mask generation in Elephant

1. Takes an input *k*-bit key *K* and pads *n*-*k* number of 0s.
2. Then applies the P permutation on the state.
3. Applies the φ_1 LFSR a times.
4. $\varphi_2 = \varphi_1 \bigoplus$ ID where ID is the identity function.

$$\mathsf{mask}_K^{a,b} = \mathsf{mask}(K, a, b) = \varphi_2^b \circ \varphi_1^a \circ \mathrm{P}(K \| 0^{n-k})$$

There are only three values for b: {0, 1, 2}

LFSR φ_1

160-bit input X $\longrightarrow$ $\boxed{\varphi_1}$ $\longrightarrow$ 160-bit output X'

Input bytes of X: $\mathrm{state}[0], \mathrm{state}[1], \ldots., \mathrm{state}[19]$

Output bytes of X': $\mathrm{state}[1], \ldots., \mathrm{state}[19], z$

where $z = (\mathrm{state}[0] \lll 3) \oplus (\mathrm{state}[3] \ll 7) \oplus (\mathrm{state}[13] \gg 7)$

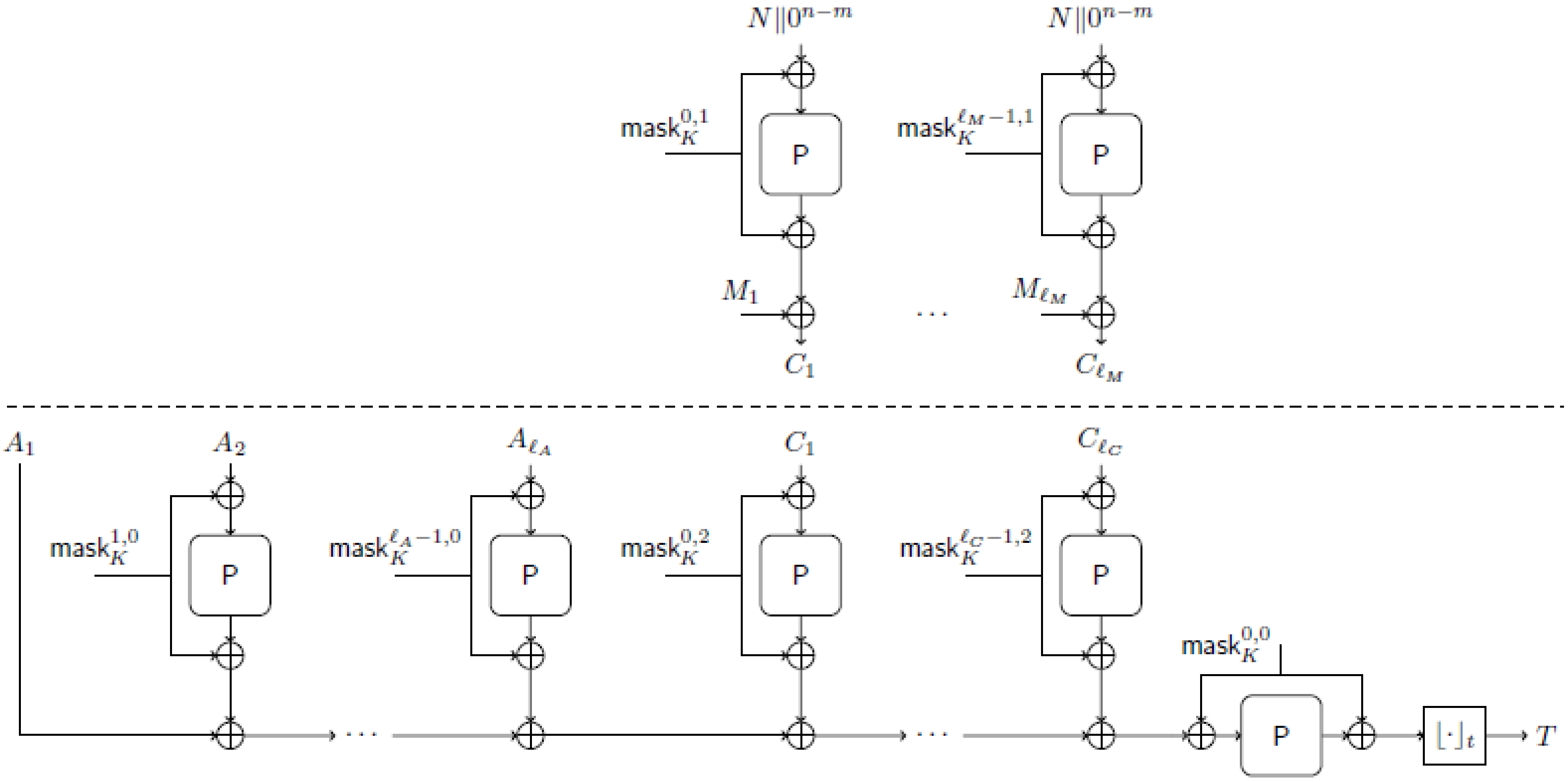
<<< is left-cyclic rotation

<< is left shift

>> is right shift

Footnote: Slide shows idea only.
Check ref. imp. for bit ordering.

Ciphertext and Tag Generation in Elephant



Main building blocks in Elephant

1. Permutation P
    - ICounter
    - S-box
    - Bit permutation
2. LFSR φ_1
3. State-machine for managing the operations

Assignment 1 on Elephant’s Encryption

What I presented is a *simplification* of the original Elephant.

Your implementation must meet the original specification

- You will implement the “Dumbo” version of Elephant.

  It uses 160-bit permutation.

- Read Section 2 of Elephant’s specification.
- **See the source reference C code of Elephant**.

https://csrc.nist.gov/projects/lightweight-cryptography/round-2-candidates

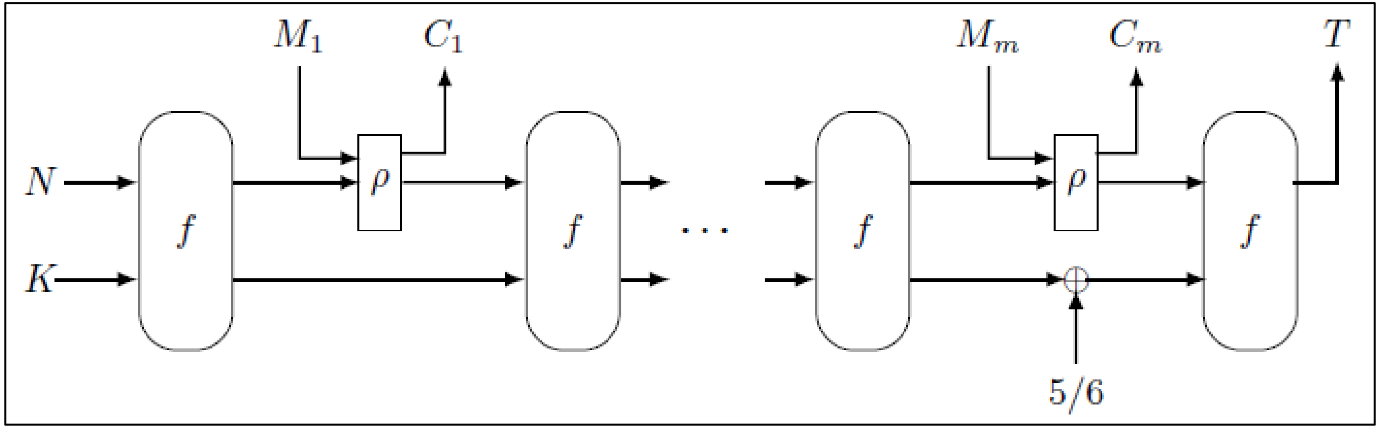
Encryption of PHOTON-Beetle
(AEAD[128] will be implemented in Assignment 1)

Notice

I will present the concept of the cipher. For exact parameters and orientation of bits, please follow the specification and reference implementation.

Ciphertext and Tag generation

Message M has m blocks M_i.
C_i is encryption of M_i.
Message block M_i and ciphertext block C_i are 128 bits.
Nonce N and key K are 128 bits.



$f(\,)$ is the PHOTON$_{256}$ permutation function.
ρ is a linear function.

State representation in PHOTON$_{256}$(X) permutation

It works on the 256-bit state X.

X is represented as a 2D matrix of 4-bit elements.

$$\begin{pmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,7} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,7} \\ & & \ddots & \\ x_{7,0} & x_{7,1} & \cdots & x_{7,7} \end{pmatrix}_{8 \times 8}$$

$x_{i,j}$ are 4-bit state elements.

PHOTON$_{256}$(X) permutation

- The permutation has 12 rounds.
- Each round has four layers.

$\texttt{PHOTON}_{256}(X)$

$\texttt{for}\ i = 0\ \texttt{to}\ 11:$

$\quad X \leftarrow \mathsf{AddConstant}(X, i);$

$\quad X \leftarrow \mathsf{SubCells}(X);$

$\quad X \leftarrow \mathsf{ShiftRows}(X);$

$\quad X \leftarrow \mathsf{MixColumnSerial}(X);$

$\textbf{return}\ X;$

PHOTON$_{256}$(*X*) permutation: AddConstant(*X*, *k*)

AddConstant(X, k)

$RC[12] \leftarrow \{1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10\};$

$IC[8] \leftarrow \{0, 1, 3, 7, 15, 14, 12, 8\};$

$\texttt{for } i = 0 \texttt{ to } 7:$

$\quad X[i, 0] \leftarrow X[i, 0] \oplus RC[k] \oplus IC[i];$

return $X;$

Adds constants to the first column of state matrix X.

Round constant *RC*[*k*] depends on the iteration counter within PHOTON$_{256}$.

PHOTON$_{256}$(X) permutation: SubCells(X)

SubCells(X)

```
1:   for i = 0 to 7, j = 0 to 7 :
2:       X[i, j] ← S-Box(X[i, j]);
return X;
```

This substitutes each 4-bit state element according to the table:

Table 2.1: The PHOTON S-box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S-box	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Example: X[2,3] = 7 after substitution becomes X[2,3] = D.

PHOTON$_{256}$(X) permutation: ShiftRows(X)

ShiftRows(X)

$1:\quad \texttt{for } i = 0 \texttt{ to } 7, j = 0 \texttt{ to } 7:$

$2:\quad\quad X'[i,j] \leftarrow X[i,(j+i)\%8]);$

$\textbf{return } X';$

State element within a row are cyclically rotated.

Example: Let the 3rd row of X be X[2] = [5, D, A, 3, 4, F, 2, 7].

After ShiftRows() it becomes X'[2] = [A, 3, 4, F, 2, 7, 5, D]

Check exact left vs. right ordering from ref. imp.

PHOTON$_{256}$(X) permutation: MixColumnSerial(X)

MixColumnSerial(X)

---

1 : $M \leftarrow \mathsf{Serial}[2, 4, 2, 11, 2, 8, 5, 6];$

2 : $X \leftarrow M^8 \odot X;$

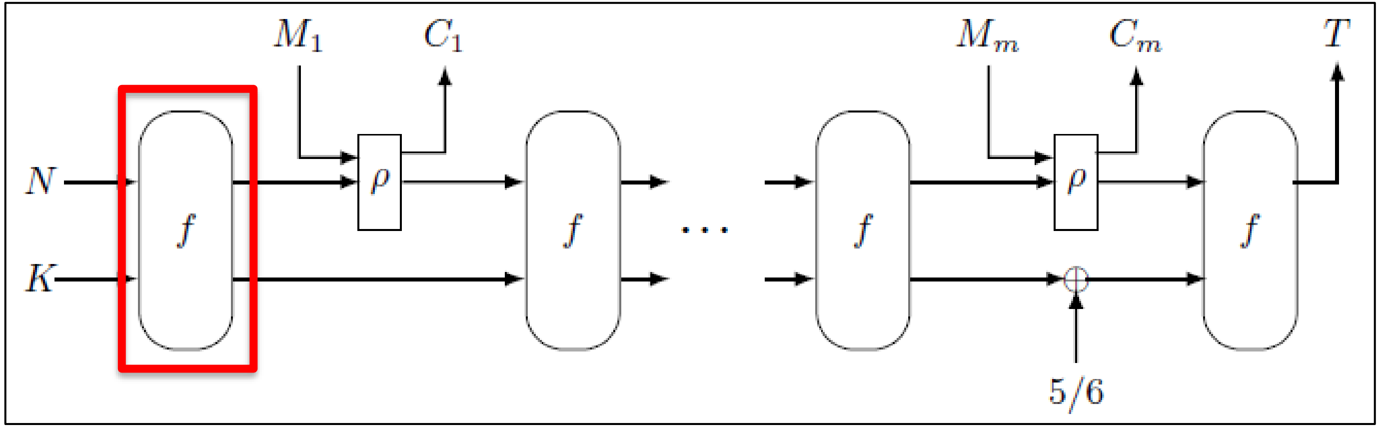
return $X;$

M^8 is a constant matrix.

M is the constant 'serial' matrix = $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 4 & 2 & 11 & 2 & 8 & 5 & 6 \end{pmatrix}$

Obtain constants from ref. imp.

PHOTON-Beetle: Overall block diagram



We have studied *f*( )= PHOTON$_{256}$ permutation function

Next: Structure of *ρ* is a linear function.

PHOTON-Beetle: *ρ* linear function.

- Two inputs: $S \in \{0, 1\}^r$ and $U \in \{0, 1\}^{\leq r}$.
- Two outputs: $S \in \{0, 1\}^r$ and $V \in \{0, 1\}^{|U|}$.
- where r is 128.

$\rho(S, U)$

1 : $V \leftarrow \mathsf{Trunc}(\mathsf{Shuffle}(S), |U|) \oplus U$;
2 : $S \leftarrow S \oplus \mathsf{Ozs}_r(U)$;
return (S, V);

$\mathsf{Shuffle}(S)$

1 : $S_1 \| S_2 \xleftarrow{r/2} S$;
return $S_2 \| (S_1 \ggg 1)$;

- Trunc(X, i) is a truncation function. It returns most significant i bits of X.
- $\mathrm{Ozs}_r(U)$ appends 10* to U and outputs $U \,\|\, 1 \,\|\, 0^{r-|U|-1}$
- In general, S, U and V are all r-bits in PHOTON-Beetle-AEAD.

PHOTON-Beetle: *ρ* linear function.

- Two inputs: $S \in \{0, 1\}^{r}$ and $U \in \{0, 1\}^{\leq r}$.
- Two outputs: $S \in \{0, 1\}^{r}$ and $V \in \{0, 1\}^{|U|}$.
-

Becomes simpler for the following specific case

- Trunc(X, i) is a truncation function. It returns most significant i bits of X.
- $\mathrm{Ozs}_r(U)$ appends 10* to U and outputs $U \,\|\, 1 \,\|\, 0^{r-|U|-1}$
- In general, S, U and V are all r-bits in PHOTON-Beetle-AEAD.

Simplified ρ when |S|, |U|, and |V| are of length 128

$\rho(S, U)$

$S_1 \parallel S_2 \leftarrow S$ /* S1 and S2 are 64-bit words */

$temp \leftarrow S_2 \parallel (S_1 >>> 1)$ /* Rotate S1 right-cyclic and rearrange S1, S2 */

$S \leftarrow S \oplus U$ /* Output state S is computed from S and data U */

$V \leftarrow temp \oplus U$ /* Output data V is computed shuffled state and U */

return (S, V);

Footnote: Check reference implementation for exact information.

Main building blocks in PHOTON-Beetle

1. Permutation PHOTON_{256}
    - Constant addition (XOR)
    - S-box (Table access)
    - Shift rows
    - Mix Columns (matrix multiplication $M^8 \odot X$)
        - Field multiplication and XOR
2. Simplified linear function ρ
3. State-machine for managing the operations

Next: matrix multiplication $M^8 \odot X$

Elements are multiplied in a binary field

Field multiplication

4-bit values are multiplied with reduction polynomial $z^4 + z + 1$.

$$z^4 = z + 1 \bmod GF(2^4)$$

Let two 4-bit values be $a=\{a_3, a_2, a_1, a_0\}$ and $b=\{b_3, b_2, b_1, b_0\}$.

We can write them as polynomial

$a(z) = a_3z^3 + a_2z^2 + a_1z + a_0$

$b(z) = b_3z^3 + b_2z^2 + b_1z + b_0$

Field multiplication (2)

$a(z) = a_3z^3 + a_2z^2 + a_1z + a_0$
$b(z) = b_3z^3 + b_2z^2 + b_1z + b_0$

$a(z)*b(z)$ gives $c(z) = c_6z^6 + c_5z^5 + \dots c_3z^3 + \dots + c_0$

$$\left.\begin{aligned} \text{where } c_0 &= a_0\&b_0 \\ c_1 &= (a_0\&b_1) \wedge (a_1\&b_0) \\ c_2 &= (a_0\&b_2) \wedge (a_1\&b_1) \wedge (a_2\&b_0) \\ &\dots \\ c_6 &= (a_3\&b_b) \end{aligned}\right\} \text{These } c_i \text{ are bits}$$

& is AND gate
^ is XOR gate

Multiplication result has 7 bits c_0 to c_6

$$c(z) = c_6z^6 + c_5z^5 + \dots\ c_3z^3 + \dots + c_0$$

Result is reduced to 4 bits using $z^4 = z + 1$

Field multiplication (3)

Next, reduce $c_6z^6 + c_5z^5 + c_4z^4$ using $z^4 = z + 1$, $z^5 = z^2 + z$, $z^6 = z^3 + z^2$.

That gives:

$c_4z^4 = c_4z + c_4$

$c_5z^5 = c_5z^2 + c_5z$

$c_6z^6 = c_6z^3 + c_6z^2$

Field multiplication (3)

Next, reduce $c_6z^6 + c_5z^5 + c_4z^4$ using $z^4 = z + 1$, $z^5 = z^2 + z$, $z^6 = z^3 + z^2$.

That gives:

$c_4z^4 = c_4z + c_4$

$c_5z^5 = c_5z^2 + c_5z$

$c_6z^6 = c_6z^3 + c_6z^2$

$$c_6z^6 + c_5z^5 + \ldots c_3z^3 + \ldots + c_0 \rightarrow (c_6z^3 + c_6z^2) + (c_5z^2 + c_5z) + (c_4z + c_4) + c_3z^3 + \ldots + c_0$$

Field multiplication (4)

Next, reduce $c_6z^6 + c_5z^5 + c_4z^4$ using $z^4 = z + 1$, $z^5 = z^2 + z$, $z^6 = z^3 + z^2$.

That gives:

$c_4z^4 = c_4z + c_4$

$c_5z^5 = c_5z^2 + c_5z$

$c_6z^6 = c_6z^3 + c_6z^2$

$$c_6z^6 + c_5z^5 + \ldots c_3z^3 + \ldots + c_0 \rightarrow (c_6z^3 + c_6z^2) + (c_5z^2 + c_5z) + (c_4z + c_4) + c_3z^3 + \ldots + c_0$$

Final result is 4 bits: $d(z) = (c_6 + c_3)z^3 + \ldots + (c_5+c_4+c_1)z + (c_4 + c_0)$

where bit addition of two values is XOR operation.

Assignment 1 on PHOTON-Beetle Encryption

What I presented is a *simplification* of the original PHOTON-Beetle

Your implementation must meet the original specification

- You will implement PHOTON-Beetle-AEAD[128].
- Read Chapter 3 on the specification.
- See the source reference C code of PHOTON-Beetle-AEAD[128].

https://csrc.nist.gov/projects/lightweight-cryptography/round-2-candidates