π–*Cipher v1*[1]

Designers: Danilo Gligoroski[2] and Hristina Mihajloska[3] and Simona Samardjiska[23] and Håkon Jacobsen[2] and Mohamed El-Hadedy[2] and Rune Erlend Jensen[4]

Submitter: Danilo Gligoroski
`danilog@item.ntnu.no`

14.03.2014

[1]Since, the name of the cipher contains the Greek letter π, in the software implementations we will use the name `PiCipher`. More precisely in this document we propose the following six variants of the cipher: `Pi16Cipher096v1`, `Pi16Cipher128v1`, `Pi32Cipher128v1`, `Pi32Cipher256v1`, `Pi64Cipher128v1`, `Pi64Cipher256v1`

[2]ITEM, Norwegian University of Science and Technology, Trondheim, Norway

[3]FCSE, "Ss Cyril and Methodius" University, Skopje, Republic of Macedonia

[4]IDI, Norwegian University of Science and Technology, Trondheim, Norway

Chapter 1

Specification

1.1 Parameters, variables and constants

The following parameters and variables are used in the specification of π-Cipher:

$\pi\omega$-Cipher$n$	AEAD cipher defined with ω-bit words and n-bit security.
$\omega = 16, 32, 64$	Size of binary words in bits that are used in π-Cipher.
π function	Main permutation function of the cipher.
M	Message or plaintext.
$mlen$	Length of a message less than $2^{64} - 1$ bytes.
m	Number of message blocks.
M_m	Padding rule for the last message block $M_m \leftarrow \begin{cases} M_m & \text{if } \lvert M_m\rvert = bitrate, \\ M_m \vert\vert 10^* & \text{if } \lvert M_m\rvert < bitrate \end{cases}$ where 1 represents the byte `0x01`, and 0 represents the byte `0x00`.
K	Secret key.
$klen$	Length of a key K in bytes. It can be 12 bytes (96 bits), 16 bytes (128 bits) or 32 bytes (256 bits).

AD	Associated data. This data can not be encrypted or decrypted.
$adlen$	Length of an associated data less than $2^{64} - 1$ bytes.
a	Number of associated data blocks. Thus, $AD = AD_1 \| \| AD_2 \| \| \ldots \| \| AD_a$.
AD_a	Padding rule for the last AD block $AD_a \leftarrow \begin{cases} AD_a & \text{if } \|AD_a\| = bitrate, \\ AD_a \| \| 10^* & \text{if } \|AD_a\| < bitrate \end{cases}$ where 1 represents the byte `0x01`, and 0 represents the byte `0x00`.
IS	Internal state, bijectively transformed by the π function. Throughout this document when IS is used as a common internal state for many parallel computations, we will use the abbreviation CIS (*Common Internal State*).
I_i	Chunk from the internal state IS. It is a 4-tuple of ω–bit words. $I_i = (IS_{i1}, IS_{i2}, IS_{i3}, IS_{i4})$.
N	The internal state IS is divided into N chunks of length $4 \times \omega$. N is even number and $N \geqslant 4$. Thus, $IS = (I_1, I_2, \ldots, I_N)$.
b	Size of IS in bits. It is constrained by the following relation: $b = N \times 4 \times \omega$.
$bitrate$	Rate of the IS from the sponge construction paradigm point of view. $IS_{bitrate} = I_1 \ \| \| \ I_3 \ \| \| \ \ldots \ \| \| \ I_{N-1}$.
$capacity$	Capacity of the IS from the sponge construction paradigm point of view. $IS_{capaity} = I_2 \ \| \| \ I_4 \ \| \| \ \ldots \ \| \| \ I_N$.
$\| \| \| \|$	This is an operator of interleaved concatenation in order to correctly denote a concatenation of $IS_{bitrate}$ and $IS_{capaity}$ that restores IS i.e., $IS = IS_{bitrate} \ \| \| \| \| \ IS_{capaity}$.
PMN	Public message number. The size $\|PMN\|$ in bits is constrained by the following relation: $8 \times klen + \|PMN\| + 8 \leqslant b$.

SMN	Secret message number. The size $\lvert SMN\rvert$ in bits is constrained by the following relations: $\lvert SMN\rvert = 0$ or $\lvert SMN\rvert = bitrate$.
$NONCE$	$NONCE = (PMN, SMN)$.
C	Ciphertext.
$clen$	Length of the ciphertext in bytes, where $clen = mlen + \lvert SMN\rvert + tlen$ (here $\lvert SMN\rvert$ is in bytes).
T	Authentication tag for the message, $NONCE$ and associated data.
$tlen$	Length of the authentication tag in bytes. It is constrained by the following relation: $tlen = \frac{bitrate}{8}$.
R	Tweakable parameter that represents the number of rounds in π function.
ctr	64-bit counter used in the cipher. It is initialized from the first 64 bits of the $IS_{capacity}$.
$\boxplus_d$	Operation of componentwise addition of two d-dimensional vectors of ω-bit words in $(\mathbb{Z}_{2^\omega})^d$.

π-Cipher is designed for different word sizes and different security levels. The recommended variants are presented in Table 1.1.

1.2 General design properties

π-Cipher is parallel, incremental, nonce based, tag second-preimage resistant, authenticated encryption cipher with associated data. It involves several solid cryptographic concepts such as:

1. The design belongs to the category of Encrypt-then-MAC authenticated ciphers.
2. Its parallel and incremental design is similar to the design of the counter based

Table 1.1: Basic characteristics of all variants of the π-Cipher

	Word size ω (in bits)	*klen* (in bits)	PMN (in bits)	SMN (in bits)	b (in bits)	N	*bitrate* (in bits)	Tag t (in bits)	R
π16-Cipher096	16	96	32	0 or 128	256	4	128	128	4
π16-Cipher128	16	128	32	0 or 128	256	4	128	128	4
π32-Cipher128	32	128	128	0 or 256	512	4	256	256	4
π32-Cipher256	32	256	128	0 or 256	512	4	256	256	4
π64-Cipher128	64	128	128	0 or 512	1024	4	512	512	4
π64-Cipher256	64	256	128	0 or 512	1024	4	512	512	4

XOR MAC scheme of [1], but in order to achieve the second-preimage resistance for the MAC tags, instead of XOR operations for the intermediate tag components we use componentwise additions in $(\mathbb{Z}_{2^\omega})^d$.

3. In [4, Sec. 3.3] the authors mention that it is possible to construct an authenticated encryption using two pass sponge construction [3] that is proven to be secure as long as the underlying sponge permutation has no structural distinguishers. In the same paper [4] the duplex sponge construction is introduced. Although these constructions have the property to be tag second-preimage resistant, neither of them is incremental. An incremental and parallel two pass scheme is proposed in [10]. However, the design goal for that scheme was not to be tag second-preimage resistant. Combining all these ideas, in our design we use a two pass counter based sponge component that we call *triplex component*. Our design is fully parallelizable, incremental and tag second-preimage resistant. The used π permutation is based on ARX (Addition, Rotation and XOR) operations.

1.2.1 Authenticated encryption

The encryption/authentication procedure of π-Cipher accepts key K with fixed-length of *klen* bytes, message M with *mlen* bytes and associated data AD with *adlen* bytes. The cipher uses a fixed-length public message number PMN and secret message number SMN. The output of the encryption/authentication procedure is a ciphertext C with *clen* bytes and a tag T with fixed-length of *tlen* bytes. The length *clen* of the ciphertext C is a sum of the byte length of the message, the authentication tag and the encrypted secret message number. The decryption/verification procedure accepts key K, associated

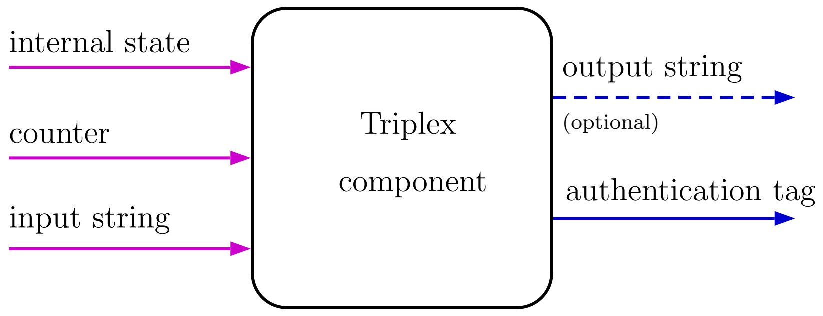

Figure 1.1: A general scheme of the triplex component

data AD, ciphertext C, public message number PMN, secret message number SMN and tag T, and returns the decrypted message M if the tag has been verified or $\perp$ otherwise.

The main building element in the operations of encryption/authentication and decryption/verification is our new construction related to the duplex sponge, called triplex component. It uses the permutation function π twice, it injects a counter into the internal state and digests an input string. The triplex component always outputs a tag. Optionally after the first call of the permutation function it can output a string (that can be a ciphertext block or a message block). The general scheme of the triplex component is presented in Figure 1.1.

Because of the differences in the encryption/authentication and decryption/verification procedures, there are two different variants of the triplex component. We call them *e-triplex* (for the phase of encryption) and *d-triplex* (for the phase of decryption). The only difference in these two components is how the input string is treated after the first call of the permutation function. In the first one, the input string (plaintext) is XORed with the current internal state and the result proceeds to the second invocation of the permutation function π. In the d-triplex component, the input string (ciphertext) is directly injected as a part of the internal state before the second invocation of the permutation function π. The graphical representation of the e-triplex and d-triplex components is given in Figure 1.2.

The encryption/authentication operation of π-Cipher can be described in four phases:

1. **Initialization**. In this phase we append a single "1" (i.e., `0x01` in hexadecimal byte representation) and the smallest number of 0's (i.e., `0x00` in hexadecimal byte representation) to the concatenated value of the key and the public message

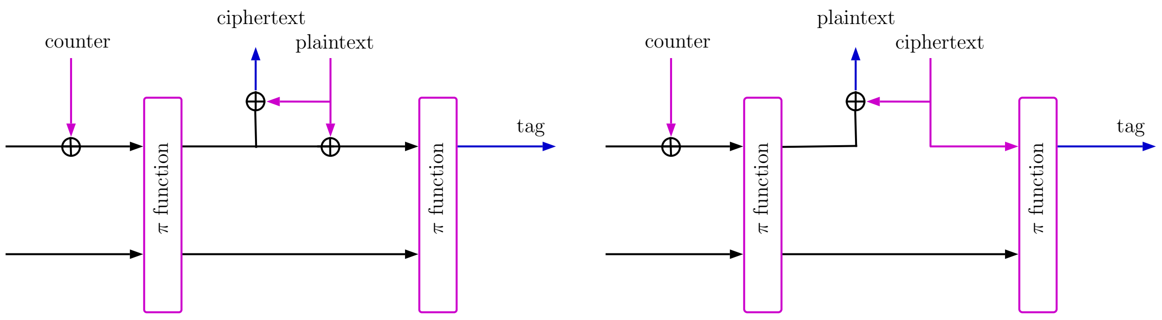

Figure 1.2: Triplex component

number. The length of the result should be less than or equal to the length of the internal state of the permutation function π. In other words the internal state is initialized with $K||PMN||10^*$. Then the internal state is updated by applying the permutation function π. Because, π-Cipher works in parallel mode, it has an initial value for the state for all of the parallel parts. We call it *Common Internal State (CIS)*. The CIS is initialized as:

$$CIS \leftarrow \pi(K||PMN||10^*).$$

The next part of this phase is initializing the counter ctr.

Since $CIS = CIS_{bitrate}||||CIS_{capacity}$ we initialize the ctr as the first 64 bits (little endian representation) of the $CIS_{capacity}$.

The graphical representation of the Initialization phase is given in Figure 1.3.

2. **Processing the associated data**.
The associated data $AD = AD_1|| \dots ||AD_i|| \dots ||AD_a$ is processed block by block in parallel using e–triplex components. The padding rule for the last block is the following:

$$AD_a \leftarrow \begin{cases} AD_a & \text{if } |AD_a| = bitrate, \\ AD_a||10^* & \text{if } |AD_a| < bitrate, \end{cases}$$

where 1 represents the byte `0x01`, and 0 represents the byte `0x00`.

To every block AD_i we associate a unique counter calculated as a sum of the initial

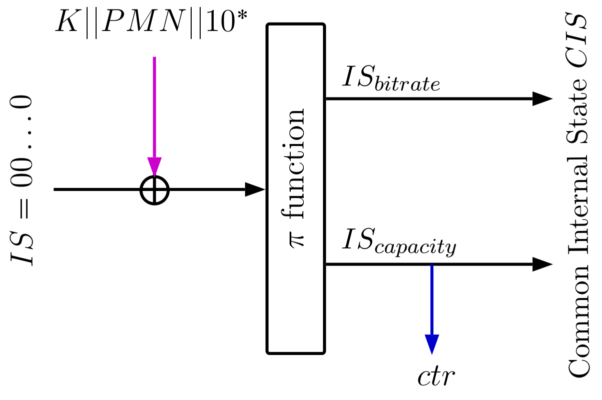

Figure 1.3: Initialization step

counter *ctr* and the ordinal number of the processed block i. The input to every e–triplex component is CIS, $ctr + i$ and AD_i, and the output is a block tag t'_i.

The tag T' for the associated data is computed as a component-wise addition $\boxplus_d$ of a, the d-dimensional vectors $t'_i \in (\mathbb{Z}_{2^\omega})^d$, where d is the number of ω-bit words in the *bitrate*. In other words,

$$T' = \boxplus_{d\,i=1}^{\,a} t'_i = t'_1 \boxplus_d t'_2 \boxplus_d \ldots \boxplus_d t'_a$$

where $t'_i = (t'_{i1}, t'_{i2}, \ldots, t'_{id})$ is a d-dimensional vector of ω-bit words.

The final part of this phase is to update the value of the *Common Internal State* CIS. Updating the CIS is done by xoring it with the tag T' and applying the π permutation function i.e. by the following expression:

$$CIS \leftarrow \pi(CIS_{bitrate} \bigoplus T' \;\; |||| \;\; CIS_{capacity})$$

This step is described graphically in Figure 1.4.

3. **Processing the secret message number**. This phase is omitted if the length of the secret message number SMN is 0 (it is the empty string). If SMN is not the empty string, then the first step in this phase is a call to the e-triplex component. The input is the following triplet: $(CIS, ctr + a + 1, SMN)$, and the output is the following pair: (C_0, t_0). The second step of this phase is updating the CIS (for

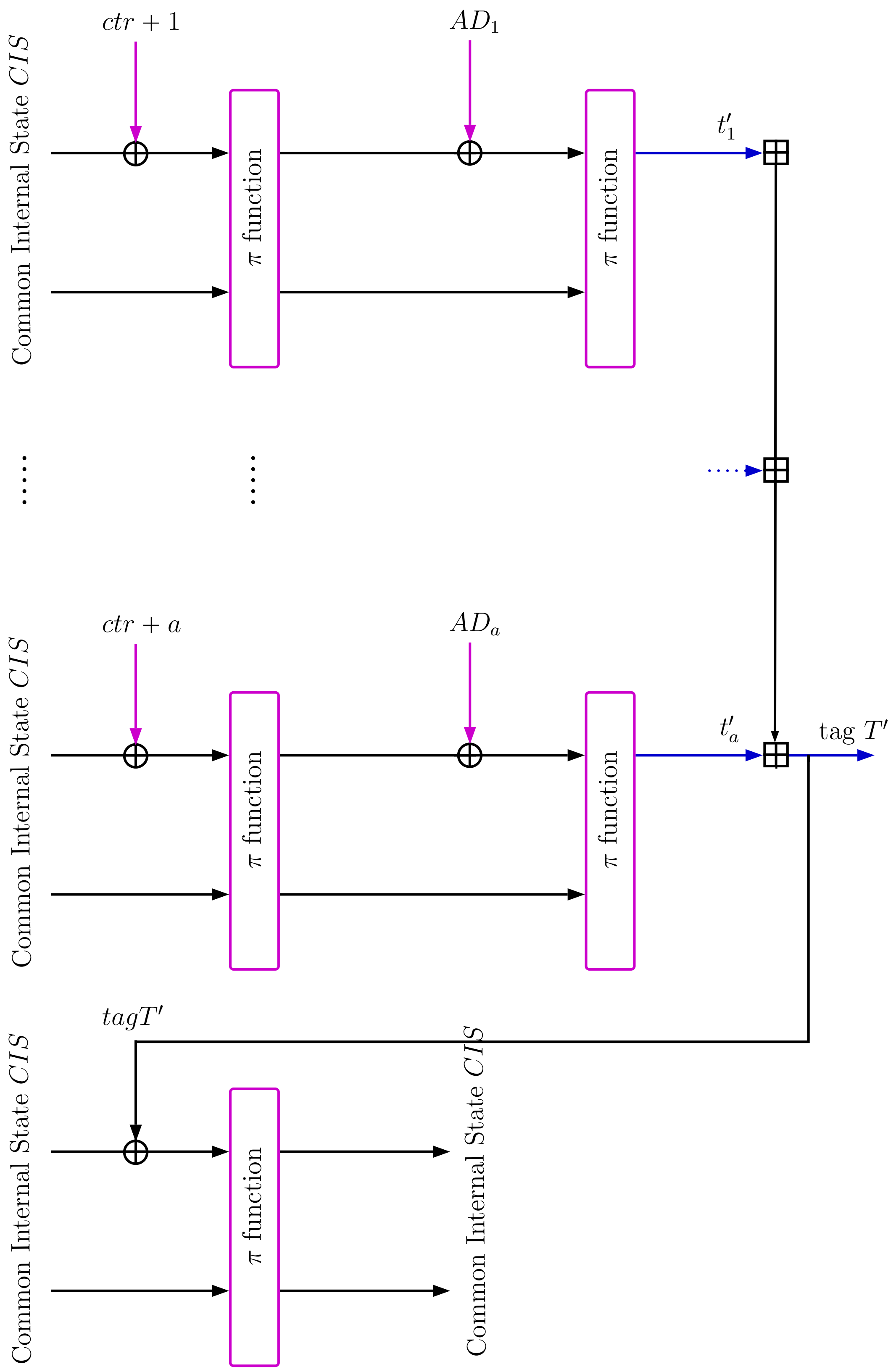
ctr + 1
AD_1
Common Internal State CIS
π function
π function
t'_1
ctr + a
AD_a
t'_a
tag T'
tagT'
Common Internal State CIS
Common Internal State CIS
Common Internal State CIS

Figure 1.4: Processing the associated data AD with a blocks in parallel

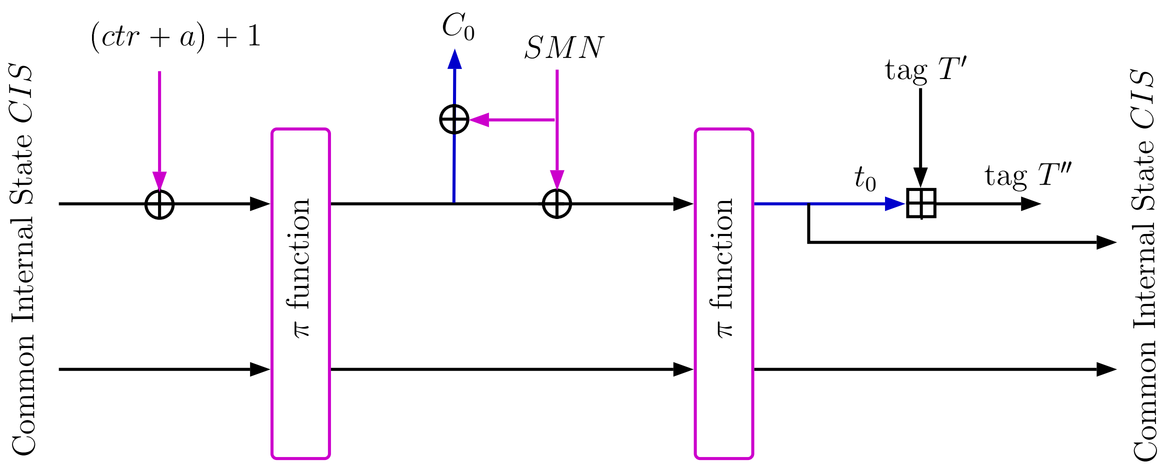

Figure 1.5: Processing the secret message number SMN

free) which becomes the value of the current internal state after the processing of SMN. Formally, the updating can be described by the following two expressions:

$$IS \leftarrow \pi(CIS_{bitrate} \bigoplus (ctr + a + 1) \;\; |||| \;\; CIS_{capacity}),$$
$$CIS \leftarrow \pi(IS_{bitrate} \bigoplus SMN \;\; |||| \;\; IS_{capacity})$$

The tag produced from this phase is

$$T'' = T' \boxplus_d t_0.$$

This phase is described graphically in Figure 1.5.

4. **Processing the message**. The message $M = M_1||\ldots||M_j||\ldots||M_m$ is processed block by block in parallel by e–triplex components. The padding rule for the last block is the following:

$$M_m \leftarrow \begin{cases} M_m & \text{if } |M_m| = bitrate, \\ M_m||10^* & \text{if } |M_m| < bitrate, \end{cases}$$

where 1 represents the byte `0x01`, and 0 represents the byte `0x00`.

To every block M_j we associate a unique block counter. It can be calculated as:

$$ctr \leftarrow \begin{cases} ctr + a + j & \text{if } |SMN| = 0, \\ ctr + a + 1 + j & \text{if } |SMN| = bitrate, \end{cases}$$

where j is the ordinal number of the processed block in the message, and $0 < j \leqslant m$. The input to every e–triplex component is the CIS, block ctr and M_j, and the output is a pair (C_j, t_j). By definition we put that the length of the final ciphertext block C_m is the same as the length of the un-padded last plaintext block M_m i.e., $|C_m| = |M_m|$.

The final tag T is obtained as a $\boxplus_d$ sum of all block tags t_j and the previously obtained tag T''.

$$T = T'' \boxplus_d t_1 \boxplus_d \ldots \boxplus_d t_j \boxplus_d \ldots \boxplus_d t_m.$$

where $t_j = (t_{j1}, t_{j2}, \ldots, t_{jd})$ is a d-dimensional vector of ω-bit words.

This phase is described graphically in Figure 1.6.

The output of the encryption/authentication procedure is the ciphertext

$$C = C_0||C_1||\ldots||C_m||T.$$

1.2.2 Decryption and verification

The decryption/verification procedure is defined correspondingly. There are four phases and the only difference is in the last two (so the Initialization phase and Processing the associated data phase are completely the same as in the encryption/authentication procedure).

The decryption of the SMN is performed in the phase of Processing the secret message number. Thus, instead of using an e-triplex component, we use a d-triplex component. The input parameters are: CIS, incremented counter $ctr + a + 1$ and the ciphertext block C_0. The output is a pair (SMN, t_0). The tag is processed in the same way as in the encryption/authentication procedure.

For the decryption of the rest of the ciphertext we continue to use a d-triplex component (instead of e-triplex). The output is now a decrypted message block and a tag value.

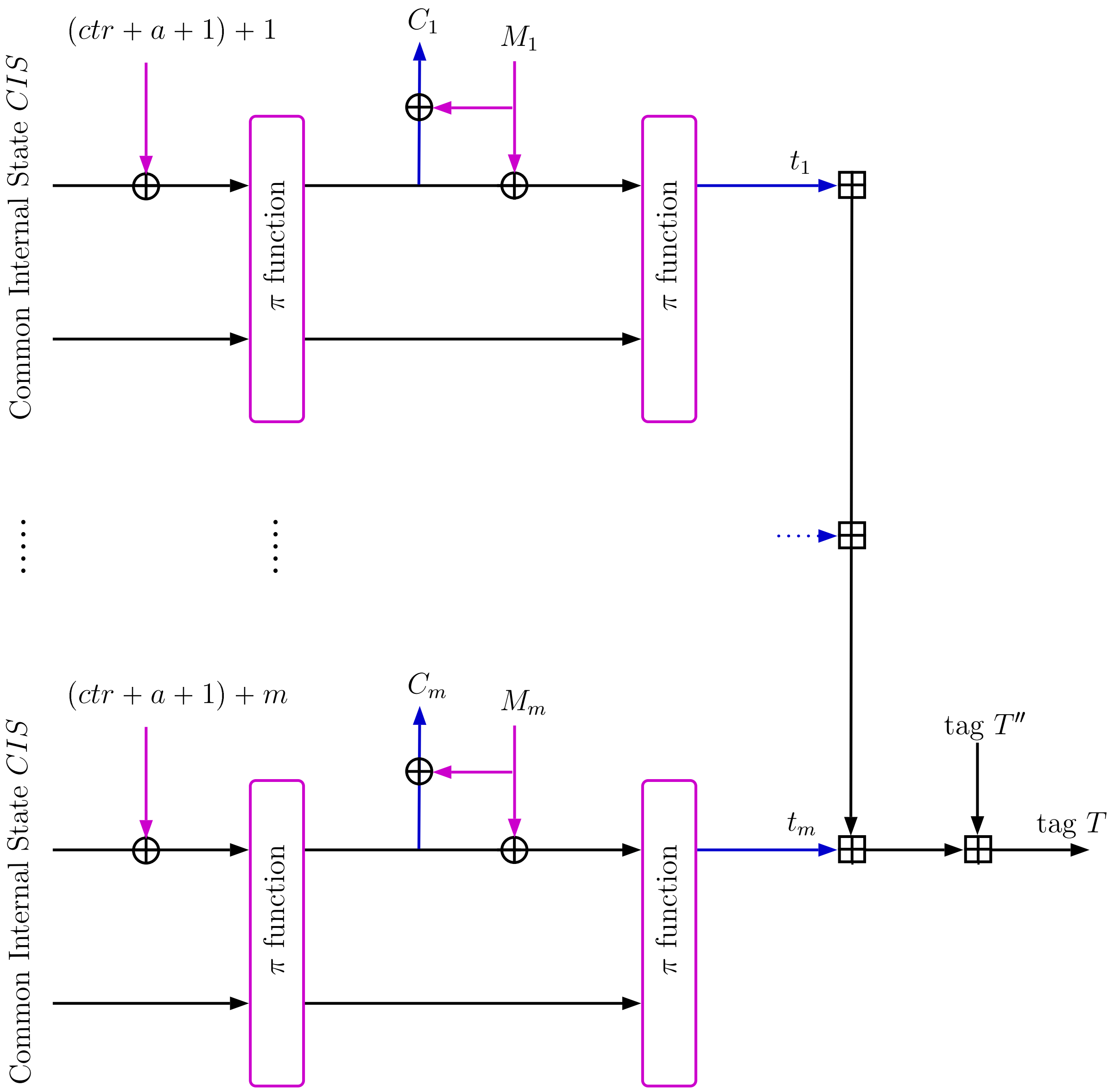

Figure 1.6: Processing the message M with m blocks in parallel

At the end, the supplied tag value T is compared to the one computed by the algorithm. Only if the tag is correct, the decrypted message is returned.

1.3 The π-function

The core part of every sponge construction is the permutation function, and the whole security of the primitive relies on it. The design goal for our sponge construction was to obtain a strong permutation, which for different values of the parameter ω (the bit size of the words) provides different features, i.e. to be very efficient when $\omega = 64$ and lightweight when $\omega = 16$.

π-Cipher has ARX based permutation function which we denote as π function. It uses similar operations as the operations used in the hash function Edon-R [8] but instead of using 8–tuples here we use 4–tuples. The permutation operates on a b bits state and updates the internal state through a sequence of R successive transformations - rounds. The state IS can be represented as a list of N 4-tuples, each of length ω-bits, where $b = N \times 4 \times \omega$, i.e.,

$$IS = (\underbrace{(IS_{11}, IS_{12}, IS_{13}, IS_{14})}_{I_1}, \underbrace{(IS_{21}, IS_{22}, IS_{23}, IS_{24})}_{I_2}, \ldots, \underbrace{(IS_{N1}, IS_{N2}, IS_{N3}, IS_{N4})}_{I_N}). \tag{1.1}$$

The general permutation function π consists of three main transformations $\mu, \nu, \sigma : \mathbb{Z}_{2^\omega}^4 \to \mathbb{Z}_{2^\omega}^4$, where $\mathbb{Z}_{2^\omega}$ is the set of all integers between 0 and $2^\omega - 1$. These transformations do the work of diffusion and nonlinear mixing of the input.

The following operations are applied:

- Addition $+$ modulo 2^ω;
- Rotate left (circular left shift) operation, $ROTL^r(X)$, where X is a ω-bit word and r is an integer with $0 \leqslant r < \omega$;
- Bitwise XOR operation $\oplus$ on ω–bit words.

Let $\mathbf{X} = (X_0, X_1, X_2, X_3)$, $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$ and $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ be three 4-tuples of ω–bit words.

Further, let us denote by $*$ the following operation:

$$\mathbf{Z} = \mathbf{X} * \mathbf{Y} \equiv \sigma(\mu(\mathbf{X}) \boxplus_4 \nu(\mathbf{Y})) \tag{1.2}$$

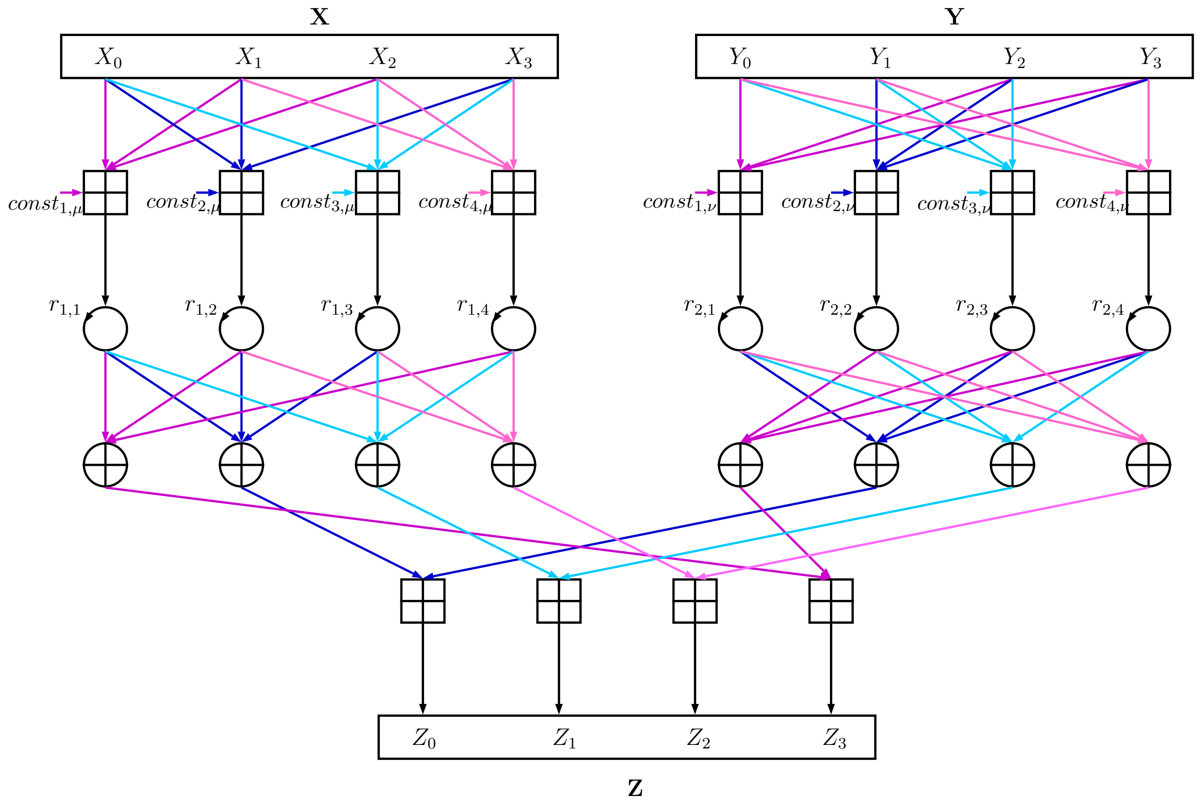

Figure 1.7: Graphical representation of the ARX operation $*$.

Table 1.2: The rotation vectors used in μ and ν.

ω	$\mathbf{r}_{1,\omega}$	$\mathbf{r}_{2,\omega}$
16	(1, 4, 9, 11)	(2, 5, 7, 13)
32	(5, 11, 17, 23)	(3, 10, 19, 29)
64	(7, 19, 31, 53)	(11, 23, 37, 59)

where $\boxplus_4$ is the component-wise addition of two 4-dimensional vectors in $\left(\mathbb{Z}_{2^\omega}\right)^4$.

An algorithmic definition of the $*$ operation over two 4–dimensional vectors $\mathbf{X}$ and $\mathbf{Y}$ for different word sizes is given in Table 1.9, Table 1.10 and Table 1.11.

Also a graphical representation of the $*$ operation is given in Figure 1.7.

The following is a formalization of the $*$ operation (adopted from [8]).

The left rotation of a ω–bit word X by r positions denoted by $ROTL^r(X)$, can be expressed as a linear matrix–vector multiplication over the ring $(\mathbb{Z}_2, +, \cdot)$ i.e. $ROTL^r(X) = \mathbf{E}^r \cdot X$ where $\mathbf{E}^r \in \mathbb{Z}_2^\omega \times \mathbb{Z}_2^\omega$ is a matrix obtained from the identity matrix by rotating its columns by r positions in the direction top to bottom. Further on, if we have a vector $X \in \left(\mathbb{Z}_{2^\omega}\right)^4$ represented as $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and we want to rotate all X_i by r_i $(0 \leqslant i < 4)$ positions to the left, then we denote the operation by $ROTL^{\mathbf{r}}(\mathbf{X})$, where $\mathbf{r} = (r_0, \ldots, r_3) \in \{0, 1, \ldots, \omega - 1\}^4$ is a rotation vector. The operation $ROTL^{\mathbf{r}}(\mathbf{X})$ can also be represented as a linear matrix–vector multiplication over the ring $(\mathbb{Z}_2, +, \cdot)$ i.e. $ROTL^{\mathbf{r}}(\mathbf{X}) = \mathbf{D}^{\mathbf{r}} \cdot \mathbf{X}$ where $\mathbf{D}^{\mathbf{r}} \in \mathbb{Z}_2^{4\omega} \times \mathbb{Z}_2^{4\omega}$,

$$\mathbf{D}^{\mathbf{r}} = \begin{pmatrix} \mathbf{E}^{r_0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{E}^{r_1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{E}^{r_2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{E}^{r_3} \end{pmatrix},$$

and the submatrices $\mathbf{E}^{r_i} \in \mathbb{Z}_2^\omega \times \mathbb{Z}_2^\omega$, $0 \leqslant i < 4$ are obtained from the identity matrix by rotating its columns by r_i positions in the direction top to bottom, and the submatrices $\mathbf{0} \in \mathbb{Z}_2^\omega \times \mathbb{Z}_2^\omega$ are the zero matrices.

Furthermore, we use the following notations:

- $\widehat{\mathbb{A}}_1, \widehat{\mathbb{A}}_3 : \mathbb{Z}_{2^\omega}^4 \to \mathbb{Z}_{2^\omega}^4$ are two bijective transformations in $\mathbb{Z}_{2^\omega}^4$ over the ring $(\mathbb{Z}_{2^\omega}, +, \cdot)$

Table 1.3: The matrices $\widehat{\mathbb{A}}_1$, $\mathbb{A}_2$, $\widehat{\mathbb{A}}_3$ and $\mathbb{A}_4$.

$\widehat{\mathbb{A}}_1$	$\mathbb{A}_2$	$\widehat{\mathbb{A}}_3$	$\mathbb{A}_4$
$\begin{pmatrix} const_{1,\mu\omega} \\ const_{2,\mu\omega} \\ const_{3,\mu\omega} \\ const_{4,\mu\omega} \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \end{pmatrix}$	$\begin{pmatrix} const_{1,\nu\omega} \\ const_{2,\nu\omega} \\ const_{3,\nu\omega} \\ const_{4,\nu\omega} \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \end{pmatrix}$

where $\omega = 8, 16, 32, 64$. The mappings $\widehat{\mathbb{A}}_i, i = 1, 3$ can be described as:

$$\widehat{\mathbb{A}}_i(\mathbf{X}) = \mathbf{C}_i + \mathbb{A}_i \cdot \mathbf{X},$$

where $\mathbf{C}_i \in \mathbb{Z}_{2^\omega}^4, i = 1, 2$ are two constant vectors and $\mathbb{A}_1$ and $\mathbb{A}_3$ are two 4×4 invertible matrices over the ring $(\mathbb{Z}_{2^\omega}, +, \cdot)$. Also these matrices are MDS (maximum distance separable) matrices which redound to have maximal diffusion of the bits. All elements in these two matrices are either 0 or 1, since we want to avoid the operations of multiplication (as more costly microprocessor operations) in the ring $(\mathbb{Z}_{2^\omega}, +, \cdot)$, and stay only with operations of addition.

- $\mathbb{A}_2, \mathbb{A}_4 : \mathbb{Z}_{2^\omega}^4 \to \mathbb{Z}_{2^\omega}^4$ are two linear bijective transformations that are described by two invertible matrices (we use the same notation: $\mathbb{A}_2, \mathbb{A}_4$) of order $q \times q$ over the ring $(\mathbb{Z}_2, +, \cdot)$ $(q = 4\omega)$. Since we want to apply XOR operations on ω–bit registers, the matrices $\mathbb{A}_2$ and $\mathbb{A}_4$ will be of the form

$$\begin{pmatrix} \mathbb{B}_{1,1} & \mathbb{B}_{1,2} & \mathbb{B}_{1,3} & \mathbb{B}_{1,4} \\ \mathbb{B}_{2,1} & \mathbb{B}_{2,2} & \mathbb{B}_{2,3} & \mathbb{B}_{2,4} \\ \mathbb{B}_{3,1} & \mathbb{B}_{3,2} & \mathbb{B}_{3,3} & \mathbb{B}_{3,4} \\ \mathbb{B}_{4,1} & \mathbb{B}_{4,2} & \mathbb{B}_{4,3} & \mathbb{B}_{4,4} \end{pmatrix},$$

  where $\mathbb{B}_{i,j} \in \mathbb{Z}_2^\omega \times \mathbb{Z}_2^\omega,\ \ 1 \leqslant i, j \leqslant 4$ are either the identity matrix or the zero matrix i.e. $\mathbb{B}_{i,j} \in \{\mathbf{0}, \mathbf{1}\}$.

Now we give the formal definitions for the permutations: σ, μ and ν.

Definition 1. *The transformation $\sigma : \mathbb{Z}_{2^\omega}^4 \rightarrow \mathbb{Z}_{2^\omega}^4$ is defined as:*

$$\sigma(X_0, X_1, X_2, X_3) = (X_3, X_0, X_1, X_2)$$

Lemma 1. *The transformation σ is a permutation.* □

Definition 2. *The transformations $\mu : \mathbb{Z}_{2^\omega}^4 \rightarrow \mathbb{Z}_{2^\omega}^4$ and $\nu : \mathbb{Z}_{2^\omega}^4 \rightarrow \mathbb{Z}_{2^\omega}^4$ are defined as:*

$$\mu \equiv \widehat{\mathbb{A}}_1 \circ ROTL^{\mathbf{r}_{1,\omega}} \circ \mathbb{A}_2$$
$$\nu \equiv \widehat{\mathbb{A}}_3 \circ ROTL^{\mathbf{r}_{2,\omega}} \circ \mathbb{A}_4$$

where the rotation vectors $\mathbf{r}_{i,\omega}$, $i = 1, 2$, $\omega = 16, 32, 64$ are given in Table 1.2, and the matrices $\widehat{\mathbb{A}}_1$, $\mathbb{A}_2$, $\widehat{\mathbb{A}}_3$ and $\mathbb{A}_4$ are given in Table 1.3. In Table 1.3, the symbols $\mathbf{1}, \mathbf{0} \in \mathbb{Z}_2^\omega \times \mathbb{Z}_2^\omega$ denote the identity matrix and the zero matrix, and the constants $const_{i,\omega}$, $i = 1, 2$, $\omega = 16, 32, 64$ are given (in hexadecimal notation) in Table 1.4 and Table 1.5. The rationale for choosing these constants is given in Section 5.2.

Table 1.4: List of constants used in μ transformation.

$const_{i,\mu\omega}$	$\mu16$	$\mu32$	$\mu64$
1	0xF0E8	0x8D8B8778	0xF0E8E4E2E1D8D4D2
2	0xE4E2	0x7472716C	0xD1CCCAC9C6C5C3B8
3	0xE1D8	0x6A696665	0xB4B2B1ACAAA9A6A5
4	0xD4D2	0x635C5A59	0xA39C9A999695938E

Table 1.5: List of constants used in ν transformation.

$const_{i,\nu\omega}$	$\nu16$	$\nu32$	$\nu64$
1	0xD1CC	0x5655534E	0x8D8B87787472716C
2	0xCAC9	0x4D4B473C	0x6A696665635C5A59
3	0xC6C5	0x3A393635	0x5655534E4D4B473C
4	0xC3B8	0x332E2D2B	0x3A393635332E2D2B

Lemma 2. *The transformations μ and ν are permutations of $\mathbb{Z}_{2^\omega}, \omega = 16, 32, 64$.*

Proof. The proof follows immediately from the fact that all transformations $\mathbb{A}_i$, $i = 1,2,3,4$ and $ROTL^{\mathbf{r}_{i,\omega}}$, $i = 1,2, \omega = 16,32,64$ are expressed by invertible matrices over the rings $(\mathbb{Z}_{2^\omega}, +, \dots), \omega = 16,32,64$ or over the ring $(\mathbb{Z}_2, +, \dots)$. □

Theorem 1. *The operation* $* : (\mathbb{Z}_{2^\omega}^4)^2 \rightarrow \mathbb{Z}_{2^\omega}^4$ *defined as:*

$$\mathbf{X} * \mathbf{Y} = \sigma(\mu(\mathbf{X}) \boxplus_4 \nu(\mathbf{Y}))$$

is a permutation. □

Let us recall equation (1.1) where the internal state is presented as $IS = (I_1, I_2, \dots, I_N)$. One round of the π function consists of two consecutive transformations E_1 and E_2 defined as follows.

Definition 3. *The function* $E_1 : (\mathbb{Z}_{2^\omega}^4)^{N+1} \rightarrow (\mathbb{Z}_{2^\omega}^4)^N$ *used in the* π *function is defined as:*

$$\begin{aligned} E_1(C, I_1, \dots, I_N) &= (J_1, \dots, J_N), \quad \textit{where} \\ J_1 &= C * I_1, \\ J_i &= J_{i-1} * I_i, \; i = 2, \dots, N \end{aligned} \tag{1.3}$$

where C *is a 4-tuple of* ω*-bit constant values.*

Definition 4. *The function* $E_2 : (\mathbb{Z}_{2^\omega}^4)^{N+1} \rightarrow (\mathbb{Z}_{2^\omega}^4)^N$ *used in* π *function is defined as:*

$$\begin{aligned} E_2(C, I_1, \dots, I_N) &= (J_1, \dots, J_N), \quad \textit{where} \\ J_N &= I_N * C, \\ J_{N-i} &= I_{N-i} * J_{N-i+1}, \; i = 1, \dots, N-1 \end{aligned} \tag{1.4}$$

where C *is a 4-tuple of* ω*-bit constant values.*

Finally, one round of the π function is defined as:

$$\pi(I_1, \dots, I_N) = E_2(C2, E_1(C1, I_1, \dots, I_N)) \tag{1.5}$$

One round of the cipher is graphically described in Figure 1.8. In the figure, the diagonal arrows can be interpreted as $*$ operations between the source and destination, and the vertical or horizontal arrows as equality signs " = ".

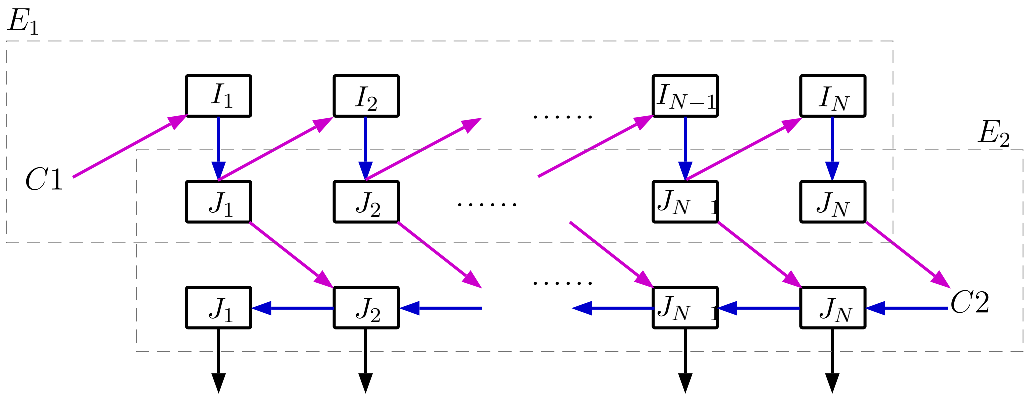

Figure 1.8: One round of π-Cipher

The number of rounds R is a tweakable parameter. We recommend $R = 4$. The complete formula for the π function with $R = 4$ is the following:

$$\pi(I_1, \ldots, I_N) = E_2(C8, E_1(C7, E_2(C6, E_1(C5, E_2(C4, E_1(C3, E_2(C2, E_1(C1, I_1, \ldots, I_N))))))))$$

The constants $C1, C2, \ldots, C8$ are generated in the same way as the constants of the $*$ operation and their values (in hexadecimal notation) for different word sizes are given in Table 1.6, Table 1.7 and Table 1.8.

Table 1.6: Round constants for π16-Cipher

$C1 = \{$0xB4B2, 0xB1AC, 0xAAA9, 0xA6A5$\}$
$C2 = \{$0xA39C, 0x9A99, 0x9695, 0x938E$\}$
$C3 = \{$0x8D8B, 0x8778, 0x7472, 0x716C$\}$
$C4 = \{$0x6A69, 0x6665, 0x635C, 0x5A59$\}$
$C5 = \{$0x5655, 0x534E, 0x4D4B, 0x473C$\}$
$C6 = \{$0x3A39, 0x3635, 0x332E, 0x2D2B$\}$
$C7 = \{$0x271E, 0x1D1B, 0x170F, 0xF0E8$\}$
$C8 = \{$0xE4E2, 0xE1D8, 0xD4D2, 0xD1CC$\}$

Table 1.7: Round constants for π32-Cipher

$C1$ =	{0x8D8B8778, 0x7472716C, 0x6A696665, 0x635C5A59}
$C2$ =	{0x5655534E, 0x4D4B473C, 0x3A393635, 0x332E2D2B}
$C3$ =	{0x271E1D1B, 0x170FF0E8, 0xE4E2E1D8, 0xD4D2D1CC}
$C4$ =	{0xCAC9C6C5, 0xC3B8B4B2, 0xB1ACAAA9, 0xA6A5A39C}
$C5$ =	{0x9A999695, 0x938E8D8B, 0x87787472, 0x716C6A69}
$C6$ =	{0x6665635C, 0x5A595655, 0x534E4D4B, 0x473C3A39}
$C7$ =	{0x3635332E, 0x2D2B271E, 0x1D1B170F, 0xF0E8E4E2}
$C8$ =	{0xE1D8D4D2, 0xD1CCCAC9, 0xC6C5C3B8, 0xB4B2B1AC}

Table 1.8: Round constants for π64-Cipher

$C1$ =	{0x271E1D1B170FF0E8, 0xE4E2E1D8D4D2D1CC, 0xCAC9C6C5C3B8B4B2, 0xB1ACAAA9A6A5A39C}
$C2$ =	{0x9A999695938E8D8B, 0x87787472716C6A69, 0x6665635C5A595655, 0x534E4D4B473C3A39}
$C3$ =	{0x3635332E2D2B271E, 0x1D1B170FF0E8E4E2, 0xE1D8D4D2D1CCCAC9, 0xC6C5C3B8B4B2B1AC}
$C4$ =	{0xAAA9A6A5A39C9A99, 0x9695938E8D8B8778, 0x7472716C6A696665, 0x635C5A595655534E}
$C5$ =	{0x4D4B473C3A393635, 0x332E2D2B271E1D1B, 0x170FF0E8E4E2E1D8, 0xD4D2D1CCCAC9C6C5}
$C6$ =	{0xC3B8B4B2B1ACAAA9, 0xA6A5A39C9A999695, 0x938E8D8B87787472, 0x716C6A696665635C}
$C7$ =	{0x5A595655534E4D4B, 0x473C3A393635332E, 0x2D2B271E1D1B170F, 0xF0E8E4E2E1D8D4D2}
$C8$ =	{0xD1CCCAC9C6C5C3B8, 0xB4B2B1ACAAA9A6A5, 0xA39C9A999695938E, 0x8D8B87787472716C}

Table 1.9: An algorithmic description of the ARX operation $*$ for 16–bit words.

$*$ operation for 16–bit words

Input: $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$
where X_i and Y_i are 16–bit variables.
Output: $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where Z_i are 16–bit variables.
Temporary 16–bit variables: $T_0, \ldots, T_{11}$.

```
$\mu$–transformation for $X$:

1.  $T_0 \leftarrow ROTL^{1}(\texttt{0xF0E8} + X_0 + X_1 + X_2)$;
    $T_1 \leftarrow ROTL^{4}(\texttt{0xE4E2} + X_0 + X_1 + X_3)$;
    $T_2 \leftarrow ROTL^{9}(\texttt{0xE1D8} + X_0 + X_2 + X_3)$;
    $T_3 \leftarrow ROTL^{11}(\texttt{0xD4D2} + X_1 + X_2 + X_3)$;

2.  $T_4 \leftarrow T_0 \oplus T_1 \oplus T_3$;
    $T_5 \leftarrow T_0 \oplus T_1 \oplus T_2$;
    $T_6 \leftarrow T_1 \oplus T_2 \oplus T_3$;
    $T_7 \leftarrow T_0 \oplus T_2 \oplus T_3$;

$\nu$–transformation for $Y$:

1.  $T_0 \leftarrow ROTL^{2}(\texttt{0xD1CC} + Y_0 + Y_2 + Y_3)$;
    $T_1 \leftarrow ROTL^{5}(\texttt{0xCAC9} + Y_1 + Y_2 + Y_3)$;
    $T_2 \leftarrow ROTL^{7}(\texttt{0xC6C5} + Y_0 + Y_1 + Y_2)$;
    $T_3 \leftarrow ROTL^{13}(\texttt{0xC3B8} + Y_0 + Y_1 + Y_3)$;

2.  $T_8 \leftarrow T_1 \oplus T_2 \oplus T_3$;
    $T_9 \leftarrow T_0 \oplus T_2 \oplus T_3$;
    $T_{10} \leftarrow T_0 \oplus T_1 \oplus T_3$;
    $T_{11} \leftarrow T_0 \oplus T_1 \oplus T_2$;

$\sigma$–transformation for both $\mu(X)$ and $\nu(Y)$:

1.  $Z_3 \leftarrow T_4 + T_8$;
    $Z_0 \leftarrow T_5 + T_9$;
    $Z_1 \leftarrow T_6 + T_{10}$;
    $Z_2 \leftarrow T_7 + T_{11}$;
```

Table 1.10: An algorithmic description of the ARX operation $*$ for 32–bit words.

```
$*$ operation for 32–bit words

Input: $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$
where $X_i$ and $Y_i$ are 32–bit variables.
Output: $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where $Z_i$ are 32–bit variables.
Temporary 32–bit variables: $T_0, \ldots, T_{11}$.

$\mu$–transformation for $X$:

    1.  $T_0 \leftarrow ROTL^{5}(\texttt{0x8D8B8778} + X_0 + X_1 + X_2)$;
        $T_1 \leftarrow ROTL^{11}(\texttt{0x7472716C} + X_0 + X_1 + X_3)$;
        $T_2 \leftarrow ROTL^{17}(\texttt{0x6A696665} + X_0 + X_2 + X_3)$;
        $T_3 \leftarrow ROTL^{23}(\texttt{0x635C5A59} + X_1 + X_2 + X_3)$;

    2.  $T_4 \leftarrow T_0 \oplus T_1 \oplus T_3$;
        $T_5 \leftarrow T_0 \oplus T_1 \oplus T_2$;
        $T_6 \leftarrow T_1 \oplus T_2 \oplus T_3$;
        $T_7 \leftarrow T_0 \oplus T_2 \oplus T_3$;

$\nu$–transformation for $Y$:

    1.  $T_0 \leftarrow ROTL^{3}(\texttt{0x5655534E} + Y_0 + Y_2 + Y_3)$;
        $T_1 \leftarrow ROTL^{10}(\texttt{0x4D4B473C} + Y_1 + Y_2 + Y_3)$;
        $T_2 \leftarrow ROTL^{19}(\texttt{0x3A393635} + Y_0 + Y_1 + Y_2)$;
        $T_3 \leftarrow ROTL^{29}(\texttt{0x332E2D2B} + Y_0 + Y_1 + Y_3)$;

    2.  $T_8 \leftarrow T_1 \oplus T_2 \oplus T_3$;
        $T_9 \leftarrow T_0 \oplus T_2 \oplus T_3$;
        $T_{10} \leftarrow T_0 \oplus T_1 \oplus T_3$;
        $T_{11} \leftarrow T_0 \oplus T_1 \oplus T_2$;

$\sigma$–transformation for both $\mu(X)$ and $\nu(Y)$:

    1.  $Z_3 \leftarrow T_4 + T_8$;
        $Z_0 \leftarrow T_5 + T_9$;
        $Z_1 \leftarrow T_6 + T_{10}$;
        $Z_2 \leftarrow T_7 + T_{11}$;
```

Table 1.11: An algorithmic description of the ARX operation $*$ for 64–bit words.

```
$*$ operation for 64–bit words

Input: $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$ where $X_i$ and $Y_i$ are 64–bit variables.
Output: $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where $Z_i$ are 64–bit variables.
Temporary 64–bit variables: $T_0, \ldots, T_{11}$.

$\mu$–transformation for $X$:
    1. $T_0 \leftarrow ROTL^{7}($0xF0E8E4E2E1D8D4D2$ + X_0 + X_1 + X_2)$;
       $T_1 \leftarrow ROTL^{19}($0xD1CCCAC9C6C5C3B8$ + X_0 + X_1 + X_3)$;
       $T_2 \leftarrow ROTL^{31}($0xB4B2B1ACAAA9A6A5$ + X_0 + X_2 + X_3)$;
       $T_3 \leftarrow ROTL^{53}($0xA39C9A999695938E$ + X_1 + X_2 + X_3)$;
    2. $T_4 \leftarrow T_0 \oplus T_1 \oplus T_3$;
       $T_5 \leftarrow T_0 \oplus T_1 \oplus T_2$;
       $T_6 \leftarrow T_1 \oplus T_2 \oplus T_3$;
       $T_7 \leftarrow T_0 \oplus T_2 \oplus T_3$;
$\nu$–transformation for $Y$:
    1. $T_0 \leftarrow ROTL^{11}($0x8D8B87787472716C$ + Y_0 + Y_2 + Y_3)$;
       $T_1 \leftarrow ROTL^{23}($0x6A696665635C5A59$ + Y_1 + Y_2 + Y_3)$;
       $T_2 \leftarrow ROTL^{37}($0x5655534E4D4B473C$ + Y_0 + Y_1 + Y_2)$;
       $T_3 \leftarrow ROTL^{59}($0x3A393635332E2D2B$ + Y_0 + Y_1 + Y_3)$;
    2. $T_8 \leftarrow T_1 \oplus T_2 \oplus T_3$;
       $T_9 \leftarrow T_0 \oplus T_2 \oplus T_3$;
       $T_{10} \leftarrow T_0 \oplus T_1 \oplus T_3$;
       $T_{11} \leftarrow T_0 \oplus T_1 \oplus T_2$;
$\sigma$–transformation for both $\mu(X)$ and $\nu(Y)$:
    1. $Z_3 \leftarrow T_4 + T_8$;
       $Z_0 \leftarrow T_5 + T_9$;
       $Z_1 \leftarrow T_6 + T_{10}$;
       $Z_2 \leftarrow T_7 + T_{11}$;
```

Chapter 2

Security goals

Table 2.1: A list of security goals for the π-Cipher.

Goal	`Pi16Cipher096v1` Bits of security	`Pi16Cipher128v1` Bits of security	`Pi32Cipher128v1` Bits of security	`Pi32Cipher256v1` Bits of security	`Pi64Cipher128v1` Bits of security	`Pi64Cipher256v1` Bits of security
Confidentiality for the plaintext	96	128	128	256	128	256
Confidentiality for the secret message number SMN	96	128	128	256	128	256
Integrity for the plaintext	96	128	128	256	128	256
Integrity for the associated data	96	128	128	256	128	256
Integrity for the public message number PMN	96	128	128	256	128	256
Integrity for the secret message number SMN	96	128	128	256	128	256
Confidentiality for the plaintext M, when $(K, AD, (PMN, SMN_1))$ and $(K, AD, (PMN, SMN_2))$	96	128	128	256	128	256
Integrity for the plaintext M, when $(K, AD, (PMN, SMN_1))$ and $(K, AD, (PMN, SMN_2))$	96	128	128	256	128	256
Tag second preimage resistance $\log_2$ (time, space) complexity	(52, 56)	(52, 56)	(104, 109)	(104, 109)	(208, 214)	(208, 214)

We want to emphasize our position that the security level of 80 bits should be considered as insecure and should be abandoned in future cryptographic designs. This is due to the recent reported practical speed of many different computing systems (GPUs, supercomputers, FPGAs). That is why our lowest level for security is 96 bits of security.

Users have two options for nonces in π-Cipher. A nonce can be $NONCE = PMN$ or $NONCE = (PMN, SMN)$.

If the legitimate key holder uses the same $NONCE$ to encrypt two different pairs of (plaintext, associated data) (M_1, AD) and (M_2, AD) with the same secret key K then the confidentiality and the integrity of the plaintexts are not preserved in π–Cipher. **Thus, the first six goals in Table 2 are achieved under the assumption that PMN is always different for any two different pairs of (plaintext, associated data) (M_1, AD) and (M_2, AD) with the same secret key K.**

Additionally, π-Cipher offers an intermediate level of robustness when a legitimate key holder uses the same secret key K, the same associated data AD, the same public message number PMN but different secret message numbers SMN_1 and SMN_2 for encrypting two different plaintexts M_1 and M_2. In that case confidentiality and integrity of the plaintexts are preserved. **However, in that case the confidentiality of SMN_1 and SMN_2 is not preserved.**

Chapter 3

Security analysis

3.1 Bit diffusion analysis

We give a bit diffusion analysis for the $*$ operation and for one round of the π function of π16-Cipher, π32-Cipher and π64-Cipher.

In our analysis, we have used two experimental settings:

1. Examining the propagation of a one bit difference in a 10000 randomly generated $\mathbf{X}$ and $\mathbf{Y}$ (inputs) of the $*$ operation;

2. Examining the propagation of a one bit difference in a 1000 randomly generated Internal states IS for one round of the π function.

We performed several experiments for different word sizes ($\omega = 16, 32, 64$).

In the experimental setting under (1) we generated 10000 random values for $\mathbf{X}$ and $\mathbf{Y}$. First, we measured what is the Hamming distance between outputs $\mathbf{Z}$ and $\mathbf{Z}'$ ($\mathbf{Z} = \mathbf{X} * \mathbf{Y}$ and $\mathbf{Z}' = \mathbf{X}' * \mathbf{Y}$), where 1 bit is changed in the input $\mathbf{X}$ ($HammingDist(\mathbf{X}, \mathbf{X}') = 1$). After that we measured the Hamming distance between outputs $\mathbf{Z}$ and $\mathbf{Z}'$ of the $*$ operation ($\mathbf{Z} = \mathbf{X} * \mathbf{Y}$ and $\mathbf{Z}' = \mathbf{X} * \mathbf{Y}'$) where 1 bit is changed in $\mathbf{Y}$ ($HammingDist(\mathbf{Y}, \mathbf{Y}') = 1$). The results for $\omega = 16, 32, 64$ are shown in Figure 3.1, Figure 3.2 and Figure 3.3.

In the experimental setting under (2) we generated 1000 random values for the Internal State IS and used just one round of the π function. Here we measure the Hamming distance between the outputs ($\pi(IS)$ and $\pi(IS')$), where one bit is changed in the IS ($HammingDistance(IS, IS') = 1$). The results for $\omega = 16, 32, 64$ are shown in Figure 3.4, Figure 3.5 and Figure 3.6.

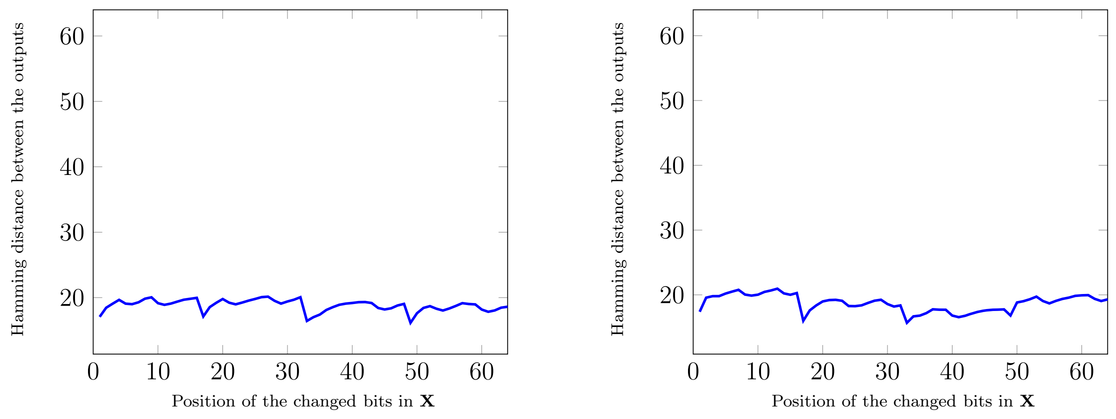

Figure 3.1: Avalanche effect of the $*$ operation for $\omega = 16$

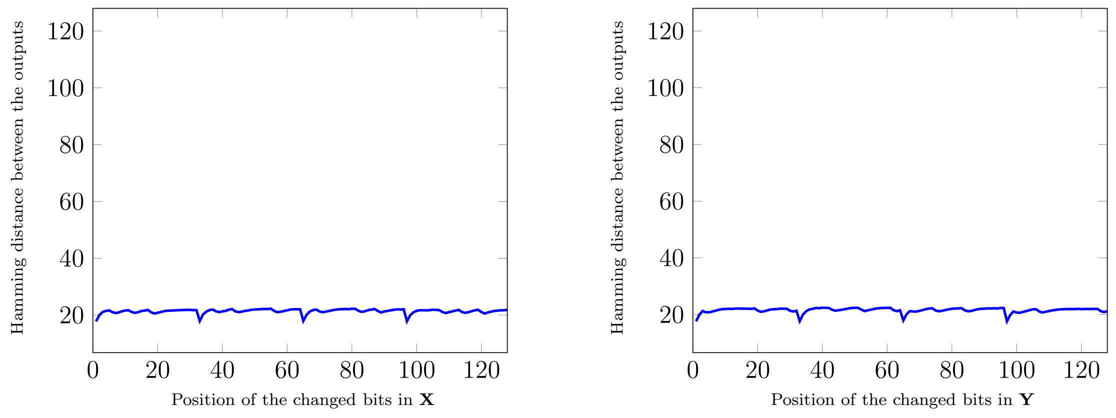

Figure 3.2: Avalanche effect of the $*$ operation for $\omega = 32$

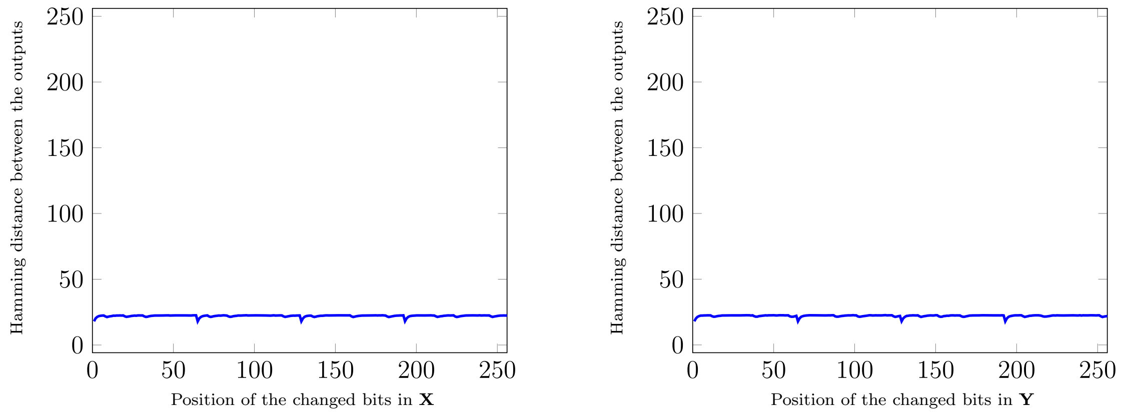

Figure 3.3: Avalanche effect of the $*$ operation for $\omega = 64$

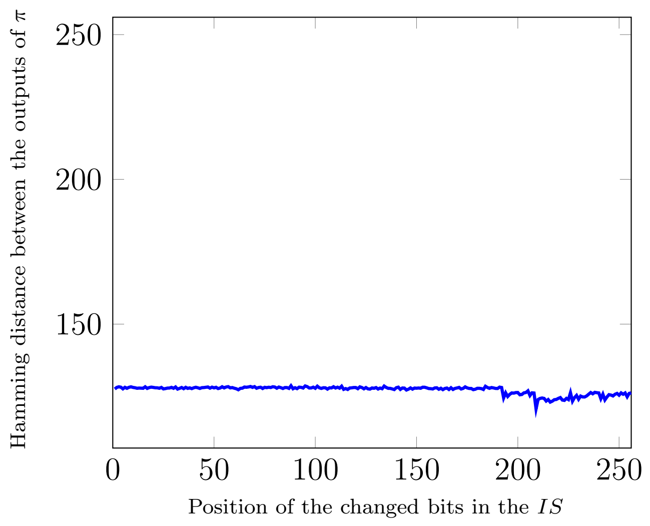

Figure 3.4: Avalanche effect of one round of the π function where $\omega = 16$ ($Min = 120.732$, $\mathbf{Avg = 127.255}$, $Max = 128.731$)

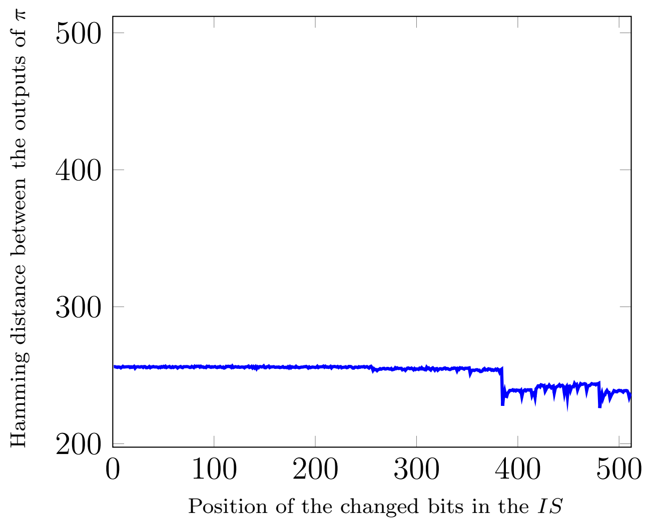

Figure 3.5: Avalanche effect of one round of the π function where $\omega = 32$ ($Min = 226.063$, $\mathbf{Avg = 256.765}$, $Max = 251.472$)

3.2 Distinguisher for one round π16-Cipher096 and π16-Cipher128

We describe a distinguisher attack for one round π16-Cipher096 and π16-Cipher128.

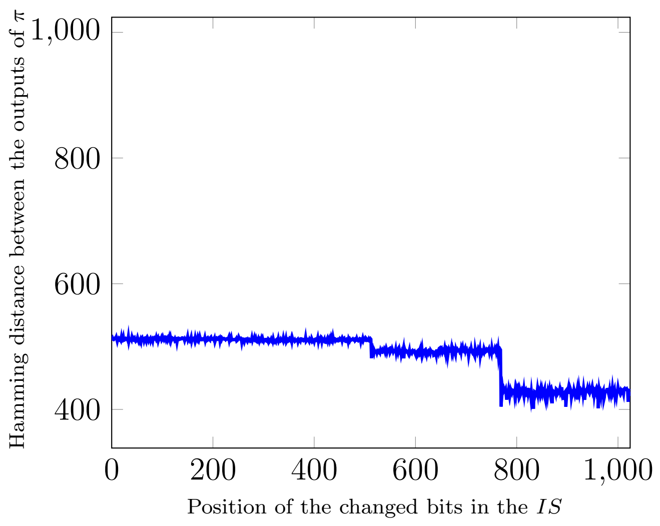

Figure 3.6: Avalanche effect of one round of the π function where $\omega = 64$ ($Min = 400.88$, $\mathbf{Avg = 485.646}$, $Max = 515.76$)

For $\omega = 16$ the size of the part J_2 in Figure 3.7 is 64 bits. Thus, by trying all 2^{64} values for J_2 (the graphical representation is in Figure 3.8) we can obtain in a unique way a list $\mathcal{L}_1 = \{(I_{1i}, I_{2i}) \mid 0 \leqslant i < 2^{64}\}$ of 2^{64} different values for the pairs (I_{1i}, I_{2i}) (the graphical representation is in Figure 3.9). We build a similar list $\mathcal{L}_2 = \{(I'_{1i}, I'_{2i}) \mid 0 \leqslant i < 2^{64}\}$ for the next encrypted block. Knowing that the encryption for the next block is by injecting a counter that was incremented by 1 from the previous counter value, and that it was injected in the position of I_1, while the value of I_2 was the same, we build a matching list $\mathcal{L}_3 = \{(I_{1i}, I'_{1i}, I_{2i}, I'_{2i}) \mid 0 \leqslant i < 2^{64}\}$ where $I'_{1i} - I_{1i} = 1$. With probability 1 there will be a pair of values where $I_{2i} = I'_{2i}$, while if the values of J_1 and J_3 were from an ideal random source, the probability that there will be a pair where $I_{2i} = I'_{2i}$ is very low (much less than 2^{-32}). In total, the complexity of the attack is 2^{65} computations of the operation $*$, and the space is $2^{65} \times 16 = 2^{69}$ bytes.

For more than one round, or for bigger values of ω, this distinguishing attack is more complex than the claimed security of the cipher.

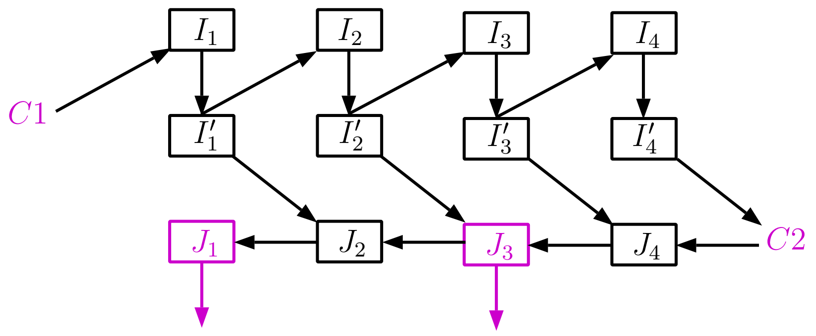

Figure 3.7: One round of π16-Cipher

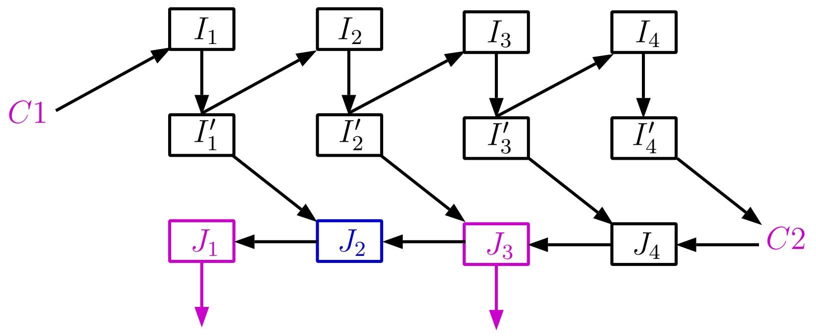

Figure 3.8: For J_2 we try all possible 2^{64} values.

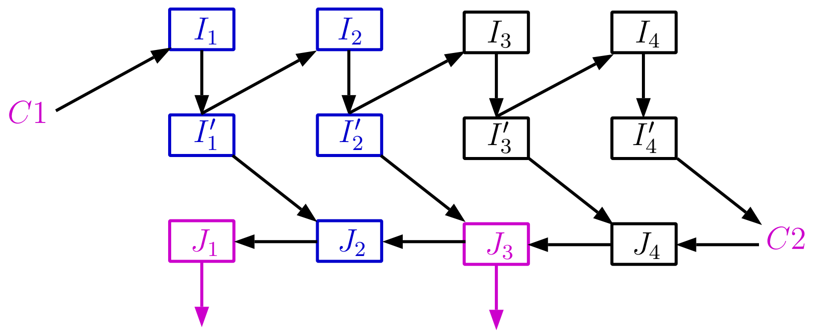

Figure 3.9: The blue squares are either guessed ones (that is J_2) or obtained in a unique way from the definition of the operation $*$.

Chapter 4

Features

In this chapter we discuss the features of π-Cipher loosely following the structure of the features section `http://competitions.cr.yp.to/features.html` on the CAESAR web site.

Cipher-structure (Encrypt than MAC). First we want to point out that it is relatively straightforward to show that the π-Cipher is an Encrypt-then-MAC authenticated cipher. Let us recall the definition for the Encrypt-then-MAC authenticated cipher: We say that the authenticated cipher is Encrypt-then-MAC if a message M is encrypted under a secret key K_1 and then the tag T is calculated with another secret key K_2 as $MAC(K_2, C)$. The pair (C, T) is the output of the authenticated encryption procedure.

If we describe the e-triplex component used in π-Cipher in a mathematical form we have the following. First the message M is encrypted producing the ciphertext C as

$$\begin{aligned} &IS \leftarrow \pi(CIS_{bitrate} \bigoplus counter \ \ |||| \ \ CIS_{capacity}), \\ &C \leftarrow M \bigoplus IS_{bitrate}. \end{aligned}$$

Then, the tag T is calculated as

$$t \leftarrow \pi(C \ \ |||| \ \ IS_{capacity})_{bitrate}.$$

Here, the value of $CIS_{bitrate} \bigoplus counter \ \ |||| \ \ CIS_{capacity}$ has the role of K_1 in the definition of Encrypt-then-MAC, and the value of $C \ \ |||| \ \ IS_{capacity}$ has the role of the pair (K_2, C) in the $MAC(K_2, C)$ part of the definition of Encrypt-then-MAC.

Associated Data and NONCE reuse. If we encrypt two different plaintexts M_1 and M_2 with the same secret key K, associated data AD and nonce $NONCE = (PMN, SMN)$, then neither the confidentiality nor the integrity of the plaintexts are preserved in the π–Cipher. However, as one measure to reduce the risks of a complete reuse of the $NONCE$ we have adopted the strategy of a composite $NONCE = (PMN, SMN)$. If either PMN or SMN are different, then both the confidentiality and integrity of plaintexts are preserved.

Plaintext corruption, associated-data corruption, message-number corruption, ciphertext corruption. We posit that the π-Cipher can straightforwardly be proven INT-CTXT secure under the assumption that the permutation π is an ideal random permutation without any structural distinguishers, by adapting the proof of the XOR-MAC scheme [1]. This is due to the close resemblance of the tag-generation part of the π-Cipher with the XOR-MAC.

Ciphertext prediction. The best distinguishing attack that we know for the π-Cipher is for the versions π16-Cipher096 and π16-Cipher128 with just one round and is described in Section 3.2. The complexity of the attack is 2^{65} computations of the operation $*$, and the space is $2^{65} \times 16 = 2^{69}$ bytes.

Replay and reordering. For the π-Cipher, the standard defense against both replay and reordering is for the sender to use strictly increasing public message numbers PMNs, and for the receiver to refuse any message whose message number is no larger than the largest number of any verified message. This requires both the sender and receiver to keep state.

Sabotage. The π-Cipher puts the encryption of the SMN value as the first block of the ciphertext C. Thus, in protocols that use the π-Cipher, the receiver can make an early reject of invalid messages by decrypting the first block (containing the SMN) and comparing it to its expected value. Only if this check passes the receiver continues with the rest of the decryption and tag computation. Note however, that this requires the protocol to not return error messages to the sender, in order to avoid timing attacks. AES-GCM does not have this property.

Plaintext espionage. Since the attacker's goal here is to figure out the user's secret message, the only feasible attack can happen when the size of the secret message is small by building a table of encrypted secret messages. To defend against this attack the π-Cipher requires the nonce pair $NONCE = (PMN, SMN)$ to have a unique value for every encryption.

Message-number espionage. In the π-Cipher there is a dedicated phase for encrypting the secret message number SMN, and figuring out the value of SMN is equivalent to breaking the whole cipher which is infeasible under the assumptions that the permutation $\pi()$ is random.

General input scheduling. The π-Cipher can offer two ways for reducing the latency: **(1)** If the key K and the public message number PMN are known in advance and used repeatedly, then it is possible to precompute phase 1. and store the resulting Common Internal State (CIS) for subsequent applications of the cipher. **(2)** If the key K, the public message number PMN and the associated data AD are known in advance and used repeatedly, then it is possible to precompute both phase 1. *and* phase 2. for subsequent uses. In both cases, in order to preserve the confidentiality and the integrity of the plaintext, for every encryption the secret message numbers SMN's must be unique.

Incrementality. The π-Cipher is an incremental authenticated cipher. Unlike AES-GCM which also has the incrementality property, the π-Cipher computes the incremented tag with a constant number of operations regardless of the relative position of the changed plaintext block in the whole encrypted plaintext.

Tag second preimage resistance - resistance against finding second preimage for an authentication tag when the key is known (insider attack). In AES-GCM, for a user who knows the secret key, it is a simple task to produce many second preimages for a given tag. For the π-Cipher, even with the knowledge of the secret key K, finding a second preimage of a tag is very expensive with a lower bound of at least 2^{52} operations for the smallest variant, i.e. the π16-Cipher.

In order to see this, recall that the tag is computed as $T = T'' \boxplus_d t_1 \boxplus_d \ldots \boxplus_d t_j \boxplus_d \ldots \boxplus_d t_m$, where t_j is obtained from processing the message and the counter through the e-triplex component. In an insider attack, the attacker knows the key and T'', so w.l.o.g.

we can write

$$T = T'' \boxplus_8 e(1, M_1) \boxplus_8 \ldots \boxplus_8 e(m, M_m) \tag{4.1}$$

where e denotes the partial output of the e-triplex component corresponding to the tag for known CIS and ctr. From the properties of the π-Cipher, we can assume that e, and thus its w-bit components, is preimage resistant hash function. The construction (4.1) can be seen as a modification of the AdHASH function [2], where instead of modular addition, we use simultaneously component-wise modular addition. From [2], a preimage of the tag T can be found by solving a system of 8 modular knapsacks of the following form:

$$\begin{cases} \sum_{i=1}^{m} x_i a_i^1 = b_1 \pmod{2^w} \\ \ldots \\ \sum_{i=1}^{m} x_i a_i^8 = b_8 \pmod{2^w} \end{cases} \tag{4.2}$$

The density of each of the knapsacks in (4.2) is bigger than 1, as is the case for all knapsack based hash functions (see for ex. [9]). This means that for big enough m, we can apply a generalized birthday problem strategy [11], previously also applied in [5] to break the knapsack based hash function [6]. In essence, we generalize the attack from [5] to the case of system of 8 modular knapsacks, and obtain that for suitable choice of $m = 26w$, we can find a preimage with $\mathcal{O}(2^{3.25w})$ time complexity and $\mathcal{O}(w2^{3.25w})$ space complexity.

We note that this complexity is for full tag second preimage. If the tag is chopped, we get a situation equivalent to partial second primage. In this case the complexity will be smaller (it depends on the amount of chopping), since a smaller m will provide a solution to the system of knapsacks. For example, if the tag is chopped in two, i.e. is of size $4w$, then $m = 13w$ will suffice to find a preimage in $\mathcal{O}(2^{1.625w})$ operations.

Software performance. For efficient software implementations, we propose to use the π64-Cipher. On modern Intel CPUs (Sandy Bridge and Haswell) the initial and slightly optimized implementation (but non-SIMD) achieves the speeds from 6.5 cpb up to 12 cpb.

Lightweight hardware performance. For a lightweight hardware implementation we propose to use the π16-Cipher. Our initial and slightly optimized implementation of the basic operation $*$ on FPGA Xilinx Virtex6-XC6VLX240T needs 41 slices and two RAM blocks.

Chapter 5

Design rationale

Disclaimer: ”The designer/designers have not hidden any weaknesses in this cipher.”

5.1 Why parallelism, incrementality and tag second-preimage resistance?

While AES-GCM can be parallelized and can be used to perform incremental updates of the ciphertext and the tag, it is not a tag second-preimage resistant. In our presentation at DIAC 2013 [7] we located several reasons why MACs should retain some hash properties when the key is known. We argued that since the *Robustness* is one of the main goals of the future AEAD standard, tag second-preimage resistance should be one of the features that an AEAD cipher should posses. We also gave two realistic scenarios (”Secure audit logs” and Multi-cast authentication) when the lack of tag second-preimage resistance in authenticated encryption can be exploited.

While the proposed sponge constructions of authenticated encryption offer tag second-preimage resistance, they lack some of the properties that are also useful and desired (such as parallelizability and incrementality). As a result of this line of reasoning, we designed a cipher for authenticated encryption that is parallel, incremental and tag second-preimage resistant.

5.2 Why constants in π–Cipher and how to choose them

In order to avoid the existence of some trivial fixed points in the permutation function π, we decided to use constants in the affine bijective transformations $\widehat{\mathbb{A}_1}$ and $\widehat{\mathbb{A}_3}$ from the μ and ν permutations. The reason why we choose these constants is that they are represented as a sequence of equal distribution of 0s and 1s. Having these constants in $\widehat{\mathbb{A}_1}$ and $\widehat{\mathbb{A}_3}$ we are not aware of any point X such that

$$\pi(\mathbf{X}) = \mathbf{X}.$$

The size and value of the constants depend on the length of the words in the π–Cipher. First we generate the set *Constants* of all possible 8-bit (1 byte) candidates with equal distribution of 0s and 1s in their binary representation. The total number of elements in this set is 70, and is given by:

$$\begin{aligned} Constants = \{&\texttt{0xF0}, \texttt{0xE8}, \texttt{0xE4}, \texttt{0xE2}, \texttt{0xE1}, \texttt{0xD8}, \texttt{0xD4}, \texttt{0xD2}, \texttt{0xD1}, \texttt{0xCC},\\ &\texttt{0xCA}, \texttt{0xC9}, \texttt{0xC6}, \texttt{0xC5}, \texttt{0xC3}, \texttt{0xB8}, \texttt{0xB4}, \texttt{0xB2}, \texttt{0xB1}, \texttt{0xAC},\\ &\texttt{0xAA}, \texttt{0xA9}, \texttt{0xA6}, \texttt{0xA5}, \texttt{0xA3}, \texttt{0x9C}, \texttt{0x9A}, \texttt{0x99}, \texttt{0x96}, \texttt{0x95},\\ &\texttt{0x93}, \texttt{0x8E}, \texttt{0x8D}, \texttt{0x8B}, \texttt{0x87}, \texttt{0x78}, \texttt{0x74}, \texttt{0x72}, \texttt{0x71}, \texttt{0x6C},\\ &\texttt{0x6A}, \texttt{0x69}, \texttt{0x66}, \texttt{0x65}, \texttt{0x63}, \texttt{0x5C}, \texttt{0x5A}, \texttt{0x59}, \texttt{0x56}, \texttt{0x55},\\ &\texttt{0x53}, \texttt{0x4E}, \texttt{0x4D}, \texttt{0x4B}, \texttt{0x47}, \texttt{0x3C}, \texttt{0x3A}, \texttt{0x39}, \texttt{0x36}, \texttt{0x35},\\ &\texttt{0x33}, \texttt{0x2E}, \texttt{0x2D}, \texttt{0x2B}, \texttt{0x27}, \texttt{0x1E}, \texttt{0x1D}, \texttt{0x1B}, \texttt{0x17}, \texttt{0x0F}\} \end{aligned}$$

The constants used in the $\pi\omega$–Ciphers with different word sizes consist of a concatenation of a consecutive 8-bit elements from the set *Constants*.

π16–Cipher uses eight 16-bit constants for the $*$ operation and eight 4-tuples of 16-bit constants for the rounds. We start from the first element of the set `0xF0` and take 16 successive byte values up to the value `0xB8` and form the constants for the $*$ operation. After that, we take every eight successive byte values to form one round constant. We repeat this procedure 8 times and generate the constants $C_1, C_2, \ldots, C_8$ for the rounds.

Since π32–Cipher uses eight 32-bit constants for the $*$ operation and eight 4-tuples of 32-bit constants for the rounds, we take 32 successive byte values starting from the first one `0xF0` and form the constants for the $*$ operation. The next 128 consecutive byte values are used for generating constants for the rounds. Because we need 128 bytes for the round constants of π32–Cipher and have 70 elements in the set, for the 71st byte we take the value of the first element in the set $Constants$ again.

π64–Cipher uses eight 64-bit constants for the $*$ operation. They are taken as 64 successive byte values from the set, starting from the first one `0xF0`. Since π64–Cipher uses 4–tuples of 8 bytes round constants, we take them successively starting from the value `0x27` in the set $Constants$.

The values of both constants (for the $*$ operation and for rounds) for different π–Ciphers are given in Section 1.3.

Chapter 6

Intellectual property

We, the designers of the π-Cipher hereby declare that, to the best of our knowledge, the design of the algorithm that we have submitted for the CAESAR competition, is not covered by any patents. We also hereby declare that we intend never to cover the design of the π-Cipher by any patent.

If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

Chapter 7

Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

References

[1] Mihir Bellare, Roch Guérin, and Phillip Rogaway. Xor macs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1995.

[2] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192. Springer, 1997.

[3] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Proceedings of the 18th International Conference on Selected Areas in Cryptography*, SAC'11, pages 320–337, 2012.

[5] Paul Camion and Jacques Patarin. The knapsack hash function proposed at crypto'89 can be broken. In Donald W. Davies, editor, *EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 1991.

[6] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.

[7] Danilo Gligoroski, Hristina Mihajloska, and Håkon Jacobsen. Should MAC's retain hash properties when the key is known in the next AEAD? Presentation at DIAC 2013, 2013. `http://2013.diac.cr.yp.to/slides/gligoroski.pdf`.

[8] Danilo Gligoroski, Rune Steinsmo Ødegård, Marija Mihova, Svein Johan Knapskog, Ljupco Kocarev, Aleš Drápal, and Vlastimil Klima. Cryptographic hash function EDON-$\mathcal{R}'$. In *1st International Workshop on Security and Communication Networks*, pages 85–95, Trondheim, Norway, May 2009. IEEE.

[9] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In Henri Gilbert, editor, *EUROCRYPT '10*, volume 6110 of *Lecture Notes in Computer Science*, pages 235–256. Springer, 2010.

[10] Pawel Morawiecki and Josef Pieprzyk. Parallel authenticated encryption with the duplex construction. Cryptology ePrint Archive, Report 2013/658, 2013. `http://eprint.iacr.org/`.

[11] David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO '02*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.