Krzysztof CHMIEL

Poznań University of Technology

On Scalable P-box Construction

***Abstract**. In the presented paper are compared the following variants of the scalable diffusion layer in the PP-2 cipher design: auxiliary permutation Prm of the PP-1 cipher, a single rotation ROR1, a multiple rotation ROR2, and involution P of the PP-1 cipher. Permutations Prm, ROR1 and ROR2 are not involutions, and their different, inverse permutations must be used during decryption. Application of them leads to a non-involutional substitution-permutation network.*

***Streszczenie.** W prezentowanym artykule porównano następujące warianty skalowalnej warstwy dyfuzji w projekcie szyfru PP-2: pomocniczą permutację Prm szyfru PP-1, pojedynczą rotację ROR1, wielokrotną rotację ROR2 i inwolucję P szyfru PP-1. Permutacje Prm, ROR1 i ROR2 nie są inwolucjami i ich różne, odwrotne permutacje muszą być użyte podczas deszyfrowania. Ich zastosowanie prowadzi do nieinwolucyjnej sieci podstawieniowo-permutacyjnej. (**O konstrukcji skalowalnego P-bloku**).*

Keywords: block cipher, P-box construction, differential cryptanalysis, linear cryptanalysis.
Słowa kluczowe: szyfr blokowy, konstrukcja P-bloku, kryptoanaliza różnicowa, kryptoanaliza liniowa.

Introduction

In [1, 2, 3, 4, 5, 6] is proposed an n-bit (n = 64, 128, 192, 256, ...) scalable block cipher PP-1 (Fig. 1, Fig. 2), which is an involutional substitution-permutation network (SPN).

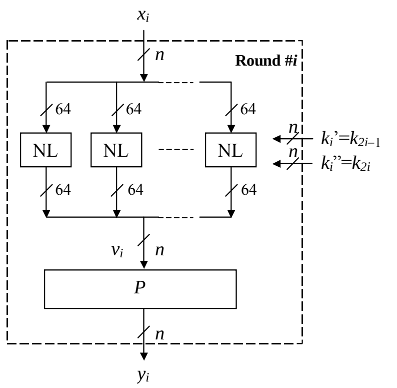

Fig.1. One round of PP-1 (i = 1, 2, ..., r)

PP-1 is a symmetric block cipher designed for platforms with limited resources, especially with restricted amount of memory needed to store its components. It uses one 8×8 bit S-box S, which is an involution (i.e. $S = S^{-1}$), and one n-bit scalable P-box P, which is also an involution (i.e. $P = P^{-1}$). As a result the same network is used in both encryption and decryption phases.

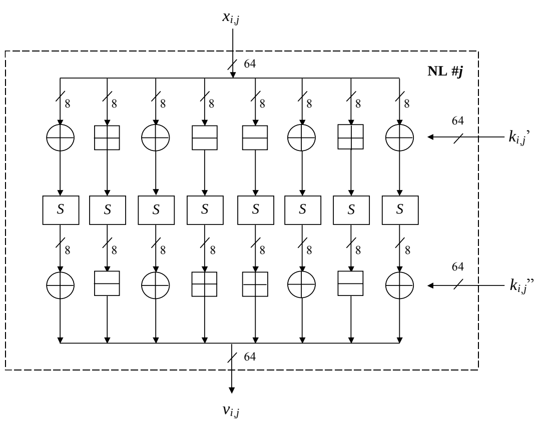

Fig.2. Nonlinear element NL (j = 1, 2, ..., t)

Main role of the permutation P [7, 8] is to scatter 8-bit output subblocks of S-boxes S in the n-bit output block of a round. In round #r, where r depends on n (r = 11, 22, 32, 43, ...), permutation P is the identity operation. For round #i, where i = 1, 2, ..., r−1, the permutation P is constructed using two algorithms, i.e. the auxiliary algorithm (Fig. 3) for the construction of auxiliary permutation Prm, and the main algorithm (Fig. 4) for the construction of permutation (involution) P.

```
Prm(v, nBb, nSb)
{argument, number of block bits (e.g. 64),
  number of S-box bits (e.g. 8)}
1. nS ← nBb div nSb           {number of S-boxes}
2. Sno ← v mod nS +1          {S-box number (from 1)}
3. Sb ← (v − 1) div  nS + 1   {S-box bit (from 1)}
4. y ← (Sno − 1)· nSb + Sb    {value of auxiliary permutation}
5. return y
```

Fig.3. Algorithm for the construction of auxiliary permutation Prm

```
P(pno, nBb, nSb)
{pair number (from 1), number of block bits (e.g. 64),
 number of S-box bits (e.g. 8)}
1. y ← Prm(pno, nBb div 2, nSb div 2)     {value of Prm}
2. pv ← 2· pno − 1    {odd argument (value) of involution}
3. py ← 2· y          {even value (argument) of involution}
4. return (pv, py)
```

Fig.4. Algorithm for the construction of permutation P (rounds #1 to #r−1)

The PP-1 cipher is designed considering its resistance against differential and linear cryptanalysis [9]. In [6] its quality is compared to the quality of a comparative algorithm with the same block length, as well as to the quality of the class of balanced Feistel ciphers, and in particular to DES quality. In [10], however, is presented a differential attack on the PP-1 cipher, with use of multiple differential approximations, which increases the number of required rounds, r, by 1, 2, 4 and 5, respectively (r = 11+1, 22+2, 32+4, 43+5, ...). The redesign of the PP-1 cipher is discussed in [11, 12].

In the presented paper are compared the following variants of the scalable diffusion layer in the PP-2 cipher design: auxiliary permutation Prm of the PP-1 cipher, a single rotation $ROR1$, a multiple rotation $ROR2$, and involution P of the PP-1 cipher. Permutations Prm, $ROR1$ and $ROR2$ are not involutions, and their inverse permutations, which are different, must be used during decryption. Application of them in the diffusion layer implies a non-involutional SPN.

Permutation

The general algorithm Prm, presented in Fig. 3, is used for the construction of a scalable P-box *Prm*. The algorithm calculates bit mappings in permutation *Prm* in order to scatter *nSb*-bit input subblocks of the permutation in its *nBb*-bit output block. The value of *nBb* is assumed to be a multiple of *nSb*, and the number *v* of input bit and the number *y* of output bit belong to the set {1, 2, ..., *nBb*}.

For *nBb* = 64 and *nSb* = 8 permutation *Prm* transforms byte number *i* of the input block, where *i* = 1, 2, ..., 8, in bit number *i* of each byte of the output block, with cyclic shift (rotation) by one byte to the right. E.g., for *i* = 1 we have:

(1) $(1,2,3,4,5,6,7,8) \rightarrow (1,9,17,25, 33,41, 49,57) \rightarrow (9,17,25,33,41,49,57,1).$

In Fig. 5 is shown diffusion for permutation *Prm* in the case of *nBb* = 64 and *nSb* = 8. The round function is restricted to the substitution and permutation layers. For simplicity we assume that the round keys are xored with the input data at each round, and therefore the key addition layers have no influence on diffusion. In an S-box is done the *local diffusion*, i.e., each output bit of an S-box depends on any of its input bits. P-box is responsible for the *global diffusion*, i.e., dependence of each bit of the cipher output block on any bit of its input block. In Fig. 5 by dots are denoted bits dependent on bit number 1 after transformations in consecutive layers. All bits of the output block are dependent on bit number 1 after 3 layers, i.e., after 2 rounds.

In the more general case of *nBb* = $t \cdot 64$ and *nSb* = 8, where *t* = 1, 2, ..., the global diffusion is obtained after *t* + 1 rounds.

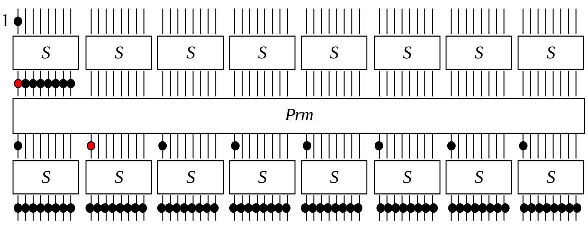

Fig.5. Diffusion for permutation *Prm* (*nBb* = 64, *nSb* = 8)

Permutation *Prm* for *nBb* = 64 and *nSb* = 8, in representation of 64-bit blocks as 8×8 bit matrices, is presented in Tab. 1. The bytes in the rows of the input matrix *A* are denoted by letters from *a* to *h*, and their bits are denoted by digits from 1 to 8. Each row of the input matrix *A* is transformed by permutation *Prm* in one column of the output matrix *B*.

Table 1. Permutation *Prm* for 8×8 bit matrices (*nBb* = 64, *nSb* = 8)

a1	**a2**	**a3**	**a4**	**a5**	**a6**	**a7**	**a8**		**a8**	b8	c8	d8	e8	f8	g8	h8
b1	b2	b3	b4	b5	b6	b7	b8		**a1**	b1	c1	d1	e1	f1	g1	h1
c1	c2	c3	c4	c5	c6	c7	c8		**a2**	b2	c2	d2	e2	f2	g2	h2
d1	d2	d3	d4	d5	d6	d7	d8	*Prm*	**a3**	b3	c3	d3	e3	f3	g3	h3
e1	e2	e3	e4	e5	e6	e7	e8	⇒	**a4**	b4	c4	d4	e4	f4	g4	h4
f1	f2	f3	f4	f5	f6	f7	f8		**a5**	b5	c5	d5	e5	f5	g5	h5
g1	g2	g3	g4	g5	g6	g7	g8		**a6**	b6	c6	d6	e6	f6	g6	h6
h1	h2	h3	h4	h5	h6	h7	h8		**a7**	b7	c7	d7	e7	f7	g7	h7

The basic algorithm computing the value of permutation *Prm*, which performs the transformation *A* => *B* of matrices from Tab. 1, is presented as algorithm 1. The algorithm computes the consecutive bytes of the output matrix *B* by composition of single bits of the input matrix *A*.

Algorithm 1. Basic algorithm computing the value of permutation *Prm* (*nBb* = 64, *nSb* = 8)

```
procedure Prm1(var A,B:Tbyte);
 {type TByte = array[1..8] of byte;}
begin
  B[1] := (A[1] and    1) shl 7    or (A[2] and    1) shl 6 or
          (A[3] and    1) shl 5    or (A[4] and    1) shl 4 or
          (A[5] and    1) shl 3    or (A[6] and    1) shl 2 or
          (A[7] and    1) shl 1    or (A[8] and    1)        ;
  B[2] := (A[1] and 128)           or (A[2] and 128) shr 1 or
          (A[3] and 128)  shr 2    or (A[4] and 128) shr 3 or
          (A[5] and 128)  shr 4    or (A[6] and 128) shr 5 or
          (A[7] and 128)  shr 6    or (A[8] and 128) shr 7;
 .......................................................................................

  B[8] := (A[1] and   2)  shl 6    or (A[2] and   2) shl 5 or
          (A[3] and   2)  shl 4    or (A[4] and   2) shl 3 or
          (A[5] and   2)  shl 2    or (A[6] and   2) shl 1 or
          (A[7] and   2)           or (A[8] and   2) shr 1;
end;
```

Algorithm 2. Fast algorithm computing the value of permutation *Prm* (*nBb* = 64, *nSb* = 8)

```
procedure Prm2(var A,B:TByte);
{type TByte = array[1..8] of byte;}
var x,y,t:LongWord;
begin

  {read A}
  x := A[1] shl 24 or A[2] shl 16 or A[3] shl 8 or A[4];
  y := A[5] shl 24 or A[6] shl 16 or A[7] shl 8 or A[8];

  {rotate A in bytes}
  x := ((x shl 7) and $80808080) or ((x shr 1) and $7F7F7F7F);
  y := ((y shl 7) and $80808080) or ((y shr 1) and $7F7F7F7F);

  {transpose A}
  {16 matrices 2*2}
  t := (x xor (x shr 7)) and $00AA00AA; x := x xor t xor (t shl 7);
  t := (y xor (y shr 7)) and $00AA00AA; y := y xor t xor (t shl 7);
  {4 matrices 2*2}
  t := (x xor (x shr 14)) and $0000CCCC; x := x xor t xor (t shl 14);
  t := (y xor (y shr 14)) and $0000CCCC; y := y xor t xor (t shl 14);
  {1 matrix 2*2}
  t := (x and $F0F0F0F0) or ((y shr 4) and $0F0F0F0F);
  y := ((x shl 4) and $F0F0F0F0) or (y and $0F0F0F0F);
  x := t;

  {write B}
  B[1] := x shr 24; B[2] := x shr 16; B[3] := x shr 8; B[4] := x;
  B[5] := y shr 24; B[6] := y shr 16; B[7] := y shr 8; B[8] := y;

end;
```

For permutation *Prm* there exists a relatively fast software implementation, based on transposition of bit matrices, with previously performed cyclic shift (rotation) of rows by 1 bit to the right. E.g., for byte *a* in Tab. 1 we have:

(2) $(a1,a2,a3,a4,a5,a6,a7,a8) \rightarrow (a8,a1,a2,a3, a4,a5,a6,a7) \rightarrow (a8,a1,a2,a3,a4,a5,a6,a7)^T.$

The fast algorithm computing the value of permutation *Prm*, which performs the transformation *A* => *B* of matrices from Tab. 1, is presented as algorithm 2. The algorithm first reads consecutive bytes of matrix *A* into 32-bit words *x*, *y* and in these words it performs cyclic shift of the bytes by 1 bit to the right. Then, in words *x*, *y*, is done transposition of matrix *A* [13]. After transposition, consecutive bytes of *x*, *y* are written to matrix *B*. Algorithm Prm2 is more than three times faster in comparison to algorithm Prm1 (Tab. 4).

Rotation

Let us first consider the case of a single rotation in the permutation layer of a SPN cipher. For block length *nBb* in bits, which is a multiple of an even number *nSb* of S-box bits, we define the single rotation by *nSb*/2 bits to the right:

(3) $ROR1 = ROR(nSb/2)$.

For the number of S-boxes *nS* = *nBb/nSb* the global diffusion is obtained after *nS* rounds. E.g., in the case of $nBb = t{\cdot}64$ and $nSb = 8$, where $t = 1, 2, ...$, the global diffusion is obtained after $t{\cdot}8$ rounds. Thus, for the single rotation *ROR1* in the permutation layer, the diffusion speed is very low.

Fast algorithm ROR1 computing the value of the single rotation *ROR1*, which performs transformation *A* => *B* of 8×8 bit matrices (*nBb* = 64 and *nSb* = 8), first reads consecutive bytes of matrix *A* into 32-bit words *x*, *y*. Then, is performed cyclic shift (rotation) of the 64-bit word *x*||*y* by 4 bits to the right. Finally, the consecutive bytes of *x*, *y* are written to matrix *B*. Algorithm ROR1 is more than five times faster in comparison to the basic algorithm, similar to algorithm Prm1 (Tab. 4).

Let us now consider the case of a multiple rotation in the permutation layer of a SPN cipher. For block length in bits $nBb = t{\cdot}64$ ($t = 1, 2, ...$) and the number of S-box bits $nSb = 8$, we define the multiple rotation to the right:

(4) $ROR2 = ROR(12, [1]) + ROR(28, [2]) + ROR(44, [3]) + ROR(60, [4])$,

where *ROR* denotes the rotation by a specified number of bits to the right for the following classes of bits:

(5) $[1] = \{1, 5, ..., t{\cdot}64–3\}$, $[2] = \{2, 6, ..., t{\cdot}64–2\}$, $[3] = \{3, 7, ..., t{\cdot}64–1\}$, $[4] = \{4, 8, ..., t{\cdot}64–0\}$.

In Fig. 6 is presented diffusion for rotation *ROR2* in the case of *nBb* = 64 and *nSb* = 8. All bits of the output block are dependent on bit number 1 after 3 layers, i.e., after 2 rounds. Rotation *ROR2* transforms bits numbered 1–8, dependent on bit number 1 after substitution *S*, as follows:

(6)
$ROR2(1) = ROR(12, 1) = 13$,
$ROR2(2) = ROR(28, 2) = 30$,
$ROR2(3) = ROR(44, 3) = 47$,
$ROR2(4) = ROR(60, 4) = 64$,
$ROR2(5) = ROR(12, 5) = 17$,
$ROR2(6) = ROR(28, 6) = 34$,
$ROR2(7) = ROR(44, 7) = 51$,
$ROR2(8) = ROR(60, 8) = 4$.

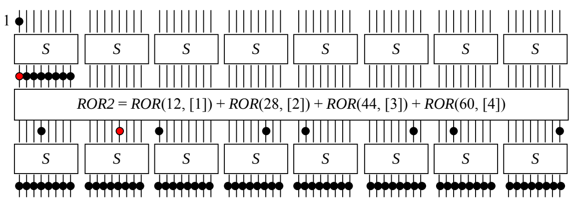

Fig.6. Diffusion for multiple rotation *ROR2* (*nBb* = 64, *nSb* = 8)

Multiple rotation *ROR2* for *nBb* = 64 and *nSb* = 8, in representation of 64-bit blocks as 8×8 bit matrices, is presented in Tab. 2. Each row of the input matrix *A* is transformed by rotation *ROR2* into eight rows of the output matrix *B*.

Table 2. Multiple rotation *ROR2* for 8×8 bit matrices (*nBb* = 64, *nSb* = 8)

a1	**a2**	**a3**	**a4**	**a5**	**a6**	**a7**	**a8**		g5	e6	c7	**a8**	h1	f2	d3	b4
b1	b2	b3	b4	b5	b6	b7	b8		h5	f6	d7	b8	**a1**	g2	e3	c4
c1	c2	c3	c4	c5	c6	c7	c8		**a5**	g6	e7	c8	b1	h2	f3	d4
d1	d2	d3	d4	d5	d6	d7	d8	*ROR2*	b5	h6	f7	d8	c1	**a2**	g3	e4
e1	e2	e3	e4	e5	e6	e7	e8	⇒	c5	**a6**	g7	e8	d1	b2	h3	f4
f1	f2	f3	f4	f5	f6	f7	f8		d5	b6	h7	f8	e1	c2	**a3**	g4
g1	g2	g3	g4	g5	g6	g7	g8		e5	c6	**a7**	g8	f1	d2	b3	h4
h1	h2	h3	h4	h5	h6	h7	h8		f5	d6	b7	h8	g1	e2	c3	**a4**

Variant 1 of the fast algorithm computing the value of the multiple rotation *ROR2*, which performs transformation *A* => *B* of matrices from Tab. 2, is presented as algorithm 3. The algorithm first reads consecutive bytes of matrix *A* into 32-bit words *x*, *y*. Then, is performed cyclic shift (rotation) to the right of the classes of bits [1], [2], [3] and [4] in the 64-bit word *x*||*y*, by 12, 28, 44 and 60 bits, respectively. Finally, the consecutive bytes of *x*, *y* are written to matrix *B*.

Algorithm 3. Fast algorithm computing the value of rotation *ROR2* – variant 1 (*nBb* = 64, *nSb* = 8)

```
procedure ROR21(var A,B:TByte);
{type TByte = array[1..8] of byte;}
var x,y,t:LongWord;
begin

  {read A}
  x := A[1] shl 24 or A[2] shl 16 or A[3] shl 8 or A[4];
  y := A[5] shl 24 or A[6] shl 16 or A[7] shl 8 or A[8];

  {rotate classes [1],[2],[3],[4] of x||y by 12,28,44,60 bits}
  t := ((x and $88888000) shr 12) or ((y and $00000888) shl 20) or
       ((x and $40000000) shr 28) or ((y and $04444444) shl 04) or
       ((y and $22222000) shr 12) or ((x and $00000222) shl 20) or
       ((y and $10000000) shr 28) or ((x and $01111111) shl 04);

  y := ((y and $88888000) shr 12) or ((x and $00000888) shl 20) or
       ((y and $40000000) shr 28) or ((x and $04444444) shl 04) or
       ((x and $22222000) shr 12) or ((y and $00000222) shl 20) or
       ((x and $10000000) shr 28) or ((y and $01111111) shl 04);
  x := t;

  {write B}
  B[1] := x shr 24; B[2] := x shr 16; B[3] := x shr 8; B[4] := x;
  B[5] := y shr 24; B[6] := y shr 16; B[7] := y shr 8; B[8] := y;

end;
```

Variant 2 of the fast algorithm computing the value of the multiple rotation *ROR2*, which performs transformation *A* => *B* of matrices from Tab. 2, is presented as algorithm 4. The algorithm first reads consecutive bytes of matrix *A* into 32-bit words *x*, *y*. Then, is performed cyclic shift (rotation) of the 64-bit word *x*||*y* by 12 bits to the right, and are calculated classes [1] and [3] of bits. Next, is done rotation to the right of the word *x*||*y* by another 16 bits, and are calculated classes [2] and [4]. In consequence, the classes [1], [2], [3] and [4] are rotated by 12, 28, 44 and 60 bits, respectively. Finally, the consecutive bytes of *x*, *y* are written to matrix *B*.

Algorithm 4. Fast algorithm computing the value of rotation *ROR2* – variant 2 (*nBb* = 64, *nSb* = 8)

```
procedure ROR22(var A,B:TByte);
{type TByte = array[1..8] of byte;}
var x,y,t,wx,wy:LongWord;
begin

  {read A}
  x := A[1] shl 24 or A[2] shl 16 or A[3] shl 8 or A[4];  {ABCDEFGH}
  y := A[5] shl 24 or A[6] shl 16 or A[7] shl 8 or A[8];  {IJKLMNOP}
```

```
  {rotate classes [1],[2],[3],[4] of x||y by 12,28,44,60 bits}
  {rotate x||y by 12 bits}
  wx := (x shr 12) or (y shl 20);             {FGHABCDE}
  wy := (y shr 12) or (x shl 20);             {NOPIJKLM}
  x := wx and $88888888 or wy and $22222222;        {12, 44}
  y := wy and $88888888 or wx and $22222222;        {12, 44}
  {rotate wx||wy by 16 bits}
  t := wx;
  wx := (wx shr 16) or (wy shl 16);           {BCDEFGHA}
  wy := (wy shr 16) or (t shl 16);            {JKLMNOPI}
  x := x or wx and $44444444 or wy and $11111111;  {28, 60}
  y := y or wy and $44444444 or wx and $11111111;  {28, 60}

  {write B}
  B[1] := x shr 24; B[2] := x shr 16; B[3] := x shr 8; B[4] := x;
  B[5] := y shr 24; B[6] := y shr 16; B[7] := y shr 8; B[8] := y;

end;
```

Involution

In involutional SPN ciphers the same algorithm is used for encryption and decryption. It is possible thanks to the fact that in designing these ciphers are applied involutional components, and in particular involutional P-boxes.

The general algorithm P, presented in Fig. 4, is used for the construction of a scalable involutional P-box P (i.e. $P^{-1} = P$). The algorithm calculates involutional pairs of bit mappings in *nBb*-bit involution P for the bit mappings in scalable, auxiliary permutation *Prm* (Fig. 3). Similar to the algorithm Prm, the value of *nBb* is assumed to be a multiple of *nSb*, and the number *pno* of calculated involutional pairs (*pv*, *py*) belongs to the set {1, 2, ..., *nBb*/2}, where $pv, py \in \{1, 2, ..., nBb\}$. An additional assumption is that *nSb* and, in consequence, *nBb* are even.

In Fig. 7 is shown diffusion for involution P in the case of *nBb* = 64 and *nSb* = 8. All bits of the output block are dependent on bit number 1 after 5 layers, i.e., after 3 rounds.

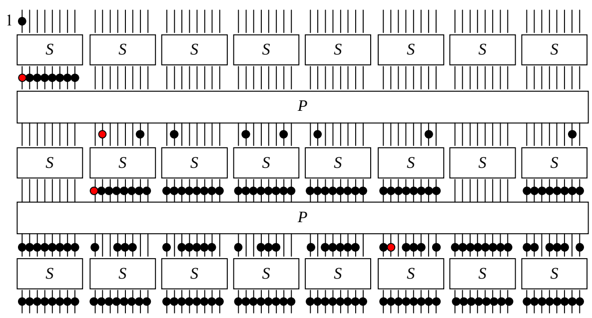

Fig.7. Diffusion for involution P (*nBb* = 64, *nSb* = 8)

Involution P for *nBb* = 64 and *nSb* = 8, in representation of 64-bit blocks as 8×8 bit matrices, is presented in Tab. 3. Each row of the input matrix A is transformed by involution P into two columns of the output matrix B.

Table 3. Involution P for 8×8 bit matrices (*nBb* = 64, *nSb* = 8)

a1	**a2**	**a3**	**a4**	**a5**	**a6**	**a7**	**a8**		b2	b7	c2	d7	d2	f7	e2	h7
b1	b2	b3	b4	b5	b6	b7	b8		f2	**a1**	g2	c1	h2	e1	**a2**	g1
c1	c2	c3	c4	c5	c6	c7	c8		b4	**a3**	c4	c3	d4	e3	e4	g3
d1	d2	d3	d4	d5	d6	d7	d8	P	f4	**a5**	g4	c5	h4	e5	**a4**	g5
e1	e2	e3	e4	e5	e6	e7	e8	$\Rightarrow$	b6	**a7**	c6	c7	d6	e7	e6	g7
f1	f2	f3	f4	f5	f6	f7	f8		f6	b1	g6	d1	h6	f1	**a6**	h1
g1	g2	g3	g4	g5	g6	g7	g8		b8	b3	c8	d3	d8	f3	e8	h3
h1	h2	h3	h4	h5	h6	h7	h8		f8	b5	g8	d5	h8	f5	**a8**	h5

The basic algorithm, InvP1, computing the value of involution P, which performs the transformation $A \Rightarrow B$ of matrices from Tab. 3, similarly to algorithm 1, computes the consecutive bytes of the output matrix B by composition of single bits of the input matrix A.

Algorithm 5. Fast algorithm computing the value of involution P (*nBb* = 64, *nSb* = 8)

```
procedure InvP2(var B,A:TByte);
{type TByte = array[1..8] of byte;}
var x,y,w,z,t,w1,w2,z1,z2:LongWord;
begin

  {read B}
  x := B[1] shl 24 or B[2] shl 16 or B[3] shl 8 or B[4];
  y := B[5] shl 24 or B[6] shl 16 or B[7] shl 8 or B[8];

  {transpose B}
  {16 matrices 2*2}
  t := (x xor (x shr 7)) and $00AA00AA; x := x xor t xor (t shl 7);
  t := (y xor (y shr 7)) and $00AA00AA; y := y xor t xor (t shl 7);
  {4 matrices 2*2}
  t := (x xor (x shr 14)) and $0000CCCC; x := x xor t xor (t shl 14);
  t := (y xor (y shr 14)) and $0000CCCC; y := y xor t xor (t shl 14);
  {1 matrix 2*2}
  t := (x and $F0F0F0F0) or ((y shr 4) and $0F0F0F0F);
  y := ((x shl 4) and $F0F0F0F0) or (y and $0F0F0F0F);
  x := t;

  {odd bits}
  w := (x and $00FF00FF) shl 8 or (y and $00FF00FF);
                {all odd bits }
  w := (w and $7F7F7F7F) shl 1 or (w and $80808080) shr 7;
                {abefcdgh - odd after rotation in bytes}
  t := (w xor (w shr 2)) and $0C0C0C0C; w := w xor t xor (t shl 2);
                {2-bit shuffle}
  t := (w xor (w shr 1)) and $22222222; w := w xor t xor (t shl 1);
                {a-b e-f c-d g-h - odd shuffled}
  w1 :=  w and $AAAAAAAA;          {a e c g - odd in bytes}
  w2 := (w and $55555555) shl 1;   {b f d h - odd in bytes}

  {even bits}
  z := (x and $FF00FF00) or (y and $FF00FF00) shr 8;
                {b-f d-h c-g e-a - even shuffled}
  z1 := (z and $AAAAAAAA) shr 1;   {b d c e - even in bytes}
  z2 :=  z and $55555555;          {f h g a - even in bytes}

  {write A}
  A[1] := (w1 shr 24) or z2;          A[2] := (w2 shr 24) or (z1 shr 24);
  A[3] := (w1 shr 8) or (z1 shr 8);   A[4] := (w2 shr 8) or (z1 shr 16);
  A[5] := (w1 shr 16) or z1;          A[6] := (w2 shr 16) or (z2 shr 24);
  A[7] :=  w1 or (z2 shr 8);          A[8] :=  w2 or (z2 shr 16);

end;
```

For involution P there exists a relatively fast software implementation based on transposition of bit matrices. Since P is an involution, the transformation $A \Rightarrow B$ of matrices is the same as the transformation $B \Rightarrow A$. The fast algorithm InvP2 computing the value of involution P, which performs the transformation $A \Rightarrow B$ of matrices from Tab. 3, is presented as algorithm 5. The algorithm first reads consecutive bytes of matrix B into 32-bit words x, y. Then, in words x, y, is done transposition of matrix B [13]. After transposition are separately processed odd and even bits. Finally, consecutive bytes of x, y are written to matrix A. Algorithm InvP2 is about two times faster in comparison to the basic algorithm InvP1 (Tab. 4).

Conclusion

In the paper are compared the following scalable diffusion layer functions: auxiliary permutation *Prm* of the PP-1 cipher, a single rotation *ROR1*, a multiple rotation *ROR2*, and involution P of the PP-1 cipher. Permutations

Prm, *ROR1* and *ROR2* are not involutions, and their inverse permutations, which are different, must be used during decryption.

Considering the resistance against cryptanalysis, permutation *Prm*, rotation *ROR1* and rotation *ROR2* are comparable, and better than involution *P*.

Considering the diffusion speed, rotation *ROR1* is worse than involution *P*, and permutation *Prm* and rotation *ROR2* are of the same quality, and better than involution *P*.

Considering the software implementation speed, the best is rotation *ROR1*, and permutation *Prm* and rotation *ROR2* are comparable, and better than involution *P* (Tab. 4).

Table 4. Time of $100 \cdot 10^6$ calculations of diffusion layer functions (*nBb* = 64, *nSb* = 8)

Algorithm	Prm1	Prm2	ROR1	ROR21	ROR22	InvP1	InvP2
Time [s]	11.31	3.406	2.140	3.625	3.187	9.859	5.859

Acknowledgement

This work was supported by the Polish Ministry of Science and Higher Education as a 2010–2013 research project.

REFERENCES

[1] Bucholc K., Chmiel K., Grocholewska-Czuryło A., Stokłosa J., PP-1 Block Cipher, *Polish Journal of Environmental Studies*, 16 (2007), No. 5B, 315−320; also in: *Proceedings of 14th International Multi-Conference on Advanced Computer Systems ACS 2007*, CD, Szczecin 2007

[2] Chmiel K., Grocholewska-Czuryło A., Stokłosa J., Involutional Block Cipher for Limited Resources, *Proceedings of the 2008 IEEE Global Telecommunications Conference, Computer and Communications Network Security Symposium*, GLOBECOM 2008, IEEE eXpress Conference Publishing, 1852−1856, New Orleans 2008

[3] Chmiel K., Grocholewska-Czuryło A., Socha P., Stokłosa J., Scalable Cipher for Limited Resources, *Polish Journal of Environmental Studies*, 17 (2008), No. 4C, 371−377

[4] Chmiel K., Grocholewska-Czuryło A., Socha P., Stokłosa J., Involutional scalable block cipher, *Metody Informatyki Stosowanej*, 16 (2008), nr 3, 65−75

[5] Bucholc K., Chmiel K., Grocholewska-Czuryło A., Idzikowska E., Janicka-Lipska I., Stokłosa J., Scalable PP-1 Block Cipher, *International Journal of Applied Mathematics and Computer Science*, 20 (2010), No. 2, 401–411

[6] Chmiel K., Grocholewska-Czuryło A., Stokłosa J., Evaluation of PP-1 Cipher Resistance against Differential and Linear Cryptanalysis in Comparison to a DES-like Cipher, *Fundamenta Informaticae*, 114 (2012), No. 3-4, 239–269

[7] Chmiel K., Obliczanie 64-bitowej permutacji P szyfru PP-1, Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, Raport 562, 1–21, Poznań 2008

[8] Chmiel K., Obliczanie permutacji P szyfru PP-1 dla bloku o długości 128 i 256 bitów, Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, Raport 563, 1–31, Poznań 2008

[9] Chmiel K., Metody różnicowej i liniowej kryptoanalizy szyfrów blokowych, Rozprawa habilitacyjna Nr 443, Wydawnictwo Politechniki Poznańskiej, 1–212, Poznań 2010

[10] Misztal M., Differential Cryptanalysis of PP-1 Cipher, *Proceedings of International Cryptology Conference – Recent Advances in Cryptology and National Telecommunication Security Systems*, MUT, Warsaw 2011; also in: *Annales UMCS Informatica AI XI*, 2 (2011), 9–24

[11] Chmiel K., Koncepcja funkcji rundy szyfru PP-2, Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, Raport 619, 1–18, Poznań 2011

[12] Chmiel K., Koncepcja warstwy dyfuzji szyfru PP-2, Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, Raport 622, 1–37, Poznań 2011

[13] Warren H.S., Uczta programistów, Helion, 1–334, Gliwice 2003

***Author**: dr inż. Krzysztof Chmiel, Politechnika Poznańska, Instytut Automatyki i Inżynierii Informatycznej, pl. Marii Skłodowskiej-Curie 5, 60-965 Poznań, E-mail: krzysztof.chmiel@put.poznan.pl.*

Nowe książki