Special block cipher family DN and new generation SNMAC-type hash function family HDN

This paper presents some parts of the project Special Block Cipher (ST20052006018) for Czech NSA

Vlastimil KLÍMA*, January 2007

* Independent consultant, v.klima (at) volny.cz, http://cryptography.hyperlink.cz. The project ST20052006018 was finished on September 30, 2006.

Abstract

Special block cipher is a new cryptographic primitive designed to be a building block of the new generation hash functions SNMAC [Kl06]. Contrary to classical block ciphers it is knowingly designed focusing to those properties which are expected to be just a “side effect” on usual cipher constructions. Its design anticipates that an attacker could exploit or know its key, or even he/she could discretionarily tamper with the key. The design criteria of SNMAC hash functions are publicly known. Limitly, these functions approach a random oracle, they are computationally resistant against pre-image and collision attacks, and different special block cipher instances can be used in their design.

In this paper, we present special block cipher family Double Net DN(n, k)-ρ with n-bit block, k-bit key and ρ rounds, their building blocks construction principles and design criteria. Based on DN, we define hash functions family HDN(n, k)-ρ with n-bit hash code working on blocks of k - n bits.

We introduce and propose to use DN(512, 8192)-10 and HDN(512, 8192)-10 as example instances. It turns out these are not just theoretical concepts, but practically employable functions with speeds only 2-3 times lower than SHA-512 and Whirlpool.

Basic idea behind the special block cipher DN is simple – contrary to classical block cipher approach, the same attention is paid to key and plaintext processing. One SP network ensures key mixing, while the second one mixes the plaintext with the key.

Once the special block cipher concept is examined and accepted in hash functions, it can be used in advance in its original purpose – data encryption. We suggest the transition from the classical block ciphers to more secure special block ciphers in the future. Its advantage is its readiness for various attacks on the secret key; the attacks which have recently started to emerge in classical block cipher cryptanalysis. Among others, these include side-channel attacks, related keys attacks and rectangular attacks (see e.g. [Bi93], [Bi03], [Ki04], [Ho05], [Ki05], [Bi05], and [Bi06]). With the expansion of the cryptographic instruments and cryptanalytic methods, these attacks will appear more and more frequently. Their common traits are the various attempts to exploit the original assumption on the attacker’s limited power over the secret key or its knowledge. The defence against these attacks is illustrated by the evolution of the functions processing the secret key, starting with simple copy-type functions used in DES and TripleDES to weak non-linear functions in AES. We believe that this trend will continue to strong non-linear functions (similar to the ones used in DN). The employment of these stronger functions in the encryption might not seem as a must in the present, but it probably will be in the future. In the hash functions, it is a necessity today already.

Contents

1. Introduction ... 4
2. Double Net functions family description ... 6
3. Network Π construction ... 11
4. Network Φ construction ... 19
5. Double Net as a strengthened encryption algorithm ... 23
6. Number of rounds in DN, its variations and hashing speed ... 24
7. Conclusion ... 26
8. References ... 27

9. Appendix A: SP networks theory ... 29
10. Appendix B: Definitions of variable elements in DN(512,8192) ... 32
11. Appendix C: Description of variable elements in HDN(512, 8192) ... 39
12. Appendix D: Original source codes of DN(512, 8192) and HDN(512, 8192) ... 40
13. Appendix E: Test vectors for DN(512, 8192) and HDN(512, 8192) ... 65

1. Introduction

An attacker of a hash function that employs a block cipher has the possibility to manipulate with the plaintext and with the key, as well. The primary goal of classical block ciphers design is not the resistance to this kind of attack – some resistance is present, it can be seen as "side effect" only, however. As the result, new generation hash functions SNMAC [Kl06] employ the special block cipher in their compression function. Once the special block cipher concept is examined and accepted in hash functions, it can be used in advance in its original purpose – data encryption. We suggest the transition from the classical block ciphers to more secure special block ciphers in the future.

Classical block cipher is the cryptographic primitive designed to protect the plaintext and its structure in the ciphertext using the secret encryption key. The fact the attacker does not know the secret key is essential for high-speed encryption in the classical block cipher construction. In most cases, very few (if any) simple modifications of the key are present. The key expansion procedure is extremely simple in the most of classical block cipher designs. For instance, DES uses a simple copy function, while AES a weak non-linear transformation. The majority of block ciphers use weak non-linear or simple functions. These weaknesses were crucially exploited in the attacks on MD and SHA hash function families. They allowed controlling many places of the inner state of the hash function while following pre-made strategy (differential path). Strong non-linearity would not have allowed this exploit.

So far, this vulnerability was not used to attack classical block ciphers, as the manipulation with the key to such extent is not possible in real life scenarios.

In the case of classical block ciphers, in the beginning it was assumed the attacker has no knowledge about the plaintext, later it was admitted he could know or even choose some of its parts. Currently, full control over the plaintext and ciphertext is taken into account. As an answer to these possibilities of the attacker, strong non-linear functions processing the plaintext were introduced.

Unfortunately, it was and still is assumed the attacker does not know the encryption key and has no means to manipulate with it. The technology development and the birth of various encryption devices (smart-cards, SSL servers, cryptographic modules, libraries, etc.) provide attacker with new possibilities weakening both of these original assumptions – not knowing the key and the impossibility to manipulate with it, as well. The most common origins of these new possibilities are various side-channels (power, electromagnetic, timing …) that allow manipulating with the key and provide its partial knowledge, as well.

Today's linear or weak non-linear processing of the key does not protect it against such attacks. The progress in the decades to come will undoubtedly show similar advancements in the key exploiting attacks. In order to have strong block ciphers in the future, it is advisable to strengthen their key processing functions.

To prevent the future attacks on classical block ciphers, functions used during the key expansion procedure should have the same security properties as the plaintext processing functions. Key and plaintext processing functions should have the same resistance against the differential and linear cryptanalysis and against other attacks, as well. Over time, the techniques used to attack the key will be similar to the ones used to attack plaintext today. However, it might take decades for these attacks to appear. Thus, the question is when to start applying the relevant countermeasures.

Key manipulation possibilities fully arose when classical block cipher was used in the hash function construction. As there is no secret element in the hash function computation, the attacker can manipulate with all of its inputs, thus with the key, as well. The countermeasures in hash function constructions need to be applied immediately, since these possibilities are at attacker's disposal already today.

For this reason the special block cipher and new generation hash function SNMAC concept were designed. We describe the first class of special block ciphers DN in this paper and class of hash functions HDN based on them.

2. Double Net functions family description

The description of Double Net DN(n, k)-ρ block ciphers family, constructions principles and design criteria are presented in this chapter.

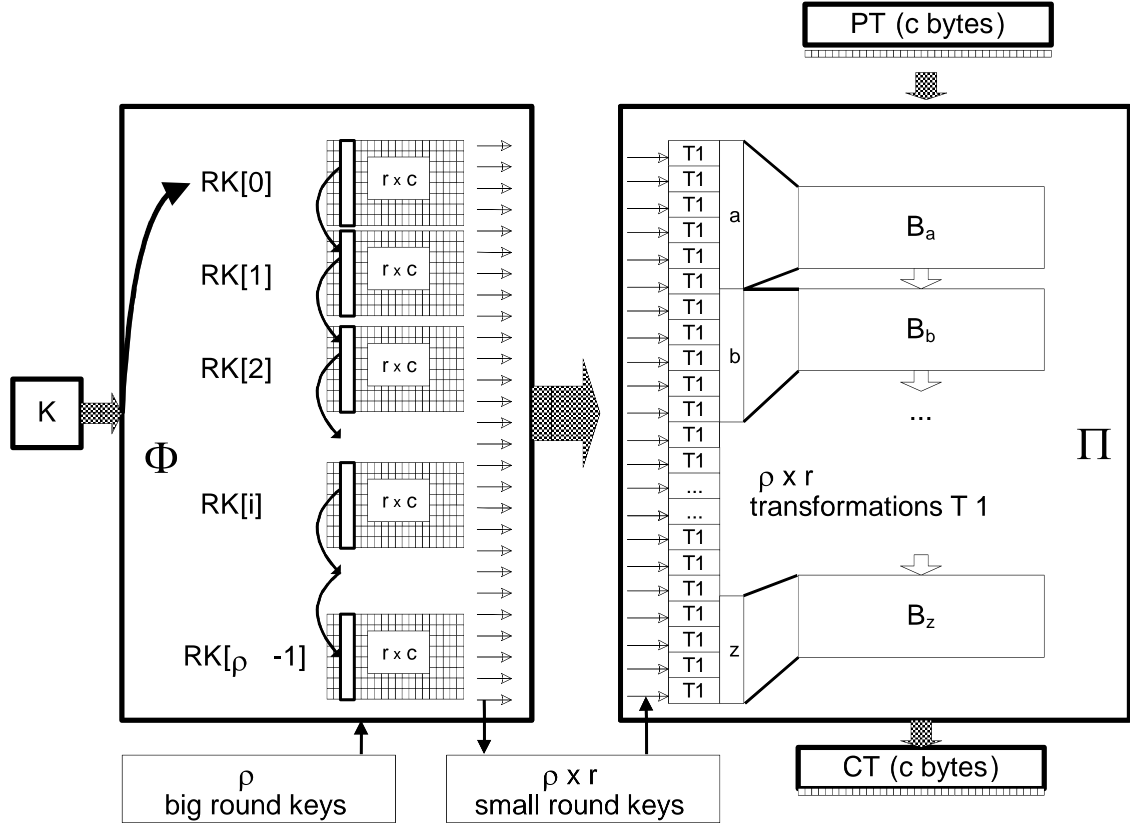

Fig. 1: DN functions family

2.1. DN(n, k)-ρ basic scheme

DN(n, k)-ρ is n-bit block cipher with k-bit encryption key K and ρ (big) rounds, where ρ** is a security parameter.

DN consists of two functions, the key expansion Φ and the product cipher Π, see Fig. 1. The basic idea behind the DN double net is that the keys $a, b, \ldots, z$ for the sub-ciphers of the product cipher $\Pi = B_z \bullet \ldots \bullet B_b \bullet B_a$ are generated by strong block cipher Φ. With increasing number of rounds, the keys (a, b,…) and (…, y, z) become computationally indistinguishable from independent random variables, since they are in plaintext-ciphertext relation for the block cipher Φ. Thus, the block ciphers (B_a, B_b , ...) and (... B_y, B_z) themselves become computationally indistinguishable from (independent) random block ciphers. As the function Φ is a strong block cipher only on columns of key array RK (see Fig. 1), reasonable efficiency is achieved. The function Π mixes the columns of array RK with each other and with the plaintext. More columns there are in round keys, more effective the whole process becomes. It is usual for the

** The variable ρ is denoted as rho in the source code

DN key to be several thousand bits long. This is an advantage when used in the hash function HDN, as the message enters into the function through the key.

Notations. The block length and the key length are rounded to bytes, the key length is a multiple of the block length and block length is a multiple of 32 bits. The scheme is described on byte level. The number of the bytes in the plaintext is denoted as $c = n/8$. It is the number of the columns in the array of the keys, as well. The number of the bytes in the key K is $k/8$. Key bytes are written into r (rows) by c (columns) array from left to right and from up to down, where $r = k/n$ ($rc = k/n \times n/8 = k/8$). The function Φ expands the encryption key into an array of round keys. It works with three-dimensional $\rho \times r \times c$ array of bytes RK[i][j][t], $i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$, $t = 0, ..., c - 1$ which is called round keys array. The first index (i) determines the big round key RK[i] as two-dimensional $r \times c$ array. The big round key RK[i] consists of r small round keys RK[i][j], $j = 0, ..., r - 1$. Small round key RK[i][j] is one row of the big round key and has c bytes RK[i][j][t], $t = 0, ..., c - 1$. The key K is the input to the function Φ. It is written into the first big round key RK[0] (left to right and up to down). From the first big round key, the function Φ progressively generates remaining $\rho - 1$ big round keys RK[i], $i = 1, ..., \rho - 1$.

The function Π mixes the plaintext with the array of round keys, see Fig. 1. Primarily, Π is the product of $\rho \times r$ elementary transformations T1, $\Pi = \Pi_{i = \rho - 1, ..., 0} \Pi_{j = r - 1, ..., 0} T1_{i,j}$, where each transformation $T1_{i,j}$ uses one small round key RK[i][j], $i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$. By grouping several (e.g. $r/2$ or $2r$) transformations T1 into one block cipher B, the function Π can be seen as the product of block ciphers B, where each one uses several small round keys, i.e. $\Pi = B_z \bullet ... \bullet B_b \bullet B_a$, where $z \,||\, ... \,||\, b \,||\, a$ = RK = RK[$\rho - 1$][$r - 1$] || RK[$\rho - 1$][$r - 2$] || ... || RK[0][1] || RK[0][0].

Transformation T1 consists of a substitution and a permutation on byte level, a linear transformation on bit level (not convertible to byte level) and small round key and round constant additions.

From the point of view of security proofs, we see the function Π as the product of the block ciphers B, from the point of view of the implementation in HW and SW we see it as $\rho \times r$ transformations T1.

2.2. Function Φ

The input to the function Φ is the encryption key K, its output is the round keys array RK. The function Φ consists of the **column transformation** and the **final key permutation**. The column transformation fills the array RK and the final key permutation permutes the bytes within this array. The column transformation is a set of independent column transformations F_t, $t = 0, ..., c - 1$, which work within the columns of array RK. Each column transformation is the product block cipher $F_t = f_{\rho-1,t} \bullet ... \bullet f_{2,t} \bullet f_{1,t}$ with r-byte block whose rounds are called partial column transformations ($f_{i,t}$). The column t of the array RK is progressively filled with the results of the partial rounds of the block cipher F_t. Each one of $(\rho - 1) \times c$ partial column transformations $f_{i,t}$, $i = 1,..., \rho - 1$, $t = 0, ..., c - 1$ is an elementary transformation (T2) that consists of byte level

substitution (r substitution boxes SubsF), bit level linear transformation (using MDS type r x r matrix) and r-byte round constant (RConstF) addition. Each column transformation F_t is thus a block cipher with constant key (round keys are constants), see Fig. 2.

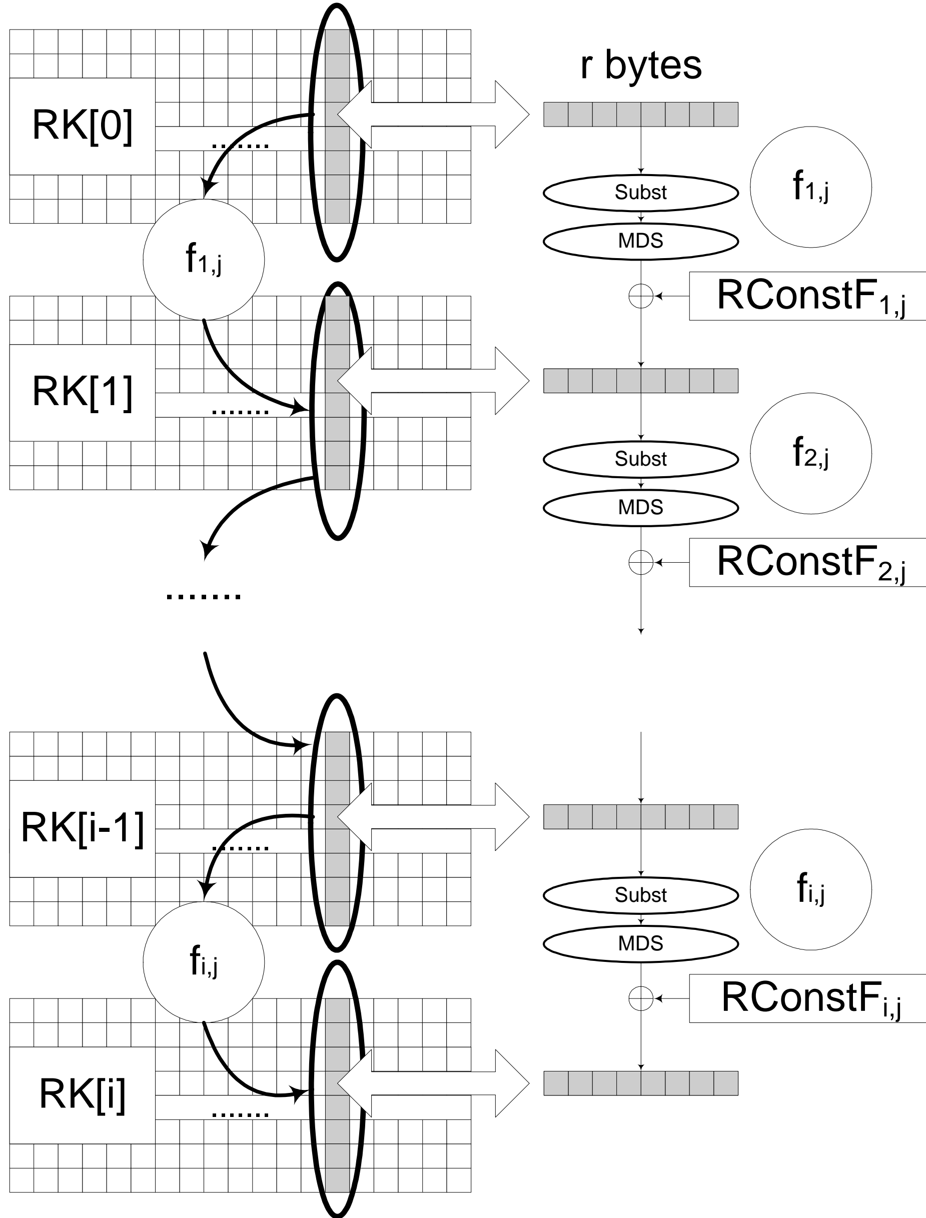

Fig.2: Column transformation

Mapping encryption key *K* to array RK

Key K is written into byte array RK[0] with dimensions r x c: RK[0][j][t] = K[$j*c + t$], j = 0, ..., r - 1, t = 0, ..., c - 1.

Array RK generation

We denote byte RK[i][j][t] as $RK_{i,j,t}$. Round keys RK[0], ..., RK[ρ - 1] are generated iteratively and independently column-wise (t = 0, ..., c - 1) using the function $F_t = f_{\rho-1,t} \bullet$

... • $f_{2,t}$ • $f_{1,t}$ in this way: RK[0] → RK[1] → ... → RK[ρ - 1]. Each function $f_{i,t}$ uses r (different in general) substitution boxes $SubsF_{i,j,t}$, $j = 0, ..., r - 1$, the matrix $MDS_{i,t}$ with dimensions r x r and r-byte round constant $RConstF_{i,t} = (RConstF_{i,0,t}, RConstF_{i,1,t}, ..., RConstF_{i,r-1,t})$. For $i = 1, ..., \rho - 1$ and $t = 0, ..., c - 1$ we have $(RK_{i,0,t}, RK_{i,1,t},..., RK_{i,r-1,t}) = f_{i,t}(RK_{i-1,0,t}, RK_{i-1,1,t}, ..., RK_{i-1,r-1,t}) = (MDS_{i,t} \bullet (SubsF_{i,0,t}(RK_{i-1,0,t}), SubsF_{i,1,t}(RK_{i-1,1,t}), ..., SubsF_{i,r-1,t}(RK_{i-1,r-1,t}))^T)^T \oplus (RConstF_{i,0,t}, RConstF_{i,1,t}, ..., RConstF_{i,r-1,t})$, where the operator ^{T} denotes transposition of a row into a column and vice versa. The matrix $MDS_{i,t}$ is MDS (maximum distance separable) type matrix and multiplications are computed in finite field $GF(2^8)$.

Final key permutation KeyPerm

The final key permutation is a permutation on set INDX = {0, 1, ..., ρ - 1} x {0, 1, ..., r - 1} x {0, 1, ..., c - 1}, KeyPerm: INDX → INDX: (i, j, t) → KeyPerm(i, j, t). It permutes the bytes of the array RK, i.e. $RK_{i,j,t} = RK_{KeyPerm(i,j,t)}$, $i = 0, ..., \rho - 1, j = 0, ..., r - 1, t = 0, ..., c - 1$. It is applied after the creation of the array RK by the column transformation. From the security point of view, this permutation is not necessary. Its purpose is to make the diffusion of the round keys columns within the function Π more efficient. The permutation can be very simple, for example the cyclic shift of bytes within small round key. More details are to follow.

2.3. Function Π

The function Π is a block cipher. The plaintext consists of c bytes indata(0), ..., indata(c - 1), the ciphertext consists of c bytes outdata(0), ..., outdata(c - 1). The encryption key is array RK, comprising ρ x r small round keys RK[i][j], $i = 0, 1, ..., \rho - 1, j = 0, 1, ..., r - 1$. Primarily, Π is a product of ρ x r elementary transformations T1, $\Pi = \Pi_{i=\rho-1,...,0} \Pi_{j=r-1,...,0} T1_{i,j}$, where $T1_{i,j}$ uses small round key RK[i][j], $i = 0, ..., \rho - 1, j = 0, ..., r - 1$. The output from one transformation T1 is the input to another transformation T1. Input to the function Π is the input to the first transformation T1, the output from the last transformation T1 is the output from the function Π.

2.3.1. Transformations $T1_{i,j}$

Each transformation $T1_{i,j}$, $i = 0, ..., \rho - 1, j = 0, ..., r - 1$, consists of a substitution and a permutation on byte level, a linear transformation on bit level (not convertible to byte level) and small round key and round constant additions. All these variables can be different for different transformations $T1_{i,j}$. For each pair (i, j), $i = 0, ..., \rho - 1, j = 0, ..., r - 1$, we have:

- c substitution boxes $SubsB_{i,j,t}$, $t = 0, ..., c - 1$, mapping a byte on a byte
- permutation on the set {0, 1, ..., c - 1}, called type "Small-Middle-Large" permutation and denoted as $SMLPerm_{i,j}$: {0, 1, ..., c - 1}→{0, 1, ..., c - 1}: t → $SMLPerm_{i,j}(t)$,
- linear transformation consisting of $n/32 = c/4$ matrices $MDS_{i,j,v}$ with dimensions 4 by 4 bytes, $v = 0, ..., c/4 - 1$,
- round constant $RConstB_{i,j}$ with c bytes $(RConstB_{i,j,0}, ..., RConstB_{i,j,c-1})$,
- small round key RK[i][j] with c bytes $(RK_{i,j,0}, ..., RK_{i,j,c-1})$.

Remark. Linear transformation in T1. The linear transformation used in T1 can be more general, in the DN construction the linear layer representation by (small) 4 x 4 matrices is employed. The matrix multiplication is computed in the field $GF(2^8)$. Using these matrices in this way requests the plaintext length to be a multiple of 32 bits. If different dimension matrices are used as the building block, the plaintext length does not necessarily need to be such a multiple. This completes the description of DN.

2.4. Block cipher DN class parameters

DN scheme is a general scheme based on two SP networks Φ and Π. One SP network expands the encryption key to the array of round keys, while the second one mixes the round keys with the plaintext. Contrary to the classical block ciphers, the key is processed with the same attention as the plaintext.

DN's parameters are its building blocks, their types, dimensions and contents. These parameters are available for DN(n, k)-ρ :

Main dimensions:

- ***n***, plaintext length in bits ($c = n/8$),
- ***k***, key K length in bits ($r = k/n$),
- ρ, number of big rounds,

Function Φ:

- S-boxes $\mathbf{SubsF}_{i,j,t}$ mapping a byte on a byte, $i = 1, ..., \rho - 1, j = 0, ..., r - 1, t = 0, ..., c - 1$,
- matrices $\mathbf{MDS}_{i,t}$ with dimension r x r , $i = 1, ..., \rho - 1, t = 0, ..., c - 1$,
- constants $\mathbf{RConstF}_{i,t}$ with r bytes, $i = 1, ..., \rho - 1, t = 0, ..., c - 1$,
- final key permutation **KeyPerm** on the set $\{0, ..., \rho - 1\}$ x $\{0, ..., r - 1\}$ x $\{0, ..., c - 1\}$,

Function Π:

- permutations $\mathbf{SMLPerm}_{i,j}$ on the set $\{0, ..., c - 1\}$,
- S-boxes $\mathbf{SubsB}_{i,j,t}$ mapping a byte on a byte, $i = 1, ..., \rho - 1, j = 0, ..., r - 1, t = 0, ..., c - 1$,
- matrices $\mathbf{MDS}_{i,j,v}$ with dimension w x w, $i = 0, ..., \rho - 1, j = 0, ..., r - 1, v = 0, ..., c/w - 1$, where w divides c (individual dimensions of matrices can vary, the case $w = 4$ will mostly be used, see next chapter for details),
- constants $\mathbf{RConstB}_{i,j}$ of c bytes, $i = 0, ..., \rho - 1, j = 0, ..., r - 1$.

Remark. The construction leaves a lot of freedom for choices for these parameters. However, there are some rules that the building blocks have to respect, briefly:

- function Π is a strong block cipher,
- all column transformations of Φ are strong block ciphers (with a fixed key), they are pair wise different if possible,
- functions Φ and Π do not share any S-box,
- all of the S-boxes have good linear and differential characteristics and they are generated non-algebraically, (pseudo)randomly if possible,
- matrices used by the functions Φ and Π are all MDS type matrices (maximum distance separable).

Next chapter explains these rules in detail.

3. Network Π construction

3.1. Π as product of block ciphers B

The function Π is the product of the block ciphers B, each employing several T1 rounds (several small round keys), i.e. $\Pi = B_z \bullet B_y \bullet ... \bullet B_b \bullet B_a$. The goal for the function Π is to be a strong block cipher, resistant to linear and differential cryptanalysis. It is possible to construct identical block ciphers B_z, ..., B_a, or eventually $B_y = ... = B_a$ (= B) while B_z would consist of “remaining number of small rounds”. B is designed to be as resistant against linear and differential cryptanalysis, as possible. To do so, we use the proofs from Appendix A about SP resistance against linear and differential cryptanalysis. We construct the function B as a nested SP network. Several papers ([Ho00], [Ka01], [Chu03], and [Sa03]) focused on nested networks, however, here it is sufficient to use the results from [Ho00]. Theorems 1 and 2 provide us with the probability bounds on the maximal differential (DP^B) and linear hull (LP^B) of the block cipher B. Block cipher B is one round of the product cipher Π, thus the estimates (DP^B) and (LP^B) somehow correspond to the quality of the function $\Pi = B_z \bullet B_y \bullet ... \bullet B_b \bullet B_a$. The estimates DP^Π and LP^Π cannot be directly deduced from DP^B x DP^B x ... x DP^B x DP^B neither LP^B x LP^B x ... x LP^B x LP^B, even if they were in the past. However, it suffices for DP^B and LP^B to be small. Let us remark, according to [NK92] the value DP^B x DP^B can be used to estimate DP^Π for $\Pi = B \bullet B \bullet B \bullet B$. The value DP^Π is probably lower than this (currently best known) estimate, however, there are no methods known so far to prove this. On the other hand, it is expectable the estimates will get better.

3.2. Network Π S-boxes

Firstly, note S-boxes can be pair wise different in network Π. Denote p_B (q_B) as the maximum value of the maximal differential probability (maximal linear probability, respectively) taken over all S-boxes used in the function B. Smaller the values of p_B and q_B are, more resistant against linear and differential cryptanalysis the function B becomes.

3.3. Network Π example for n = 512

Block size $n = 512$ bits, i.e. $c = 64$ bytes. We use the decomposition $c = 64 = c_1$ x c_2 x c_3 = 4 x 4 x 4 to construct network Π. Block cipher B is constructed as a 3-level nested SPN network.

XS-box is constructed as SDS network of S-boxes with the width $c_1 = 4$,
XXS-box is constructed as SDS network of XS-boxes with the width $c_2 = 4$,
XXXS-box is constructed as SDS network of XXS-boxes with the width $c_3 = 4$.

XXXS-box is the block cipher B, as well. It consists of 8 elementary transformations T1.

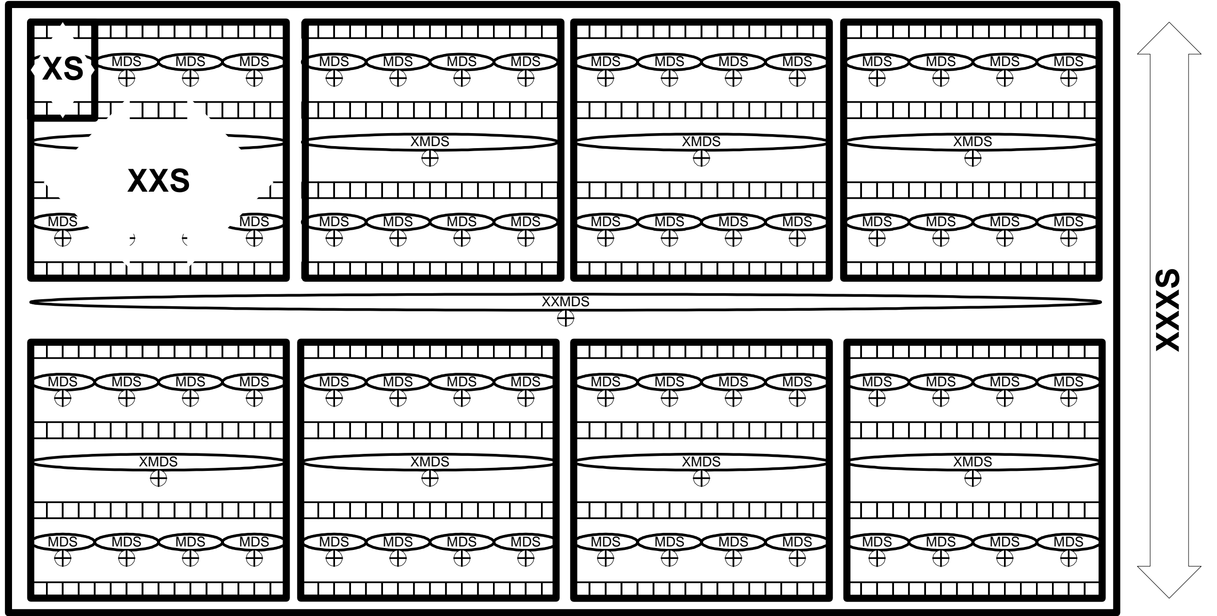

Fig.3: Block cipher B as an XXXS-box

Remark. All S-boxes, XS-boxes and XXS-boxes can be pair wise different.

Let's assume the diffusion levels within all XS, XXS and XXXS boxes are maximal. Then the following holds by Theorem 1 (see Appendix A) applied on SDS networks XS, XXS and XXXS:

$DP^{XS} \leq (p_B)^4$,
$DP^{XXS} \leq (DP^{XS})^4 \leq (p_B)^{4x4}$,
$DP^{XXXS} \leq (DP^{XXS})^4 \leq (p_B)^{4x4x4}$,
thus
$DP^B = DP^{XXXS} \leq (p_B)^{64}$ and analogically using Theorem 2 (see Appendix A) it holds $LP^B = LP^{XXXS} \leq (q_B)^{64}$.

Block cipher B resistance against DC and LC is now ensured for small and suitable p_B and q_B.

3.4. B as an N-level nested SPN

The general network Π construction is based on the size c of the plaintext block in bytes. Most of the times, c is a power of 2, with 8, 16, 32 and 64 being the most important values. During the construction of B as an N-level nested SPN, we use the decomposition $c = c_1$ x c_2 x c_3 x ... x c_N, where c_1 is the first XS network width, c_2 the second XXS (X^2S) network width, ..., c_N the last XX...XS (X^NS) network width.

X^1S-box is constructed as SDS network of S-boxes with the width c_1,
X^2S-box is constructed as SDS network of XS-boxes with the width c_2,
...
X^NS-box is constructed as SDS network of X^{N-1}S-boxes with the width c_N.

If the number of small rounds in Π is not a multiple of the number of rounds in B, we denote the remaining part of block cipher B as B_z, i.e. $\Pi = B_z \bullet B \bullet ... \bullet B \bullet B$.
The diffusion levels within all XS, …, $X^N S$ boxes are assumed to be maximal.

3.5. Network Π resistance against DC and LC

As we mentioned in the beginning of this chapter already, there is no other option (because of the lack of proving methods) than to measure network Π resistance against DC and LC with the values DP^B and LP^B. As B can be seen as big box B: $\{0, 1\}^n \to \{0, 1\}^n$, $n = 8c$, its maximum differential and maximum linear probabilities (see Appendix A) are defined as $\mathbf{DP^B} = \max DP^B(\Delta x \to \Delta y)$, where the maximum is taken over all $\Delta x \neq 0$, $\Delta x \in \{0, 1\}^n$, $\Delta y \in \{0, 1\}^n$, and $\mathbf{LP^B} = \max LP^B(\Gamma x \to \Gamma y)$, where the maximum is taken over all $\Gamma x, \Gamma y \neq 0$, $\Gamma x \in \{0, 1\}^n$, $\Gamma y \in \{0, 1\}^n$.

Theorem 3. Block cipher B resistance against DC and LC.
If B is constructed as nested SP network (as in Appendix A), the following holds
$DP^B \leq (p_B)^c$,
$LP^B \leq (q_B)^c$.
Proof. Follows from Theorem 1, if it is applied inductively on the X^1S, ..., X^NS boxes construction. We have $DP^B = DP^{X^N S} \leq (DP^{X^{N-1}S})^{c_N} \leq (DP^{X^{N-2}S})^{c_{N-1} \times c_N} \leq ... \leq (DP^S)^{c_1 \times ... \times c_{N-1} \times c_N} = (DP^S)^c = (p_B)^c$. Analogically, $LP^B \leq (q_B)^c$ holds using Theorem 2.

Corollary. Best currently known network Π resistance estimate against DC. As already mentioned, the best currently known estimate for DP^Π is $DP^B \times DP^B \leq (p_B)^c \times (p_B)^c = (p_B)^{2c}$, if Π consists of at least four blocks B. However, in reality the estimate is definitely smaller.

Corollary. Best currently known network Π resistance estimate against LC. Estimating the network Π resistance against LC, we can only use the estimate $LP^B \leq (q_B)^c$ for one round of the product cipher $\Pi = B_z \bullet B \bullet ... \bullet B \bullet B$.

Remark. Variable construction for the same *c*. Even for the same value of *c*, the network construction has several possibilities. This depends on the factorization of *c* and on the possibility to have different diffusion layers for different boxes.

Remark. Number of rounds in B. The number of rounds in block cipher B depends on the fact that every higher-level SDS network consists of two lower level SDS networks. Thus, the number of rounds (i.e. number of substitution layers) is the double of the amount of the factors of *c*, which is $2N$.

Finally, S-boxes $SubsB_{i,j,t}$ mapping a byte on a byte ($i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$, $t = 0, ..., c - 1$) can be chosen different or all identical. Random or pseudo-random S-boxes make the ideal choice, if sufficient resistance against linear and differential cryptanalysis is ensured. Both, the network Π resistance against DC and LC and the number of the block ciphers in the product $\Pi = B_z \bullet B \bullet ... \bullet B \bullet B$, depend on the

values p_B (q_B). The S-boxes used in networks Π and Φ should not be the same ones. They should not have an algebraic structure (as AES S-boxes does), even if there is no direct proof for this property.

3.6. Network Π diffusion layer maximality

Big MDS matrices with dimension C x C, where $C = c_1$ x ... x c_{i-1} x c_i can be used to ensure diffusion layer maximality in X^iS-boxes. For instance, value c for network Π from the previous example would equal $c = 64 = c_1$ x c_2 x c_3 = 4 x 4 x 4 and the matrix X^3MDS dimension would be 64 x 64 bytes. However, the implementation of such matrices is time and memory consuming. Different approaches can be used to ensure the maximality. DN function class does not prescribe these; however, we now show one such possible approach.

Instead of using one C x C MDS matrix, where $C = c_1$ x ... x c_{i-1} x c_i, we construct c_1 x ... x c_{i-1} MDS matrices with dimension c_i x c_i. In case c is a power of 2, the decomposition is done in the way that all factors are equal to 4, except possibly the first one whose value can be 2, 4 or 8. Thus, the dimensions of the matrices used can be 2 x 2, 4 x 4 and 8 x 8. Each one of the two layers in X^iS-box contains c_i X^{i-1}S-boxes. The matrix X^{i-1}MDS joins the first layer of c_i X^{i-1}S-boxes with the second layer of c_i X^{i-1}S-boxes. The width of each X^{i-1}S-box is c_1 x ... x c_{i-1} bytes.

We could construct the matrix X^{i-1}MDS as the (c_1 x ... x c_{i-1} x c_i) x (c_1 x ... x c_{i-1} x c_i) type matrix. We construct it as a set of c_1 x ... x c_{i-1} MDS matrices with dimensions c_i x c_i, instead. Each one of these small c_i x c_i matrices chooses (randomly) one single byte from each of c_i input X^{i-1}S-boxes. Thus, the input of this matrix is c_i bytes long. The output bytes are transferred (one by one) to all c_i output X^{i-1}S-boxes. The system of these matrices creates maximal diffusion layer. (As we shall see later on, the choice of the byte positions in the input X^{i-1}S-boxes that forms the MDS matrices defines accordingly permutations SMLPerm.)

Theorem 4. Diffusion layer maximality. The matrix X^{i-1}MDS constructed as a system of c_1 x ... x c_{i-1} MDS matrices with dimensions c_i x c_i is the maximal diffusion layer in X^iS-box.

Proof. Let's assume an input difference in k X^{i-1}S-boxes, $1 \leq k \leq c_i$. Let us note the input difference in X^{i-1}S-box means there is a change of at least one of the input bytes. Let's focus on the first changed input byte in the first changed X^{i-1}S-box. This byte is the input to the one of the c_1 x ... x c_{i-1} MDS matrices with dimensions c_i x c_i of the relevant diffusion layer. Denote this matrix as M and the total number of changed bytes on its input as s. It holds $1 \leq s \leq k \leq c_i$. Since M is an MDS matrix with dimensions c_i x c_i, there are at least v $c_i + 1 - s$ bytes changed on its output. We have $c_i + 1 - s \geq c_i + 1 - k$. As all the output bytes of matrix M serve as input bytes to different X^{i-1}S-boxes, there is at least $c_i + 1 - k$ changed X^{i-1}S-boxes on the output. The maximality of X^{i-1}MDS diffusion layer is now verified.

Final remark. Matrices $\mathbf{MDS}_{i,j,v}$. Matrices $MDS_{i,j,v}$ can be of different dimensions (w x w, where w is a divisor of c) and have different contents. Various matrices with

various dimensions can be used in various places of network Π, even within one diffusion layer. Two requirements have to be fulfilled:

- the diffusion maximality for all the layers,
- a bit level (not a byte level as a whole) diffusion should be ensured for all MDS matrices employed.

Particularly, this is not satisfied for the matrices consisting only of elements 0x00 and 0x01 (in hexadecimal notation). The matrices should contain as few of these elements as possible. Expressing MDS matrix in binary with dimensions $8r$ x $8r$, it should not be sparse and should not contain any obvious pattern. As a binary matrix, it should be as random as possible. Thus, ideally all the matrices $MDS_{i,j,v}$ are pair wise different and generated randomly. This is a countermeasure against the algebraic attacks. It is not strictly forbidden for all the matrices to be identical, however.

3.7. Small-Middle-Large type permutation

This section describes the construction of SML-type (Small-Middle-Large) permutations and introduces the term conjugate bytes.

3.7.1. SMLPerm and T1

If the biggest possible matrix with dimensions (c_1 x ... x c_{i-1} x c_i) x (c_1 x ... x c_{i-1} x c_i) is employed to ensure the diffusion layer X^{i-1}MDS maximality (Small - among S-boxes, Middle - among XS boxes, Large - among X^{i-1}S-boxes), the corresponding permutation SMLPerm sets the input bytes selection order for this matrix. Different permutations can be defined for different diffusion layers. A permutation can be incorporated directly into a matrix. We can thus define one matrix and different permutations for different diffusion layers, or different matrices (original matrix with permuted columns) and identical permutations.

If c_1 x ... x c_{i-1} (identical) MDS matrices with dimensions c_i x c_i are employed to ensure the diffusion layer X^{i-1}MDS maximality, the outputs from c_i X^{i-1}S-boxes can be used as inputs to these MDS matrices in variously permutated order.

The diffusion layer maximality can be achieved using matrices with other dimensions, as well, see Fig. 4.

For a given full width diffusion layer in network Π, the bytes selection to its matrices corresponds to a permutation of c_1 x ... x c_{N-1} x c_N bytes. We call this permutation SMLPerm associated to the given diffusion layer. Simultaneously, this permutation is used in the corresponding transformation T1. (Later on, we shall see the output boxes bytes selection is in fact the inverse of the input positions selection in the permutation SMLPerm during the next transformation T1.)

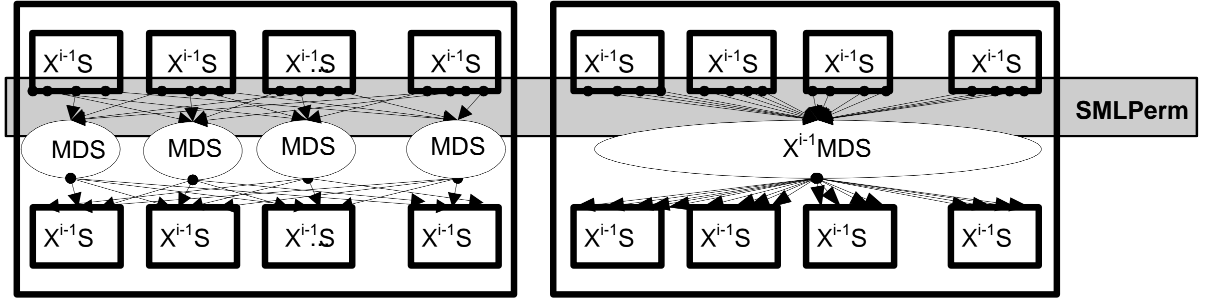

Fig.4: Permutation SMLPerm

3.7.2. SMLPerm and diversity

If the diffusion layer X^{i-1}MDS is constructed as a system of c_1 x ... x c_{i-1} MDS matrices with dimensions c_i x c_i, corresponding permutations SMLPerm$_{i,j}$ allow us to enhance the diffusion and the diversity (non-symmetry) inside the block cipher B. Each one of the small MDS matrices with dimensions c_i x c_i can freely choose exactly one byte coming from each of c_i input X^{i-1}S-boxes and it can output it to any position to each of the output X^{i-1}S-boxes. This ensures each input X^{i-1}S-box influences all output X^{i-1}S-boxes. As we will see later on, such property doesn't need to be satisfied one layer lower – the case of X^{i-2}S-boxes.

Each big input X^{i-1}S-box consists of c_{i-1} small X^{i-2}S-boxes. Let's look at the first of these small input X^{i-2}S-boxes for instance and see how many small output X^{i-2}S-boxes are influenced by it (see Fig. 5 and 6). The box has c_1 x ... x c_{i-2} bytes which influence c_1 x ... x c_{i-2} x c_i output bytes employing c_i MDS matrices. The maximality property ensures each one of c_i output X^{i-1}S-box receives exactly c_1 x ... x c_{i-2} output bytes. The position of these bytes within X^{i-1}S-box is random; they can reach all of the small X^{i-2}S-boxes; however, in the worst case they can all be placed into a single one small X^{i-2}S-box (it is exactly c_1 x ... x c_{i-2} bytes long). These small output X^{i-2}S-boxes that are influenced are called **conjugate output boxes** (with the given small input X^{i-2}S-box). The other bytes of MDS matrices that process the given small input box, acquire their input from several other small input boxes. We reference to these small input boxes as to **conjugate input boxes** (with the given small input box). In the worst case, there can be just a single small X^{i-2}S-box within each big input X^{i-1}S-box that is conjugate to a given small box (the same is true for output boxes). To reach such situation, all of j-th bytes from the big input boxes are directed to j-th MDS matrix, where j = 0, ..., c_1 x ... x c_{i-2} x c_{i-1} - 1 while the matrix output bytes are directed j-th positions of the big output boxes (see Fig. 5). Such a constructions ensures the only small conjugate boxes are k-th boxes within big input and output box (k = [j/(c_1 x ... x c_{i-2})], k = 0, ..., c_{i-1} - 1). This simple example shows the systematic selection of permutations SMLPerm might not be the best one from the diffusion point of view. We show a suitable permutation selection provides faster diffusion and allows avoiding intentional structural regularities. Two different permutation selections are pictured on two following figures. The figures illustrate small input and small output boxes conjugate to the first small input box in the first big input box.

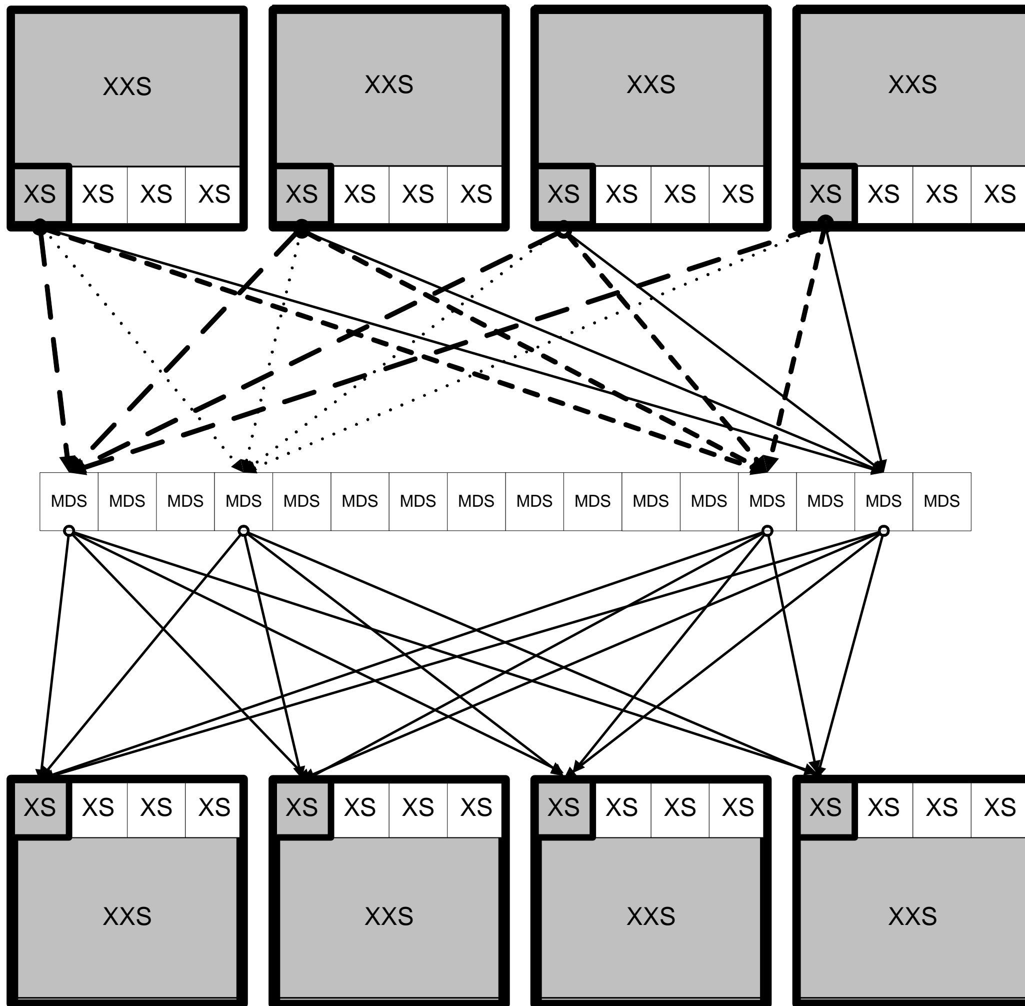

Fig.5: Systematic permutation selection

Permutations $SMLPerm_{i,j}$ are selected systematically on Fig. 5. The first bytes of the first small boxes are transferred via MDS matrices to the first bytes of the first small output boxes within the big boxes. The second, third and fourth bytes within small boxes are handled similarly. The set of the first small input boxes (within all big input boxes) influences only the set of the first small output boxes (within all big output boxes) in this diffusion layer. The sizes of conjugate input boxes set and conjugate output boxes set are minimal – only 4 bytes. However, if the permutation is selected carefully, the size of conjugate input boxes set is 13 (maximal possible) and 16 for the conjugate output boxes set (maximal possible), see the example on Fig. 6.

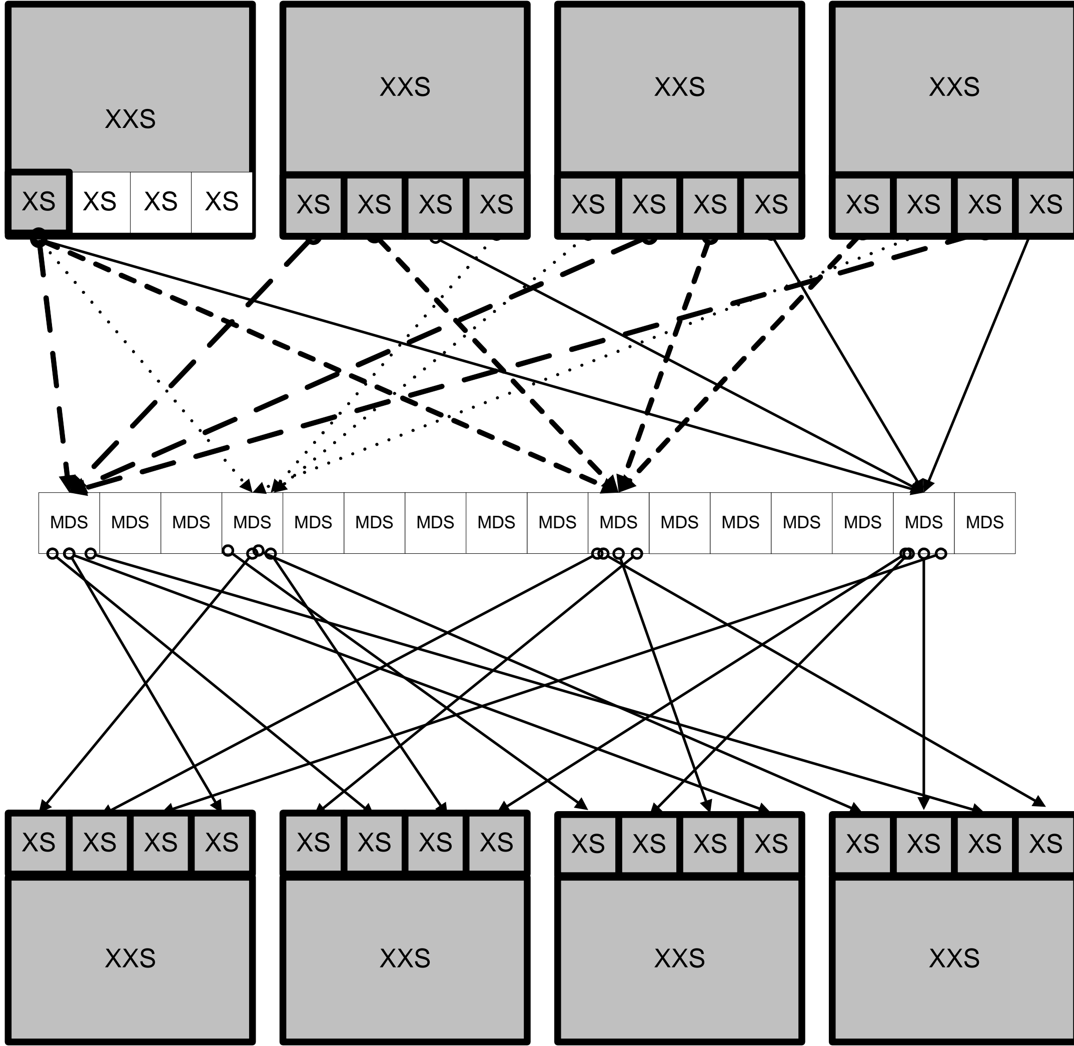

Fig.6: Random permutation selection, conjugate boxes

To conclude, Small-Middle-Large type permutations $SMLPerm_{i,j}$ on the set {0, ..., c - 1} can be selected randomly; however, the maximality of the corresponding diffusion layer has to be ensured. The random selection or sufficiently de-regularized permutations seems to be the right choice, if enough conjugates boxes are ensured.

3.8. Constants $RConstB_{i,j}$

The purpose of c-byte constants $RConstB_{i,j}$, $i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$ is to differentiate the individual transformations T1. The constants can be incorporated in the S-boxes definition, as they only translate these by another constant (see the proof later on). In case only one single S-box is used in the function Π (useful in certain HW implementations), the round constants define up to 256 of its translations. In such case, the ideal choice are random c-byte constants $RConstB_{i,j}$, $i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$. However, if all S-boxes are selected randomly, the constants can be selected as all zeroes.

4. Network Φ construction

4.1. S-boxes $SubsF_{i,j,t}$

DN functions family requests the S-boxes used in the function Φ to be different from the ones used in the function Π. Ideally, their difference is random. This is a countermeasure against the algebraic attacks aiming to have different S-boxes in the equations characterizing the functions Φ and Π. Algebraic properties (as the ones of AES's S-box) should not be present in any of S-boxes employed. Possible oversimplified expressions for the relations in the functions Φ and Π are prevented by this countermeasure. It is not strictly forbidden to use a single S-box in the function Φ, however, one ideally chooses randomly generated S-boxes with satisfying resistance against linear and differential cryptanalysis. Let's denote p_Φ (q_Φ) as the maximum value DP^S (LP^S) over all S-boxes $SubsF_{i,j,t}$ ($i = 1, ..., \rho - 1, t = 0, ..., c - 1, j = 0, ..., r - 1$) used in the function Φ. Network Φ resistance against DC and LC and the number of the big round depend on the values p_Φ and q_Φ. Smaller these values are, less big rounds ρ DN may have (more details later on). Random or pseudo-random S-boxes with sufficient resistance against linear and differential cryptanalysis are the ideal choice.

4.2. Φ as system of block ciphers F_t

Variable elements in the function Φ are S-boxes $\mathbf{SubsF}_{i,j,t}$ ($i = 1, ..., \rho - 1, j = 0, ..., r - 1, t = 0, ..., c - 1$), matrices $\mathbf{MDS}_{i,t}$ ($i = 1, ..., \rho - 1, t = 0, ..., c - 1$) and the constants $\mathbf{RConstF}_{i,t}$ ($i = 1, ..., \rho - 1, t = 0, ..., c - 1$). The rationale of these elements is to differentiate the column transformations F_t ($t = 0, ..., c - 1$) as much as possible (ideally randomly) and to make these as resistant to linear and differential cryptanalysis as possible. A random selection of these elements would mean huge memory requirements, however. Thus, the minimal requirement is for all transformations F_t ($t = 0, ..., c - 1$) to be pair wise different and resistant against linear and differential cryptanalysis.

4.3. Transformation F_t resistance against DC and LC

Each one of the column transformation F_t, $t = 0, ..., c - 1$, is a product block cipher with r-byte block length. Alternatively, it can be described as the product of $\rho/2$ SDS networks joined by the diffusion layers, see Fig. 7.

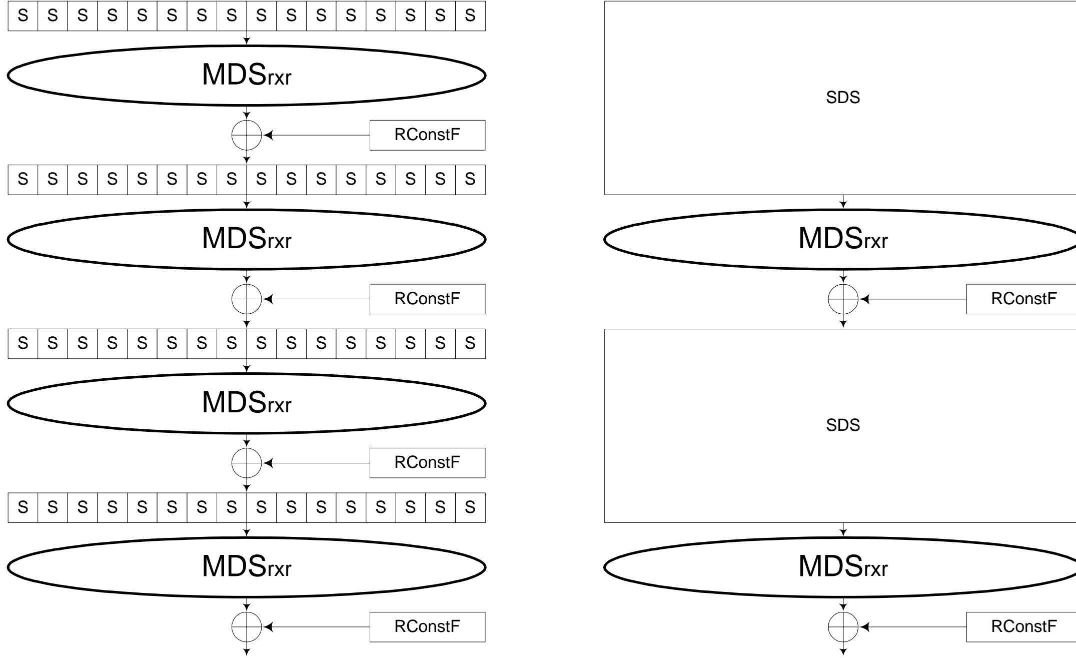

Fig. 7: Column transformation pictured as a product cipher and its SDS network round

As an SDS network can be seen as one (big) round of the block cipher F_t, its maximum differential probability DP^{SDS} and maximum linear probability LP^{SDS} can be estimated using the Theorem 1 and Theorem 2 (see Appendix A).

Theorem 5. Block cipher F_t, t = 0, ..., c - 1, resistance against DC and LC.
Joining of two consecutive rounds of block cipher $F_t = f_{\rho-1,t} \bullet ... \bullet f_{2,t} \bullet f_{1,t}$ creates an SDS network with
$DP^{SDS} \leq (p_\Phi)^r$,
$LP^{SDS} \leq (q_\Phi)^r$.
Proof. Follows directly from Theorem 1 and Theorem 2 (Appendix A).

The lack of proving methods leaves us with no other option than to use values DP^{SDS} and LP^{SDS} to measure F_t resistance against DC and LC.

Corollary 1. The best currently known estimate for F_t resistance against DC. Let us note the estimate $DP^{Ft} \leq (DP^{SDS})^2 \leq (p_\Phi)^{2r}$ can be used for F_t = SDS • ... • SDS • SDS, if at least 4 SDS networks, i.e. **8 substitution layers (8 big rounds)** are employed (see [NK92]). The value DP^{Ft} is probably lower than this (currently best known) estimate $DP^{SDS} \times DP^{SDS} \leq (p_\Phi)^{2r}$.

Corollary 2. The best currently known estimate for F_t **resistance against LC.** The only usable fact the estimate F_t resistance against LC is its "one round" estimate $LP^{SDS} \leq (q_\Phi)^r$ from the product cipher F_t = SDS • ... • SDS • SDS.

Remark to transformations F_t resistance against DC a LC. For all the transformations F_t (t = 0, ..., c - 1) it is requested to be as resistant against linear and differential cryptanalysis as possible. As the block cipher F_t has constant round key, the

classical linear and differential cryptanalysis are not applicable. In fact, the countermeasures against the linear and differential cryptanalysis are present in order to avoid possible exploitable linear or differential relations between the inputs and outputs of the function F_t (or its rounds). The values p_Φ and q_Φ and the S-boxes selection have the greatest influence on these properties.

4.4. Matrices $MDS_{i,t}$ and diffusion layer F_t maximality

During the construction of DN functions family, it has to be ensured the matrices $MDS_{i,t}$ ($i = 1, ..., \rho - 1$, $t = 0, ..., c - 1$) in the transformations F_t ($t = 0, ..., c - 1$) are MDS matrices. Moreover, they should ensure not only byte level, but also bit level diffusion. The matrices consisting only of elements 0x00 and 0x01 (hexadecimal) do not satisfy this requirement. The use of these elements should be very rare. The binary expression of the matrix should not be sparse, neither with an obvious pattern. Ideally, matrices $MDS_{i,t}$ are pair wise different and generated randomly. This is a countermeasure against the algebraic attacks. It is not strictly forbidden for all the matrices to be identical, however.

4.5. Constants $RConstF_{i,t}$

The purpose of the constants $RConstF_{i,t}$ ($i = 1, ..., \rho - 1$, $t = 0, ..., c - 1$) in the DN definition is only formal. As they translate the S-boxes by a constant, they can be incorporated in these boxes (see remark below). In case only one single S-box is used in the function Π (useful in certain HW implementations), the round constants define up to 256 of its translations. In such case, the ideal choice are random constants $RConstB_{i,j}$, $i = 0, ..., \rho - 1$, $j = 0, ..., r - 1$. However, if all S-boxes are selected randomly, the constants can be selected as all zeroes, i.e. removed.

Remark. Incorporating the round constants into S-boxes. The round constant addition can be trivially transformed to a constant S-box translation. Let's denote the translated S-box as $SubsF_{i,j,t}^*(x) = SubsF_{i,j,t}(x) \oplus a_{i,j,t}$. We compute the translation as $(a_{i,0,t}, a_{i,1,t}, ..., a_{i,r-1,t})^T = MDS_{i,t}^{-1} \bullet (RConstF_{i,0,t}, RConstF_{i,1,t}, ..., RConstF_{i,r-1,t}))^T$, thus $MDS_{i,t} \bullet (SubsF_{i,0,t}^*(RK_{i-1,0,t}), SubsF_{i,1,t}^*(RK_{i-1,1,t}), ..., SubsF_{i,r-1,t}^*(RK_{i-1,r-1,t}))^T \oplus (0, 0, ..., 0)^T = MDS_{i,t} \bullet (SubsF_{i,0,t}(RK_{i-1,0,t}) \oplus a_{i,0,t}, SubsF_{i,1,t}(RK_{i-1,1,t}) \oplus a_{i,1,t}, ..., SubsF_{i,r-1,t}(RK_{i-1,r-1,t}) \oplus a_{i,r-1,t})^T = MDS_{i,t} \bullet (SubsF_{i,0,t}(RK_{i-1,0,t}), SubsF_{i,1,t}(RK_{i-1,1,t}), ..., SubsF_{i,r-1,t}(RK_{i-1,r-1,t}))^T \oplus MDS_{i,t} \bullet (a_{i,0,t}, a_{i,1,t}, ..., a_{i,r-1,t})^T = MDS_{i,t} \bullet (SubsF_{i,0,t}(RK_{i-1,0,t}), SubsF_{i,1,t}(RK_{i-1,1,t}), ..., SubsF_{i,r-1,t}(RK_{i-1,r-1,t}))^T \oplus (RConstF_{i,0,t}, RConstF_{i,1,t}, ..., RConstF_{i,r-1,t})^T = (RK_{i,0,t}, RK_{i,1,t}, ..., RK_{i,r-1,t})^T$, q.e.d.

4.6. Final key permutation KeyPerm

The final key permutation can be freely selected in the function Φ. From the security point of view it is not indispensable, its objective is to make the round key diffusion more efficient in the function Π. As the differences in the array RK are propagated mainly within the columns, the goal of KeyPerm is to spread the differences in one column of round key array into as many boxes in the function Π as possible. KeyPerm can be a very simple permutation, for example a permutation that cyclically shifts

selected rows in array RK: RK[i][j][k] = RK[i][j][(k + shift_row_j) mod c], see Fig. 8. As shown on Fig. 9, a specific definition of KeyPerm depends on the specific structure of the function Π. As can be seen on the Fig. 9, the use of KeyPerm has little sense with the round keys processing biggest matrices XXXMDS. The matrix itself ensures the mixing among the biggest boxes in this case.

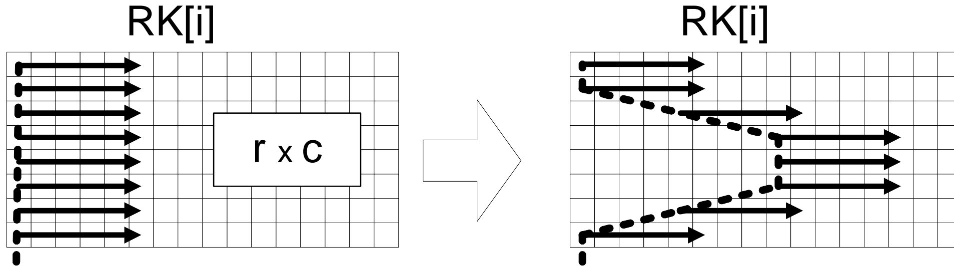

Fig. 8: An example of KeyPerm (r = 8)

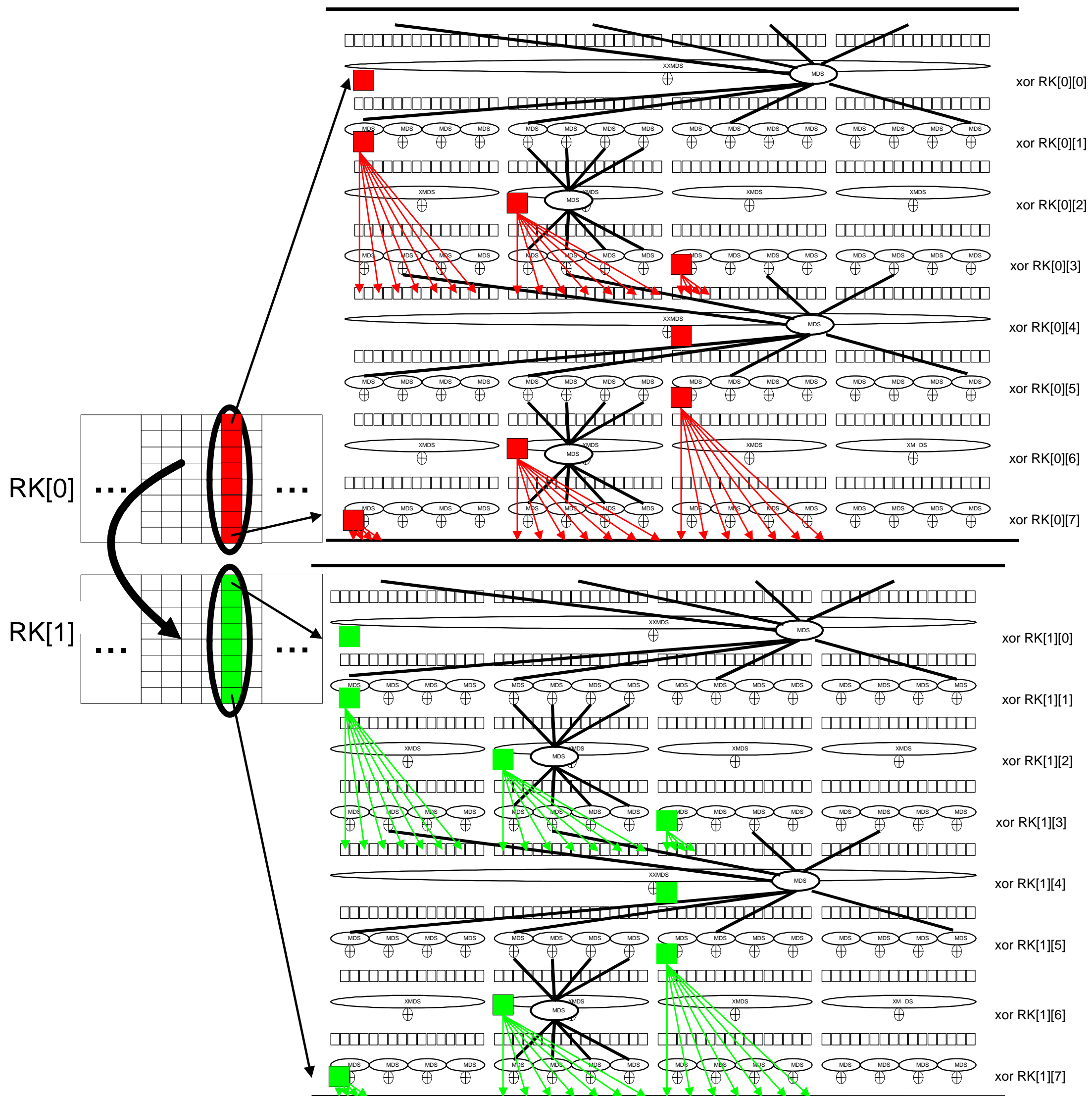

Fig.9: A diffusion example using the final key permutation (r = 8, c = 64)

5. Double Net as a strengthened encryption algorithm

DN was constructed to be used with a constant input plaintext in mind and to become a random oracle in a hash function [Kl06]. We call it special block cipher in such case.

However, if DN is used with variable plaintext, it can be used as an encryption algorithm. In this case, its strong key processing makes it favourable over the classical block ciphers, since it is protected against future attacks.

The key in DN algorithm used for the encryption will not usually be as long as the key in DN algorithm used for hashing. The array r x c can be relatively small and the dimension c (plaintext width in bytes) can be relatively small, as well. A typical 128-bit block cipher with 256-bit key, i.e. c = 16 and r = 2 can be used as an example. The column transformation principles can be preserved even when several neighbouring columns are joined and understood as one “thicker column” (e.g. two columns as on Fig. 10). The column transformation is then applied on this “thicker column”.

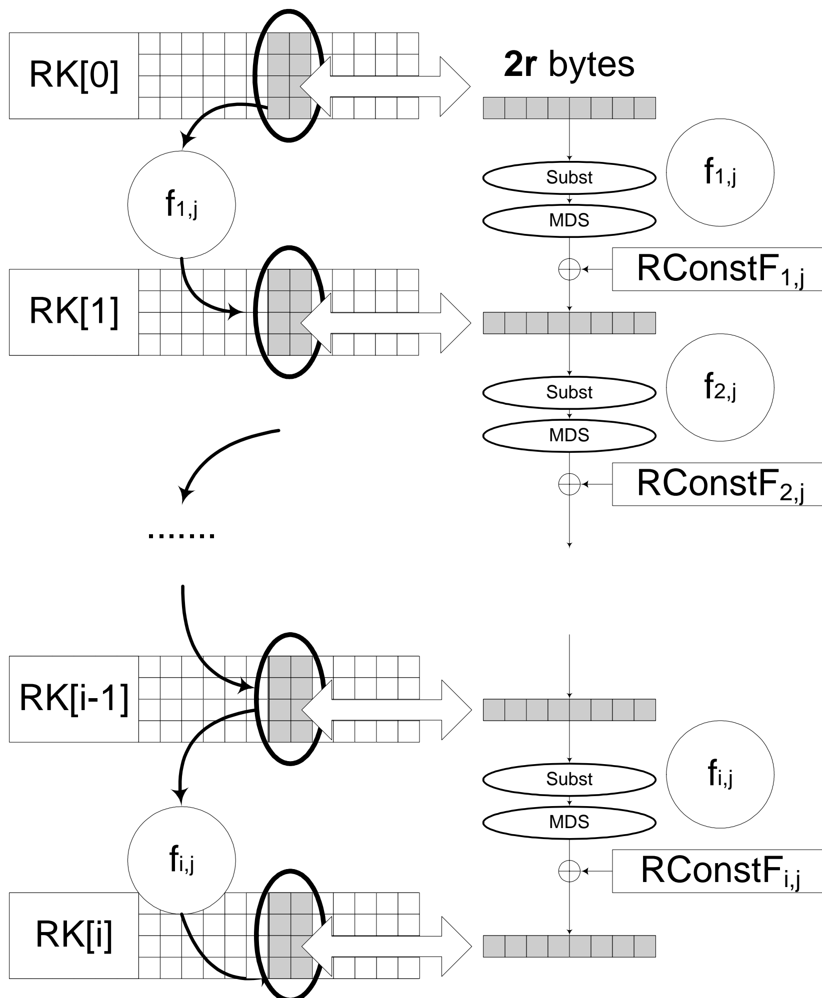

Fig. 10: Column transformation principle applied on several columns.

6. Number of rounds in DN, its variations and hashing speed

6.1. Number of rounds: 6 (10)

The quality of substitution boxes and the dimensions of the round keys used in the function Φ determine the relationship between the number of rounds and the estimate for the resistance of Φ against DC and LC. Similar estimates for the function ∏ can be found easily. To determine the number of rounds in the function Φ, it is important whether DN is used for the encryption or hashing. The security margin influences the number of rounds, as well. Using current S-boxes we set the number of rounds to 10 for the function DN(512, 8182). If higher quality S-boxes are used, the number of rounds can be lowered to as few as 6.

6.2. DN variations

The basic idea of DN is that the keys $a, b, \ldots, z$ to the sub-ciphers of product cipher $\Pi = B_z \bullet \ldots \bullet B_b \bullet B_a$ are generated by a strong block cipher Φ. With increasing number of rounds, the keys $(a, b, \ldots)$ and $(\ldots, y, z)$ become computationally indistinguishable from independent random variables, since they are in plaintext-ciphertext relation for the block cipher Φ. Thus, the block ciphers $(B_a, B_b, \ldots)$ and $(\ldots B_y, B_z)$ themselves become computationally indistinguishable from (independent) random block ciphers. The columns of array RK are mixed by the function Π. Usually, a product of only a few big rounds B will ensure the resistance of the function $\Pi = B_z \bullet \ldots \bullet B_b \bullet B_a$ against DC and LC. For this reason, we can skip the middle part in the product $\Pi = B_z \bullet \ldots \bullet B_b \bullet B_a$ and use only several block cipher B in the beginning and at the end, e.g. three and three ($\Pi = B_z \bullet B_y \bullet B_x \bullet B_c \bullet B_b \bullet B_a$).

6.3. Hashing speed

The speeds of the hash functions HDN(512, 8192), SHA-256, SHA-512 and Whirlpool are compared in this paragraph. These algorithms are all included in the publicly available library Crypto++. Its author is Wei Dei, the source code can be found at http://www.eskimo.com/~weidai/benchmarks.html. All the algorithms were written in C++, compiled (for speed) in Microsoft Visual C++.NET 2003 under Windows XP SP1. Their hashing speeds are displayed in the first part of the following table, while the second part displays our own implementations of SHA-256, SHA-512 and HDN(512, 8192). The tests of our implementations were run on a Pentium 1.6GHz notebook under Windows XP SP2 and were compiled with MS Visual C++ 6.0.

	library Crypto++	**Pentium 4 (2.1 GHz)**	
Algorithm	MB tested	speed in MByte/s	
MD5	1002	216	
SHA-1	256	68	
SHA-256	256	**44**	
SHA-512	64	**11**	
Whirlpool	64	**12**	
	Our implementation	**Pentium, 1.6 GHz**	
Algorithm	MB tested	speed in MByte/s	"three + three" variation of algorithm HDN $\Pi = B_z \bullet B_y \bullet B_x \bullet B_c \bullet B_b \bullet B_a$
SHA-256	64	32	
SHA-512	64	**17**	
HDN(512, 8192)-1	64	136	
HDN(512, 8192)-2	64	35	
HDN(512, 8192)-3	64	20	20.48
HDN(512, 8192)-4	64	14	15.70
HDN(512, 8192)-5	64	11	12.78
HDN(512, 8192)-6	64	**9.09**	10.75
HDN(512, 8192)-7	64	7.67	9.28
HDN(512, 8192)-8	64	6.65	8.15
HDN(512, 8192)-9	64	5.84	7.30
HDN(512, 8192)-10	64	**5.22**	6.57

Tab.: Hashing algorithms speed comparison

The values in Table 1 are only illustrative, as the speed heavily depends on the compiling optimizations. However, we can say HDN(612,8192)-10 is roughly 3 times slower than SHA-512 (and Whirlpool) and HDN(512, 8192)-6 roughly 2 times slower than SHA-512. The only reason to choose 10 big rounds in HDN(512, 8192) was to ensure the function Φ resistance against LC and DC. However, only 6 big rounds are sufficient to ensure the function Π resistance, thus "three + three" variation can be employed, i.e. $\Pi = B_z \bullet B_y \bullet B_x \bullet B_c \bullet B_b \bullet B_a$. The speed measurement show HDN(512, 8192) is not just a theoretical concept, but a practically employable function with speed only 2-3 times lower than SHA-512 and Whirlpool.

7. Conclusion

The special block cipher DN family and SNMAC-type [Kl06] hash function HDN family were presented in this paper. It turns out there are not just theoretical concepts, but practically employable functions only 2-3 times slower than SHA-512 and Whirlpool.

An attacker of a hash function has the possibility to manipulate freely with all of its inputs. However, the construction of the classical block cipher assumes there is a secret element unknown the attacker (the encryption key). As a result the special block cipher construction expects the attacker to know the key or even to freely manipulate with it.

The basic idea behind the special block cipher is simple – contrary to classical block cipher, the same attention is paid to plaintext and key processing. One SP network ensures key mixing, while the second one mixes the plaintext with the key.

Simultaneously, we present new vision of classical block cipher construction – it should be done similarly to the hash function construction. For a long time, it was expected the attacker has no knowledge about the encryption key, nor can manipulate with it. With extensively growing attacker's possibilities thanks to modern technologies, this assumption turns out to be little corresponding with the real-life scenarios. Some attacks are known already – side-channel attacks, related key attacks, rectangular attack, etc. (e.g. see [Bi93], [Bi03], [Ki04], [Ho05], [Ki05], [Bi05], and [Bi06]), other attacks will emerge in the decades to come. Their common traits are the various attempts to exploit the original assumption on the attacker's limited power over the secret key or its knowledge. This is the reason why the hash functions of new generation should be resistant against key originating attacks. The question is if the special block ciphers are the correct solution to this problem. One way or another, the strengthening of the keys processing functions in modern block cipher should be carefully considered.

Acknowledgment. The author thanks Tomáš Rosa for many useful comments to the drafts of this paper and Martin Hlaváč for helpful comments and for translating this paper from Czech.

8. References

[Bi93] E. Biham, *New Types of Cryptanalytic Attacks Using Related Keys*, EUROCRYPT 1993, pp. 398-409, LNCS 765, Springer-Verlag, 1993.

[Bi03] E. Biham, O. Dunkelman, N. Keller, *Rectangle Attacks on 49-Round SHACAL-1*, FSE 2003, pp. 22 - 35, LNCS 2887, Springer-Verlag, 2003.

[Bi94] E. Biham, *On Matsui's Linear Cryptanalysis*, EUROCRYPT'94, LNCS 950, pp. 341-355, Springer-Verlag, 1995.

[BS91a] E. Biham, A. Shamir, *Differential Cryptanalysis of DES-like Cryptosystem*, Journal of Cryptology, Vol.4, pp. 3-72, 1991.

[BS91b] E. Biham, A. Shamir, *Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer*, CRYPTO'91, LNCS 576, pp. 156-171, Springer-Verlag, 1992.

[Bi05] E. Biham, O. Dunkelman, N. Keller, *Related-Key Boomerang and Rectangle Attacks*, EUROCRYPT 2005, LNCS 3494, pp. 507–525, 2005.

[Bi06] E. Biham, O. Dunkelman, N. Keller, *Related-Key Impossible Differential Attacks on 8-Round AES-192*, CT-RSA 2006, LNCS 3860, pp. 21–33, Springer-Verlag, 2006.

[Da95] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis*, Doctoral Dissertation, March 1995, K.U. Leuven.

[Ho00] S. Hong, S. Lee, J. Lim, J. Sung, D. Cheon, I. Cho, *Provable Security against Differential and Linear Cryptanalysis for the SPN Structure*, FSE 2000**,** LNCS 1978, pp. 273 - 283, Springer-Verlag, 2000.

[Ho05] S. Hong, J. Kim, S. Lee, B. Preneel, *Related-Key Rectangle Attacks on Reduced Versions of SHACAL-1 and AES-192*, FSE 2005, LNCS 3557, pp. 368–383, Springer-Verlag, 2005.

[Chu03] K. Chun, S. Kim, S. Lee, S.H. Sung, S. Yoon, *Differential and linear cryptanalysis for 2-round SPNs*, Information Processing Letters, Vol. 87 (2003), pp. 277 - 282.

[Ka01] J. Kang, S. Hong, S. Lee, O. Yi, Ch. Park, J. Lim, *Practical and provable security against differential and linear cryptanalysis for substitution-permutation networks*, ETRI Journal, 23(4):158–167, 2001.

[Ki04] J. Kim, G. Kim, S. Hong, S. Lee, D. Hong, *The Related-Key Rectangle Attack-Application to SHACAL-1*, ICISP 2004, LNCS 3108, pp. 123-136, Springer - Verlag, 2004.

[Ki05] J. Kim, A. Biryukov, B. Preneel, S. Lee, *On the Security of Encryption Modes of MD4, MD5 and HAVAL*, Cryptology ePrint Archive: Report 2005/327, September - October 2005, ICICS 2005, LNCS 3783, Springer-Verlag, http://eprint.iacr.org/2005/327.pdf
[Kl06] V. Klima, *A New Concept of Hash Functions SNMAC Using a Special Block Cipher and NMAC/HMAC Constructions*, IACR ePrint archive Report 2006/376, October, 2006, http://eprint.iacr.org/2006/376.pdf

[LM91] X. Lai, J. Massey, S. Murphy, *Markov Ciphers and Differential Cryptanalysis*, EUROCRYPT'91, LNCS 547, pp 17-38, Springer-Verlag, 1992.

[Ma93] M. Matsui, *Linear cryptanalysis method for DES cipher*, EUROCRYPT' 93, LNCS 765, pp. 386-397, Springer-Verlag, 1993.

[Ma94] M. Matsui, *The first Experimental cryptanalysis of DES*, CRYPTO'94, LNCS 839, pp. 1-11, Springer-Verlag, 1994.

[NK92] K. Nyberg, L. Knudsen, *Provable security against a differential attack*, CRYPTO'92, LNCS 740, pp. 566-574, Springer-Verlag, 1992.

[Ny94] K. Nyberg, *Linear Approximation of block ciphers*, EUROCRYPT'94, LNCS 950, pp. 439-444, Springer-Verlag, 1994.

[PD05] J. Plank, Y. Ding, *Note: Correction to the 1997 tutorial on Reed-Solomon coding*, Software: Practice and Experience, Volume 35, Issue 2, pp. 189-194, 2005, http://www.cs.utk.edu/~plank/plank/papers/SPE-9-97.html

[RD97] V. Rijmen, J.Daemen et al, *The cipher SHARK*, FSE´97, LNCS 1267, pp. 137-151, Springer-Verlag, 1997.

[Ro06] R. Roth, *Introduction to Coding Theory*, Cambridge University Press, 2006, p. 148.

[Sa03] F. Sano, K. Ohkuma, H. Shimizu, S. Kawamura, *On the security of nested SPN cipher against the differential and linear cryptanalysis*, IEICE Trans. Fundamentals, Vol. E86-A, No.1, January 2003, pp. 37 - 46.

9. Appendix A: SP networks theory

In the beginning of this chapter, we present the results in SP networks theory that is used in functions Φ and Π construction. We introduce the design rules of building blocks (parameters) in functions Φ and Π, afterwards. The most of this chapter employs definitions and theorems from [Ho00].

9.1. DC, LC and SPN

Differential cryptanalysis DC ([BS91a], [BS91b], [Bi94]) and linear cryptanalysis LC ([Ma93], [Ma94]) are the most known attacks on block ciphers.

Differential cryptanalysis of a block cipher with several rounds investigates differential characteristics of individual rounds, i.e. the probabilities specific differences on the input of a round are transferred to specific differences on the round output. It turns out it is not practical to examine fixed input/output differences in a block cipher round. So-called differential [LM91] is a better resistance indicator. The differential is the probability a specific difference on the input of the (full) block cipher corresponds to a specific difference on the output of the (full) cipher, ignoring the inner-differences in the individual rounds.

Similarly, the linear characteristic was replaced by the lineal hull [Ny94] in case of LC. Clearly, the computation of the differential and the linear hull for several rounds of a block cipher becomes a very hard problem.

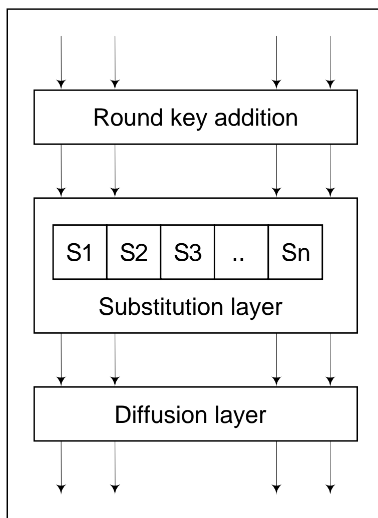

Fig. A.1: One round of SPN network

K. Nyberg and L.R. Knudsen showed in [NK92] the *r*-round scheme differential probability is bounded by the value $2p^2$, if the maximal probability of the round function differential is p and $r \geq 4$. It is only p^2, if the round function is a 1-to-1 mapping. As the diffusion layer in substitution-permutation networks causes the avalanche effect (with respect to differences and linear approximations), branch number

was put to use [Da95]. This number is very important, as fatally weak block ciphers with S-boxes resistant against LC and DC might exists, if the value of their branch number is low. SPN resistance against DC and LC is proved in [Ho00], under the condition the value of branch number is maximal. We only use maximal diffusion layer, i.e. maximal branch number, in functions Φ and Π. Our theorems proving the resistance of Φ and Π against DC and LC are based on two main theorems in [Ho00]. Let us introduce the notation needed, first.

9.2. Notation

SPN with *mn*-bit round function is considered in this paper, with $2n$ S-boxes (S_1, ..., S_{2n}). Each S-box is a 1-to-1 mapping on the set $\{0, 1\}^m$, S_i: $\{0, 1\}^m \rightarrow \{0, 1\}^m$, $i = 1, ..., 2n$.

Definition 1. Linear and differential probability of an S-box

Differential and linear probabilities of a (bijective) S-box S: $\{0, 1\}^m \rightarrow \{0, 1\}^m$ are defined for $\Delta x, \Delta y, \Gamma x, \Gamma y \in \{0, 1\}^m$ as

$DP^S(\Delta x \rightarrow \Delta y) = \#\{x \in \{0, 1\}^m \mid S(x) \oplus S(x \oplus \Delta x) = \Delta y \} / 2^m$,

$LP^S(\Gamma x \rightarrow \Gamma y) = [\ \#\{x \in \{0, 1\}^m \mid \Gamma x \bullet x = \Gamma y \bullet S(x) \}/2^{m-1} - 1\ ]^2$, where

$\Gamma x \bullet x$ is parity of $\Gamma x \oplus x$.

Definition 2. Maximal linear and differential probability of an S-box

Maximal linear and differential probability of an (bijective) S-box S: $\{0, 1\}^m \rightarrow \{0, 1\}^m$ is defined as

$\mathbf{DP^S} = \max DP^S(\Delta x \rightarrow \Delta y)$, where the maximum is taken over all $\Delta x \neq 0$, $\Delta x \in \{0, 1\}^m$, $\Delta y \in \{0, 1\}^m$,

$\mathbf{LP^S} = \max LP^S(\Gamma x \rightarrow \Gamma y)$, where the maximum is taken over all $\Gamma x, \Gamma y \neq 0$, $\Gamma x \in \{0, 1\}^m$, $\Gamma y \in \{0, 1\}^m$.

S-box is called strong, if these numbers are small. If they are small for all S-boxes in SPN, the network is called strong. For SPN let's define

$\boldsymbol{p} = \max DP^S$,

$\boldsymbol{q} = \max LP^S$,

where the maximum is taken over all S-boxes, used in SPN.

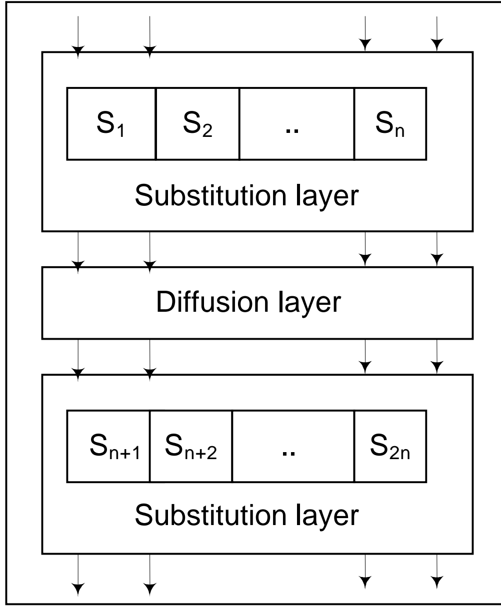

Fig. A.2: Function SDS

Function SDS. SDS is a three layers function: substitution (S), diffusion layer (D) and substitution (S), see Fig. A.2. Let's denote the input and the output difference in SDS as $\Delta x \in \{0, 1\}^{nm}$, $\Delta x \neq 0$, $\Delta y \in \{0, 1\}^{nm}$, $\Delta y = y \oplus y^* = D(x) \oplus D(x^*)$, and the input and the output mask in SDS are $\Gamma x \in \{0, 1\}^{nm}$, $\Gamma y \in \{0, 1\}^{nm}$, $\Gamma y \neq 0$ (a linear relationship between Γx and Γy exists, see [RD97] for details).

Minimal number of differentially and linearly active S-boxes. Minimal number of differentially and linearly active S-boxes of function SDS is defined as:
$n_d(D) = \min(Hw(\Delta x) + Hw(\Delta y))$, where the minimum is taken over all $\Delta x \neq 0$,
$n_l(D) = \min(Hw(\Gamma x) + Hw(\Gamma y))$, where the minimum is taken over all $\Gamma y \neq 0$.

Maximal diffusion layer. A diffusion layer is called maximal if the minimal number of differentially (or equivalently linearly) active boxes is equal to $n + 1$. It is known [Ho00] an RS (2n, n, n+1) code can be used to construct a maximal diffusion layer. If the generator of this code is a matrix in the form $[I_{nxn}\ B_{nxn}]$, then $D: GF(2^m)^n \rightarrow GF(2^m)^n : x \rightarrow Bx$ is maximal diffusion layer [RD97].

9.3. Main theorems

In this paper, we assume the round key exored to the data in each round are distributed uniformly and independently. Under this assumption, the round key addition has no effect on the number of active S-boxes in the round function. If the diffusion layer is maximal, the Theorem 1 provides an upper bound on the differential of function SDS (seen as a whole).

Theorem 1. Upper bound on differential of function SDS [Ho00]. Let the round keys exored to input data in each round be distributed uniformly and independently. If the diffusion layer D is maximal (i.e. $n_d = n + 1$), then the probability of each differential of function SDS is bounded by p^n.

Corollary. By Theorem 1 $DP^{SDS}(\Delta x \rightarrow \Delta y) \leq p^n$ for $\Delta x \in \{0, 1\}^{nm}$, $\Delta x \neq 0$, $\Delta y \in \{0, 1\}^{nm}$. It follows
$\mathbf{DP^{SDS}} = \max DP^{SDS}(\Delta x \rightarrow \Delta y) \leq p^n$,
where the maximum is taken over all $\Delta x \neq 0$, $\Delta x \in \{0, 1\}^{nm}$, $\Delta y \in \{0, 1\}^{nm}$.

A similar bound is valid for the linear hull of function SDS.

Theorem 2. Upper bound of linear hull of function SDS [Ho00]. Let the diffusion layer D be maximal (i.e.. n_l(D) = n + 1 or equivalently n_d(D) = n + 1), then the probability of each linear hull of function SDS is bounded by q^n.

Corollary 1. By Theorem 2 $LP^{SDS}(\Gamma x \rightarrow \Gamma y) \leq q^n$ for each $\Gamma x \in \{0, 1\}^{nm}$, $\Gamma y \neq 0$, $\Gamma y \in \{0, 1\}^{nm}$. It follows
$\mathbf{LP^{SDS}} = \max LP^{SDS}(\Gamma x \rightarrow \Gamma y) \leq q^n$,
where the maximum is taken over all $\Gamma x \in \{0, 1\}^{nm}$, $\Gamma y \neq 0$, $\Gamma y \in \{0, 1\}^{nm}$.

We will keep investigating the differentials only; similar propositions hold for the linear hulls, however, due to the similarities between Theorems 1 and 2. Theorems 1 and 2 are fundamental, as they provide the bounds of the differential and the linear hull. Until now, such bounds were hard to achieve with a classical block cipher. The product of round characteristics was used as their substitute, in this case.

Corollary 2. Compared to S-box, an SDS network can be seen as a bigger S-box, a so-called “XS”-box. As box XS is constructed as an SDS network consisting of small boxes S, by Theorem 1 (2), its maximal differential (linear hull) can be estimated by the maximal differentials (linear hulls) of these small boxes. Bigger XXS-boxes can be constructed from XS-boxes, etc. We will use this principle when constructing and proving the properties of networks Φ and Π.

10. Appendix B: Definitions of variable elements in DN(512,8192)

We present the specific selection of the parameters of function DN(512, 8192) with ρ big rounds ρ = 1, ..., 10, in this chapter. We set r = 16, c = 64.

As columns process key, its size can be relatively large (8192 bits). When DN is used in hash function construction, function HDN(512,8192) is obtained that has 512-bit hash code and processes messages by 7680 bit blocks (7680 = 8192 - 512). The definition of variable elements in function HDN(512, 8192) are presented in the next chapter.

Function DN employs the substitution boxes coming from the Whirlpool algorithm. The original version of block cipher W in Whirlpool sent to NESSIE project used a (pseudo)randomly generated S-box 8 x 8 with no special algebraic properties. For the reasons of a more efficient HW implementation, this box was changed to two smaller S-boxes 4 x 4 later.

Sources:
(a) The most recent version of specification – corresponds to selected algorithm NESSIE and ISO norm ISO/IEC 10118-3 (changed S-box and changed matrix MDS): Paulo S.L.M. Barreto and Vincent Rijmen: The WHIRLPOOL Hashing Function, (Revised on May 24, 2003)
http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip
(b) The second most recent specification (changed S-box) from 7.3.2003
https://www.cosic.esat.kuleuven.be/nessie/updatedPhase2Specs/WHIRLPOOL/Whirlpool-tweak2.zip
(c) Original specification from September 2000 (original S-box)
https://www.cosic.esat.kuleuven.be/nessie/workshop/submissions/whirlpool.zip

10.1. Function F: original S-box in Whirlpool algorithm

The original S-box kept being generated (pseudo)randomly until it satisfied these conditions (
https://www.cosic.esat.kuleuven.be/nessie/workshop/submissions/whirlpool.zip, September 3, 2000):
(a) $\delta \leq 8*2^{-8}$,
(b) $\lambda \leq 16 * 2^{-6}$,
(c) $\nu = 7$,
(d) no fixed points,
(e) there is no value appearing more than twice in the set $(x \oplus S(x))$.
The chosen S-box had these properties
(a) $\delta = 8*2^{-8} = 2^{-5}$,
(b) $\lambda = 16 * 2^{-6} = 2^{-2}$,
(c) $\nu = 7$,
(d) no fixed points,
(e) there is no value appearing more than twice in the set $(x \oplus S(x))$.

We call it the **original S-box** in Whirlpool algorithm. We use it in function F, as it has less inner structure than the second S-box in Whirlpool.

```
unsigned char SubsF[256] = {
0x68, 0xd0, 0xeb, 0x2b, 0x48, 0x9d, 0x6a, 0xe4,
0xe3, 0xa3, 0x56, 0x81, 0x7d, 0xf1, 0x85, 0x9e,
0x2c, 0x8e, 0x78, 0xca, 0x17, 0xa9, 0x61, 0xd5,
0x5d, 0x0b, 0x8c, 0x3c, 0x77, 0x51, 0x22, 0x42,
0x3f, 0x54, 0x41, 0x80, 0xcc, 0x86, 0xb3, 0x18,
0x2e, 0x57, 0x06, 0x62, 0xf4, 0x36, 0xd1, 0x6b,
0x1b, 0x65, 0x75, 0x10, 0xda, 0x49, 0x26, 0xf9,
0xcb, 0x66, 0xe7, 0xba, 0xae, 0x50, 0x52, 0xab,
0x05, 0xf0, 0x0d, 0x73, 0x3b, 0x04, 0x20, 0xfe,
0xdd, 0xf5, 0xb4, 0x5f, 0x0a, 0xb5, 0xc0, 0xa0,
0x71, 0xa5, 0x2d, 0x60, 0x72, 0x93, 0x39, 0x08,
0x83, 0x21, 0x5c, 0x87, 0xb1, 0xe0, 0x00, 0xc3,
0x12, 0x91, 0x8a, 0x02, 0x1c, 0xe6, 0x45, 0xc2,
0xc4, 0xfd, 0xbf, 0x44, 0xa1, 0x4c, 0x33, 0xc5,
```

```
0x84, 0x23, 0x7c, 0xb0, 0x25, 0x15, 0x35, 0x69,
0xff, 0x94, 0x4d, 0x70, 0xa2, 0xaf, 0xcd, 0xd6,
0x6c, 0xb7, 0xf8, 0x09, 0xf3, 0x67, 0xa4, 0xea,
0xec, 0xb6, 0xd4, 0xd2, 0x14, 0x1e, 0xe1, 0x24,
0x38, 0xc6, 0xdb, 0x4b, 0x7a, 0x3a, 0xde, 0x5e,
0xdf, 0x95, 0xfc, 0xaa, 0xd7, 0xce, 0x07, 0x0f,
0x3d, 0x58, 0x9a, 0x98, 0x9c, 0xf2, 0xa7, 0x11,
0x7e, 0x8b, 0x43, 0x03, 0xe2, 0xdc, 0xe5, 0xb2,
0x4e, 0xc7, 0x6d, 0xe9, 0x27, 0x40, 0xd8, 0x37,
0x92, 0x8f, 0x01, 0x1d, 0x53, 0x3e, 0x59, 0xc1,
0x4f, 0x32, 0x16, 0xfa, 0x74, 0xfb, 0x63, 0x9f,
0x34, 0x1a, 0x2a, 0x5a, 0x8d, 0xc9, 0xcf, 0xf6,
0x90, 0x28, 0x88, 0x9b, 0x31, 0x0e, 0xbd, 0x4a,
0xe8, 0x96, 0xa6, 0x0c, 0xc8, 0x79, 0xbc, 0xbe,
0xef, 0x6e, 0x46, 0x97, 0x5b, 0xed, 0x19, 0xd9,
0xac, 0x99, 0xa8, 0x29, 0x64, 0x1f, 0xad, 0x55,
0x13, 0xbb, 0xf7, 0x6f, 0xb9, 0x47, 0x2f, 0xee,
0xb8, 0x7b, 0x89, 0x30, 0xd3, 0x7f, 0x76, 0x82
};
```
Fig. B.1: Original S-box in Whirlpool algorithm

10.2. Number of rounds ρ

For implementation reasons, we choose all S-boxes identical with parameters $p = DP^S = 2^{-5}$ and $q = LP^S = 2^{-2}$, in the definition of function F. We use the original pseudorandom S-box in Whirlpool algorithm that is not generated algebraically. However, it has lower resistance against linear cryptanalysis (q) for this reason. To ensure sufficient resistance including security margin, we have to set the number of round ρ unreasonably big. **Instead of sufficient 6 rounds, we set ρ = 10.** As soon as there is a publicly generated S-box with better properties available, we can use it in function DN with lower number of rounds (we recommend $\rho = 6$).

Five iterated SDS networks connected with MDS type matrices with dimension 16 x 16 form 10 rounds of F. The diffusion layer inside the SDS network is ensured by MDS matrix with dimensions 16 x 16, as well. To see this, it suffices to apply Theorems 1 and 2 to obtain the bounds for DP^{SDS} and LP^{SDS}, i.e. $p_{SDS} \leq 2^{-80}$ and $q_{SDS} \leq 2^{-32}$. Relatively low bound $LP^{SDS} \leq 2^{-32}$ is caused by the linear characteristic of the box employed ($q = 2^{-2}$). There are better choices for S-box in F, e.g. $q = 2^{-6}$ as in AES or with the expected coefficient $q = 2^{-4}$ for (pseudo) randomly generated S-boxes. Completely sufficient bound $LP^{SDS} \leq 2^{-96}$ is attainable with AES S-box $q = 2^{-6}$ and relatively good bound $LP^{SDS} \leq 2^{-64}$ with coefficient $q = 2^{-4}$. From the point of view of the resistance of F against LC, three SDS consecutive networks are sufficient (i.e. 6 big rounds), for both of these cases. As the resistance of F against differential cryptanalysis is achieved in similar way, three consecutive SDS networks (i.e.6 big rounds) are sufficient, as well.

10.3. Round constants RConstF

We showed the round constants cause an affine transformation of each S-box. For the reasons of efficient SW and HW implementations, we choose the constants that can be generated on-the-fly: $RConstF[i][j] = ((CONST_A * (i + 1)) \bmod 2^{32} \oplus ((CONST_B * (j +$

1)) mod 2^{32}), where $CONST_A$ = 0xfedc1357, $CONST_B$ = 0x84736251. These constants are only four bytes long (first 12 bytes are all zeroes) and pair wise different.

10.4. Field GF(2^8)

The irreducible polynomial used for finite field GF(2^8) representation is $q(x) = x^8 + x^4 + x^3 + x^1 + x^0$. The same lookup tables Logtable and Alogtable as in AES are used to perform the field multiplication in the source code.

10.5. 16 x 16 matrix MDS

A single MDS matrix with dimensions 16 x 16 is used in function F for the reasons of efficiency in SW and HW. Its selection was based on [PD05] and [Ro06] and is elaborated over the field GF(2^8) with irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x^1 + x^0$. The foundation is the matrix G of Vandermond type 16 x 32, $G = (g_{i,j})_{i = 0 \ldots 15,\, j = 0 \ldots 31}$, where 32 pair wise different elements $a_0, a_1, a_2, \ldots, a_{31}$, are selected with $a_0 = 1$.We set $a_1 = 12$, $a_2 = 13$, ..., $a_{31} = 42$, i.e. $a_j = (j + 11)$ for $j = 1, \ldots, 31$. We define $g_{i,j} = a_j^{\,i}$, where $i = 0, \ldots, 15$, $j = 0, \ldots, 31$.

$$G = \begin{matrix} 1 & 1 & 1 & \ldots\ldots & 1 & 1 \\ 1 & 12 & 13 & & 41 & 42 \\ 1 & 12^2 & 13^2 & & 41^2 & 42^2 \\ 1 & \ldots & & (a_j)^i & & \ldots \\ 1 & \ldots & & & & \ldots \\ 1 & 12^{15} & 13^{15} & \ldots & 41^{15} & 42^{15} \end{matrix}$$

Let us denote the left half of the matrix G as G1 and the right half as G2. By elementary transformations, we transform the matrix G = (G1, G2) to the form G = (I , F), where I is identity type 16 x 16 matrix and F is a type 16 x 16 matrix. The resulting matrix F is a type 16 x 16 MDS matrix.

The elementary transformations done on the rows of the matrix G are these

- swapping two rows,
- multiplying or dividing a row with a non-zero element of the field,
- adding a non-zero multiple of a row to another row.

We use the transposed matrix F as the resulting MDS matrix (hexadecimal).

```
4A 7B BA CF 84 8D B7 C6 72 9F 24 B2 7A 40 B1 CD,
70 70 A8 4F 79 8B BB 60 A1 38 99 99 F5 AA FA F3,
91 09 E8 D7 B2 DC 10 C0 69 CF D2 6F 6F 56 5B 61,
17 99 94 CF B4 4D 92 62 6E 9A 62 EA 0D 6B 29 EE,
54 1B A9 49 F4 28 21 65 E4 D3 54 50 C9 CF B1 B2,
80 4A 39 F2 62 16 72 B6 8C 06 57 5A D0 22 AE 6B,
2B 34 BF B4 3C 3C 9E E8 0E 9D CB 66 48 B1 91 35,
11 DF E7 64 B2 64 AE 66 33 9F 47 85 80 7C 61 A1,
5A 3A CD 6F 58 A3 D3 2C 73 AC 22 A9 EE EC EC 7D,
0C 2C 83 77 1B 4C AC 79 A9 83 C8 E8 A7 87 D0 AD,
8D A2 42 DD 5D 4D B7 B7 71 93 93 E6 CE 72 EF 65,
```

```
E6 A7 61 CA 05 70 B8 D4 11 86 E2 6D 61 93 11 09,
FA D3 BD D0 E8 11 5B 61 F8 EC 6A 86 D0 36 4A D2,
08 F5 C5 15 D8 E2 55 EC 30 63 74 7E 2A C2 6C 72,
F4 6B C4 AB 68 40 09 C0 96 62 C6 86 6E 9F 7E 2B,
34 F0 19 66 6A 6D 73 08 22 16 11 9B 33 F4 5D E2.
```

10.6. Final key permutation

In DN(512, 8192), the final key permutation is a simple cyclical shift within the rows of array RK.

```
row  0 is cyclically shifted to right by  0 positions
row  1 is cyclically shifted to right by  0 positions
row  2 is cyclically shifted to right by 16 positions
row  3 is cyclically shifted to right by 32 positions
row  4 is cyclically shifted to right by 32 positions
row  5 is cyclically shifted to right by 32 positions
row  6 is cyclically shifted to right by 16 positions
row  7 is cyclically shifted to right by  0 positions

row  8 is cyclically shifted to right by  0 positions
row  9 is cyclically shifted to right by  0 positions
row 10 is cyclically shifted to right by 16 positions
row 11 is cyclically shifted to right by 32 positions
row 12 is cyclically shifted to right by 32 positions
row 13 is cyclically shifted to right by 32 positions
row 14 is cyclically shifted to right by 16 positions
row 15 is cyclically shifted to right by  0 positions
```

This pattern is periodically repeated until the row number 159.

10.7. Function B: S-box generated for Whirlpool algorithm

S-box generated in the updated Whirlpool algorithm consists of two small 4 x 4 S-boxes due to better implementation in HW. Its description is included in NESSIE report from March 7, 2003 (and later ones from May 24, 2003 as well, with the adjusted matrix MDS), it makes part of the final Whirlpool algorithm in NESSIE project and is employed in ISO norm. This S-box with characteristics $p = 2^{-5}$ a $q = 14*2^{-6}$ is employed in function B.

```
unsigned char SubsB[256] = {
0x18,0x23,0xc6,0xE8,0x87,0xB8,0x01,0x4F,
0x36,0xA6,0xd2,0xF5,0x79,0x6F,0x91,0x52,
0x60,0xBc,0x9B,0x8E,0xA3,0x0c,0x7B,0x35,
0x1d,0xE0,0xd7,0xc2,0x2E,0x4B,0xFE,0x57,
0x15,0x77,0x37,0xE5,0x9F,0xF0,0x4A,0xdA,
0x58,0xc9,0x29,0x0A,0xB1,0xA0,0x6B,0x85,
0xBd,0x5d,0x10,0xF4,0xcB,0x3E,0x05,0x67,
0xE4,0x27,0x41,0x8B,0xA7,0x7d,0x95,0xd8,
0xFB,0xEE,0x7c,0x66,0xdd,0x17,0x47,0x9E,
0xcA,0x2d,0xBF,0x07,0xAd,0x5A,0x83,0x33,
0x63,0x02,0xAA,0x71,0xc8,0x19,0x49,0xd9,
0xF2,0xE3,0x5B,0x88,0x9A,0x26,0x32,0xB0,
```

```
0xE9,0x0F,0xd5,0x80,0xBE,0xcd,0x34,0x48,
0xFF,0x7A,0x90,0x5F,0x20,0x68,0x1A,0xAE,
0xB4,0x54,0x93,0x22,0x64,0xF1,0x73,0x12,
0x40,0x08,0xc3,0xEc,0xdB,0xA1,0x8d,0x3d,
0x97,0x00,0xcF,0x2B,0x76,0x82,0xd6,0x1B,
0xB5,0xAF,0x6A,0x50,0x45,0xF3,0x30,0xEF,
0x3F,0x55,0xA2,0xEA,0x65,0xBA,0x2F,0xc0,
0xdE,0x1c,0xFd,0x4d,0x92,0x75,0x06,0x8A,
0xB2,0xE6,0x0E,0x1F,0x62,0xd4,0xA8,0x96,
0xF9,0xc5,0x25,0x59,0x84,0x72,0x39,0x4c,
0x5E,0x78,0x38,0x8c,0xd1,0xA5,0xE2,0x61,
0xB3,0x21,0x9c,0x1E,0x43,0xc7,0xFc,0x04,
0x51,0x99,0x6d,0x0d,0xFA,0xdF,0x7E,0x24,
0x3B,0xAB,0xcE,0x11,0x8F,0x4E,0xB7,0xEB,
0x3c,0x81,0x94,0xF7,0xB9,0x13,0x2c,0xd3,
0xE7,0x6E,0xc4,0x03,0x56,0x44,0x7F,0xA9,
0x2A,0xBB,0xc1,0x53,0xdc,0x0B,0x9d,0x6c,
0x31,0x74,0xF6,0x46,0xAc,0x89,0x14,0xE1,
0x16,0x3A,0x69,0x09,0x70,0xB6,0xd0,0xEd,
0xcc,0x42,0x98,0xA4,0x28,0x5c,0xF8,0x86
};
```

Fig. B.2: S-box generated in the updated Whirlpool algorithm

10.8. Permutation SMLPerm

Function DN family uses different partial permutations SMLPerm on the set 0, ..., c - 1, for different transformations T1. Only four different permutations on the set 0, ..., 63 (decimally) are used in DN(512, 8192):

```
23,14,49,32,41, 8,50,18,46,16,15,57,55,27,43, 2,
60, 7,22,42,38,26,53,12, 9,62,37,28, 0,36,51,20,
17,39, 4,56,59, 3,47,31, 6,25,45,48,24,58,11,33,
29,13,40,61, 1,19,63,34,52,35, 5,30,44,54,10,21,

17,42,57, 6,62, 8,24,12, 3,21,55,51,44,34,39,31,
36, 2,25,58, 7,47,53,14,49, 9,16,30,33,60,22,40,
41,37,50,15, 1,45,19,63,35,10,59,52,27,20, 4,28,
13,56,23,46,48,32,26,18,61,43,29,54, 5,11, 0,38,

10,15, 4, 1, 5, 0,14,11, 2, 8, 7,13, 6, 9, 3,12,
26,31,20,17,21,16,30,27,18,24,23,29,22,25,19,28,
42,47,36,33,37,32,46,43,34,40,39,45,38,41,35,44,
58,63,52,49,53,48,62,59,50,56,55,61,54,57,51,60,

10, 5,12, 2, 7, 9, 0,15, 1,11, 4,14, 8, 3,13, 6,
26,21,28,18,23,25,16,31,17,27,20,30,24,19,29,22,
42,37,44,34,39,41,32,47,33,43,36,46,40,35,45,38,
58,53,60,50,55,57,48,63,49,59,52,62,56,51,61,54.
```

This block of four permutations is repeated three more time within each big round. The big rounds use the same set of SMLPerm. The values of these permutations are selected so that the matrices MDS (XMDS, XXMDS, XXXMDS) would provide maximal diffusion layer.

10.9. 4 x 4 MDS matrices

For the reasons of simple implementation, a single 4 x 4 MDS matrix was chosen for DN, the one from AES algorithm.

$$M = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

10.10. Round constant RConstB

The reasons of efficient SW and HW implementations, we choose the round constants that can be generated on the fly in function B and set their first 60 bytes as zeros. The last four bytes, seen as 32-bit numbers are generated by the formula RConstB[i][j] = ($CONST_C$*(16*i + j + 1)) mod 2^{32}, where $CONST_C$ = 0x24687531. The least significant byte of the 32-bit number is 61st byte of the constant, the most significant byte is 64th.

The values of all of the parameters are to be seen in the source code later on.

11. Appendix C: Description of variable elements in HDN(512, 8192)

If DN(512, 8192) is used in a hash function following the construction SNMAC [Kl06], hash function HDN(512, 8192) is obtained with 512-bit code, processing the blocks of 7680 bits.

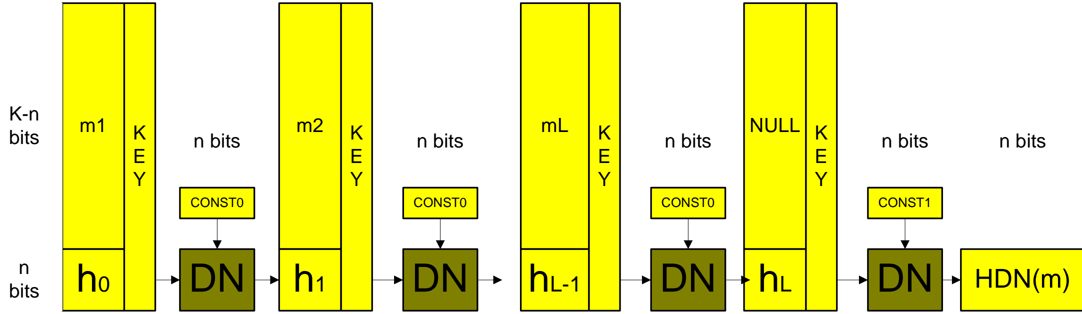

Fig. C.1: HDN(512, 8192) defined as SNMAC based on special block cipher DN(512, 8192)

Definition. Hash function HDN(512, 8192) is a SNMAC type hash function based on special block cipher DN(512, 8192). It has n-bit hash code (n = 512), K bit key (K = 8192) and processes K - n bit data blocks (K - n = 7680). It employs compression function f and final modification function g, where

f: $\{0, 1\}^K \rightarrow \{0, 1\}^n : X \rightarrow E_X(Const_0)$,

g: $\{0, 1\}^n \rightarrow \{0, 1\}^n : X \rightarrow E_{X \,||\, NULL}(Const_1)$,

and E is DN(512, 8192).

$Const_0$ and $Const_1$ are different constants and NULL is an array of K - n zero bits.

Message hashing is completed in three steps.

Step 1. Padding

Message m being hashed is padded by this (bit) string: a single bit 1, the least possible amount of bits 0 and 128 bit long number D (expression the binary length of m), so that the final message length could be expressed as $L(K - n)$ bits, for L an integer. The bit and byte orientation is the same as in SHA-512 standard, i.e. the last bit of block m_L contains the least signification bit of number D. The padded message is divided into L blocks of K - n bits, $m = m_1 \,||\, ... \,||\, m_{L-1} \,||\, m_L$. The same padding is used in function SHA-512.

Step 2. Iteration

$h_i = f(h_{i-1} \,||\, m_i)$, $i = 1, ..., L$,

where h_0 is constant initialization value (IV).

Step 3. Final modification

SNMAC(m) = $g(h_L)$.

$Const_0$, $Const_1$ and h_0 (IV)

The constants $Const_0$, $Const_1$ and h_0 (IV) could be selected randomly (but different). For the reasons of easy implementation, we select them to be easily generated on the fly. Their values (decimally) are:

```
Const0:
128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,

Const1:
 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94,
96, 98,100,102,104,106,108,110,112,114,116,118,120,122,124,126.

IV:
 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63.
```

The values of all of the parameters are to be seen in the source code attached.

12. Appendix D: Original source codes of DN(512, 8192) and HDN(512, 8192)

Note that updates are available on http://cryptography.hyperlink.cz/ .

12.1. Module dn.h

```
/*  dn.h */
#ifndef __TRANS_H__
#define __TRANS_H__

#include <stdio.h>
#include <memory.h>

#define MAXRHO 10 //number of big rounds
#define c 64 // number of columns
#define r 16 // number of rows

unsigned char mul(unsigned char a, unsigned char b);
unsigned char Inv(unsigned char a);

void Copy64(unsigned char* in,unsigned char* out);

void Init_MDS4x4_tables(void);
void Init_MDS16x16_tables(void);

int Check_Matrix(void);
int Check_Const(void);

void ExpandRK(unsigned char RK[MAXRHO][r][c],int rho,int print);
void DN(unsigned char RK[MAXRHO][r][c],
                    int rho,
                    unsigned char indata[c],
                    unsigned char outdata[c],
```

```
                      int print);
#endif
```

12.2. Module dn_constants.h

```
/* dn_constants.h */
/* Tables for multiplication in GF(2^8), the same as in AES. Irreducible
polynomial q(x) = x^8 + x^4 + x^3 + x^1 + x^0
*/

unsigned char Logtable[256] = {
  0,   0,  25,   1,  50,   2,  26, 198,  75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,  76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,   9, 120,
101,  47, 138,   5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
102, 221, 253,  48, 191,   6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
 43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
 44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,
151, 178, 135, 144,  97, 190, 220, 252, 188, 149, 207, 205,  55,  63,  91, 209,
 83,  57, 132,  60,  65, 162, 109,  71,  20,  42, 158,  93,  86, 242, 211, 171,
 68,  17, 146, 217,  35,  32,  46, 137, 180, 124, 184,  38, 119, 153, 227, 165,
103,  74, 237, 222, 197,  49, 254,  24,  13,  99, 140, 128, 192, 247, 112,   7
};

unsigned char Alogtable[256] = {
  1,   3,   5,  15,  17,  51,  85, 255,  26,  46, 114, 150, 161, 248,  19,  53,
 95, 225,  56,  72, 216, 115, 149, 164, 247,   2,   6,  10,  30,  34, 102, 170,
229,  52,  92, 228,  55,  89, 235,  38, 106, 190, 217, 112, 144, 171, 230,  49,
 83, 245,   4,  12,  20,  60,  68, 204,  79, 209, 104, 184, 211, 110, 178, 205,
 76, 212, 103, 169, 224,  59,  77, 215,  98, 166, 241,   8,  24,  40, 120, 136,
131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206,  73, 219, 118, 154,
181, 196,  87, 249,  16,  48,  80, 240,  11,  29,  39, 105, 187, 214,  97, 163,
254,  25,  43, 125, 135, 146, 173, 236,  47, 113, 147, 174, 233,  32,  96, 160,
251,  22,  58,  78, 210, 109, 183, 194,  93, 231,  50,  86, 250,  21,  63,  65,
195,  94, 226,  61,  71, 201,  64, 192,  91, 237,  44, 116, 156, 191, 218, 117,
159, 186, 213, 100, 172, 239,  42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
155, 182, 193,  88, 232,  35, 101, 175, 234,  37, 111, 177, 200,  67, 197,  84,
252,  31,  33,  99, 165, 244,   7,   9,  27,  45, 119, 153, 176, 203,  70, 202,
 69, 207,  74, 222, 121, 139, 134, 145, 168, 227,  62,  66, 198,  81, 243,  14,
 18,  54,  90, 238,  41, 123, 141, 140, 143, 138, 133, 148, 167, 242,  13,  23,
 57,  75, 221, 124, 132, 151, 162, 253,  28,  36, 108, 180, 199,  82, 246,   1
};

/* "Generated" S-box, the same as in Whirlpool report, May 23, 2003. It is
used in the function B
*/
unsigned char SubsB[256] = {
0x18,0x23,0xc6,0xE8,0x87,0xB8,0x01,0x4F,
0x36,0xA6,0xd2,0xF5,0x79,0x6F,0x91,0x52,
0x60,0xBc,0x9B,0x8E,0xA3,0x0c,0x7B,0x35,
0x1d,0xE0,0xd7,0xc2,0x2E,0x4B,0xFE,0x57,
0x15,0x77,0x37,0xE5,0x9F,0xF0,0x4A,0xdA,
0x58,0xc9,0x29,0x0A,0xB1,0xA0,0x6B,0x85,
0xBd,0x5d,0x10,0xF4,0xcB,0x3E,0x05,0x67,
0xE4,0x27,0x41,0x8B,0xA7,0x7d,0x95,0xd8,
0xFB,0xEE,0x7c,0x66,0xdd,0x17,0x47,0x9E,
0xcA,0x2d,0xBF,0x07,0xAd,0x5A,0x83,0x33,
0x63,0x02,0xAA,0x71,0xc8,0x19,0x49,0xd9,
0xF2,0xE3,0x5B,0x88,0x9A,0x26,0x32,0xB0,
0xE9,0x0F,0xd5,0x80,0xBE,0xcd,0x34,0x48,
0xFF,0x7A,0x90,0x5F,0x20,0x68,0x1A,0xAE,
```

```
0xB4,0x54,0x93,0x22,0x64,0xF1,0x73,0x12,
0x40,0x08,0xc3,0xEc,0xdB,0xA1,0x8d,0x3d,
0x97,0x00,0xcF,0x2B,0x76,0x82,0xd6,0x1B,
0xB5,0xAF,0x6A,0x50,0x45,0xF3,0x30,0xEF,
0x3F,0x55,0xA2,0xEA,0x65,0xBA,0x2F,0xc0,
0xdE,0x1c,0xFd,0x4d,0x92,0x75,0x06,0x8A,
0xB2,0xE6,0x0E,0x1F,0x62,0xd4,0xA8,0x96,
0xF9,0xc5,0x25,0x59,0x84,0x72,0x39,0x4c,
0x5E,0x78,0x38,0x8c,0xd1,0xA5,0xE2,0x61,
0xB3,0x21,0x9c,0x1E,0x43,0xc7,0xFc,0x04,
0x51,0x99,0x6d,0x0d,0xFA,0xdF,0x7E,0x24,
0x3B,0xAB,0xcE,0x11,0x8F,0x4E,0xB7,0xEB,
0x3c,0x81,0x94,0xF7,0xB9,0x13,0x2c,0xd3,
0xE7,0x6E,0xc4,0x03,0x56,0x44,0x7F,0xA9,
0x2A,0xBB,0xc1,0x53,0xdc,0x0B,0x9d,0x6c,
0x31,0x74,0xF6,0x46,0xAc,0x89,0x14,0xE1,
0x16,0x3A,0x69,0x09,0x70,0xB6,0xd0,0xEd,
0xcc,0x42,0x98,0xA4,0x28,0x5c,0xF8,0x86
};

/* "Pseudorandom" S-box, the same as in Whirlpool report, September 3, 2000.
It is used in the function F.
*/
unsigned char SubsF[256] = {
0x68, 0xd0, 0xeb, 0x2b, 0x48, 0x9d, 0x6a, 0xe4,
0xe3, 0xa3, 0x56, 0x81, 0x7d, 0xf1, 0x85, 0x9e,
0x2c, 0x8e, 0x78, 0xca, 0x17, 0xa9, 0x61, 0xd5,
0x5d, 0x0b, 0x8c, 0x3c, 0x77, 0x51, 0x22, 0x42,
0x3f, 0x54, 0x41, 0x80, 0xcc, 0x86, 0xb3, 0x18,
0x2e, 0x57, 0x06, 0x62, 0xf4, 0x36, 0xd1, 0x6b,
0x1b, 0x65, 0x75, 0x10, 0xda, 0x49, 0x26, 0xf9,
0xcb, 0x66, 0xe7, 0xba, 0xae, 0x50, 0x52, 0xab,
0x05, 0xf0, 0x0d, 0x73, 0x3b, 0x04, 0x20, 0xfe,
0xdd, 0xf5, 0xb4, 0x5f, 0x0a, 0xb5, 0xc0, 0xa0,
0x71, 0xa5, 0x2d, 0x60, 0x72, 0x93, 0x39, 0x08,
0x83, 0x21, 0x5c, 0x87, 0xb1, 0xe0, 0x00, 0xc3,
0x12, 0x91, 0x8a, 0x02, 0x1c, 0xe6, 0x45, 0xc2,
0xc4, 0xfd, 0xbf, 0x44, 0xa1, 0x4c, 0x33, 0xc5,
0x84, 0x23, 0x7c, 0xb0, 0x25, 0x15, 0x35, 0x69,
0xff, 0x94, 0x4d, 0x70, 0xa2, 0xaf, 0xcd, 0xd6,
0x6c, 0xb7, 0xf8, 0x09, 0xf3, 0x67, 0xa4, 0xea,
0xec, 0xb6, 0xd4, 0xd2, 0x14, 0x1e, 0xe1, 0x24,
0x38, 0xc6, 0xdb, 0x4b, 0x7a, 0x3a, 0xde, 0x5e,
0xdf, 0x95, 0xfc, 0xaa, 0xd7, 0xce, 0x07, 0x0f,
0x3d, 0x58, 0x9a, 0x98, 0x9c, 0xf2, 0xa7, 0x11,
0x7e, 0x8b, 0x43, 0x03, 0xe2, 0xdc, 0xe5, 0xb2,
0x4e, 0xc7, 0x6d, 0xe9, 0x27, 0x40, 0xd8, 0x37,
0x92, 0x8f, 0x01, 0x1d, 0x53, 0x3e, 0x59, 0xc1,
0x4f, 0x32, 0x16, 0xfa, 0x74, 0xfb, 0x63, 0x9f,
0x34, 0x1a, 0x2a, 0x5a, 0x8d, 0xc9, 0xcf, 0xf6,
0x90, 0x28, 0x88, 0x9b, 0x31, 0x0e, 0xbd, 0x4a,
0xe8, 0x96, 0xa6, 0x0c, 0xc8, 0x79, 0xbc, 0xbe,
0xef, 0x6e, 0x46, 0x97, 0x5b, 0xed, 0x19, 0xd9,
0xac, 0x99, 0xa8, 0x29, 0x64, 0x1f, 0xad, 0x55,
0x13, 0xbb, 0xf7, 0x6f, 0xb9, 0x47, 0x2f, 0xee,
0xb8, 0x7b, 0x89, 0x30, 0xd3, 0x7f, 0x76, 0x82
};

// MDS matrix of the type 4x4, used in the function B
unsigned char MDS4x4[4][4] =
{
0x02, 0x03, 0x01, 0x01,
0x01, 0x02, 0x03, 0x01,
0x01, 0x01, 0x02, 0x03,
```

```
0x03, 0x01, 0x01, 0x02
};

// MDS matrix of the type 16x16, used in key expansion
unsigned char MDS16x16[r][r] =
{
0x4A,0x7B,0xBA,0xCF,0x84,0x8D,0xB7,0xC6,0x72,0x9F,0x24,0xB2,0x7A,0x40,0xB1,0xCD,
0x70,0x70,0xA8,0x4F,0x79,0x8B,0xBB,0x60,0xA1,0x38,0x99,0x99,0xF5,0xAA,0xFA,0xF3,
0x91,0x09,0xE8,0xD7,0xB2,0xDC,0x10,0xC0,0x69,0xCF,0xD2,0x6F,0x6F,0x56,0x5B,0x61,
0x17,0x99,0x94,0xCF,0xB4,0x4D,0x92,0x62,0x6E,0x9A,0x62,0xEA,0x0D,0x6B,0x29,0xEE,
0x54,0x1B,0xA9,0x49,0xF4,0x28,0x21,0x65,0xE4,0xD3,0x54,0x50,0xC9,0xCF,0xB1,0xB2,
0x80,0x4A,0x39,0xF2,0x62,0x16,0x72,0xB6,0x8C,0x06,0x57,0x5A,0xD0,0x22,0xAE,0x6B,
0x2B,0x34,0xBF,0xB4,0x3C,0x3C,0x9E,0xE8,0x0E,0x9D,0xCB,0x66,0x48,0xB1,0x91,0x35,
0x11,0xDF,0xE7,0x64,0xB2,0x64,0xAE,0x66,0x33,0x9F,0x47,0x85,0x80,0x7C,0x61,0xA1,
0x5A,0x3A,0xCD,0x6F,0x58,0xA3,0xD3,0x2C,0x73,0xAC,0x22,0xA9,0xEE,0xEC,0xEC,0x7D,
0x0C,0x2C,0x83,0x77,0x1B,0x4C,0xAC,0x79,0xA9,0x83,0xC8,0xE8,0xA7,0x87,0xD0,0xAD,
0x8D,0xA2,0x42,0xDD,0x5D,0x4D,0xB7,0xB7,0x71,0x93,0x93,0xE6,0xCE,0x72,0xEF,0x65,
0xE6,0xA7,0x61,0xCA,0x05,0x70,0xB8,0xD4,0x11,0x86,0xE2,0x6D,0x61,0x93,0x11,0x09,
0xFA,0xD3,0xBD,0xD0,0xE8,0x11,0x5B,0x61,0xF8,0xEC,0x6A,0x86,0xD0,0x36,0x4A,0xD2,
0x08,0xF5,0xC5,0x15,0xD8,0xE2,0x55,0xEC,0x30,0x63,0x74,0x7E,0x2A,0xC2,0x6C,0x72,
0xF4,0x6B,0xC4,0xAB,0x68,0x40,0x09,0xC0,0x96,0x62,0xC6,0x86,0x6E,0x9F,0x7E,0x2B,
0x34,0xF0,0x19,0x66,0x6A,0x6D,0x73,0x08,0x22,0x16,0x11,0x9B,0x33,0xF4,0x5D,0xE2
};

// Permutations
unsigned char SMLPerm[r][c] =
{
23,14,49,32,41, 8,50,18,46,16,15,57,55,27,43, 2,
60, 7,22,42,38,26,53,12, 9,62,37,28, 0,36,51,20,
17,39, 4,56,59, 3,47,31, 6,25,45,48,24,58,11,33,
29,13,40,61, 1,19,63,34,52,35, 5,30,44,54,10,21,

17,42,57, 6,62, 8,24,12, 3,21,55,51,44,34,39,31,
36, 2,25,58, 7,47,53,14,49, 9,16,30,33,60,22,40,
41,37,50,15, 1,45,19,63,35,10,59,52,27,20, 4,28,
13,56,23,46,48,32,26,18,61,43,29,54, 5,11, 0,38,

10,15, 4, 1, 5, 0,14,11, 2, 8, 7,13, 6, 9, 3,12,
26,31,20,17,21,16,30,27,18,24,23,29,22,25,19,28,
42,47,36,33,37,32,46,43,34,40,39,45,38,41,35,44,
58,63,52,49,53,48,62,59,50,56,55,61,54,57,51,60,

10, 5,12, 2, 7, 9, 0,15, 1,11, 4,14, 8, 3,13, 6,
26,21,28,18,23,25,16,31,17,27,20,30,24,19,29,22,
42,37,44,34,39,41,32,47,33,43,36,46,40,35,45,38,
58,53,60,50,55,57,48,63,49,59,52,62,56,51,61,54,



23,14,49,32,41, 8,50,18,46,16,15,57,55,27,43, 2,
60, 7,22,42,38,26,53,12, 9,62,37,28, 0,36,51,20,
17,39, 4,56,59, 3,47,31, 6,25,45,48,24,58,11,33,
29,13,40,61, 1,19,63,34,52,35, 5,30,44,54,10,21,

17,42,57, 6,62, 8,24,12, 3,21,55,51,44,34,39,31,
36, 2,25,58, 7,47,53,14,49, 9,16,30,33,60,22,40,
41,37,50,15, 1,45,19,63,35,10,59,52,27,20, 4,28,
13,56,23,46,48,32,26,18,61,43,29,54, 5,11, 0,38,

10,15, 4, 1, 5, 0,14,11, 2, 8, 7,13, 6, 9, 3,12,
26,31,20,17,21,16,30,27,18,24,23,29,22,25,19,28,
42,47,36,33,37,32,46,43,34,40,39,45,38,41,35,44,
58,63,52,49,53,48,62,59,50,56,55,61,54,57,51,60,

10, 5,12, 2, 7, 9, 0,15, 1,11, 4,14, 8, 3,13, 6,
26,21,28,18,23,25,16,31,17,27,20,30,24,19,29,22,
```

```
42,37,44,34,39,41,32,47,33,43,36,46,40,35,45,38,
58,53,60,50,55,57,48,63,49,59,52,62,56,51,61,54,



23,14,49,32,41, 8,50,18,46,16,15,57,55,27,43, 2,
60, 7,22,42,38,26,53,12, 9,62,37,28, 0,36,51,20,
17,39, 4,56,59, 3,47,31, 6,25,45,48,24,58,11,33,
29,13,40,61, 1,19,63,34,52,35, 5,30,44,54,10,21,

17,42,57, 6,62, 8,24,12, 3,21,55,51,44,34,39,31,
36, 2,25,58, 7,47,53,14,49, 9,16,30,33,60,22,40,
41,37,50,15, 1,45,19,63,35,10,59,52,27,20, 4,28,
13,56,23,46,48,32,26,18,61,43,29,54, 5,11, 0,38,

10,15, 4, 1, 5, 0,14,11, 2, 8, 7,13, 6, 9, 3,12,
26,31,20,17,21,16,30,27,18,24,23,29,22,25,19,28,
42,47,36,33,37,32,46,43,34,40,39,45,38,41,35,44,
58,63,52,49,53,48,62,59,50,56,55,61,54,57,51,60,

10, 5,12, 2, 7, 9, 0,15, 1,11, 4,14, 8, 3,13, 6,
26,21,28,18,23,25,16,31,17,27,20,30,24,19,29,22,
42,37,44,34,39,41,32,47,33,43,36,46,40,35,45,38,
58,53,60,50,55,57,48,63,49,59,52,62,56,51,61,54,



23,14,49,32,41, 8,50,18,46,16,15,57,55,27,43, 2,
60, 7,22,42,38,26,53,12, 9,62,37,28, 0,36,51,20,
17,39, 4,56,59, 3,47,31, 6,25,45,48,24,58,11,33,
29,13,40,61, 1,19,63,34,52,35, 5,30,44,54,10,21,

17,42,57, 6,62, 8,24,12, 3,21,55,51,44,34,39,31,
36, 2,25,58, 7,47,53,14,49, 9,16,30,33,60,22,40,
41,37,50,15, 1,45,19,63,35,10,59,52,27,20, 4,28,
13,56,23,46,48,32,26,18,61,43,29,54, 5,11, 0,38,

10,15, 4, 1, 5, 0,14,11, 2, 8, 7,13, 6, 9, 3,12,
26,31,20,17,21,16,30,27,18,24,23,29,22,25,19,28,
42,47,36,33,37,32,46,43,34,40,39,45,38,41,35,44,
58,63,52,49,53,48,62,59,50,56,55,61,54,57,51,60,

10, 5,12, 2, 7, 9, 0,15, 1,11, 4,14, 8, 3,13, 6,
26,21,28,18,23,25,16,31,17,27,20,30,24,19,29,22,
42,37,44,34,39,41,32,47,33,43,36,46,40,35,45,38,
58,53,60,50,55,57,48,63,49,59,52,62,56,51,61,54
};

//IV
unsigned char IV_HDN[c] =
{
 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63
};

//CONST0
unsigned char CONST0[c] =
{
128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191
```

```
};

//CONST1
unsigned char CONST1[c] =
{
 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94,
96, 98,100,102,104,106,108,110,112,114,116,118,120,122,124,126
};

// Constants in F
unsigned long RConstF[9][64] =
{
0x7AAF7106, 0xF63AD7F5, 0x738635A4, 0xEF119A13,
0x689CF8C2, 0xE4685EB1, 0x61FBA360, 0xDD4701DF,
0x56D2678E, 0xD25DC47D, 0x4E292A2C, 0xCBB4889B,
0x4707ED4A, 0xC0937339, 0x3C1ED1E8, 0xB9EA3647,
0x35759436, 0xAEC0FAE5, 0x2A4C5F54, 0xA7DFBD03,
0x23AB03F2, 0x9F3661A1, 0x1881C610, 0x940D24CF,
0x11988ABE, 0x8D6BEF6D, 0x06F74DDC, 0x8242D38B,
0xFFCE307A, 0x7B599629, 0xF724F498, 0x70B05977,
0xEC03BF26, 0x698F1D95, 0xE51A6244, 0x5EE5C033,
0xDA7126E2, 0x57FC8B51, 0xD34FE900, 0x4CDB4FFF,
0xC8A6ADAE, 0x4432321D, 0xC1BD90CC, 0x3D08F6BB,
0xB6945B6A, 0x3267B9D9, 0xAFF31F88, 0x2B7E7C67,
0xA4C9C2D6, 0x20552085, 0x9C208574, 0x19B3EB23,
0x953F4992, 0x0E8AAE41, 0x8A160C30, 0x07E192EF,
0x836CF75E, 0xFCF8550D, 0x784BBBFC, 0xF5D719AB,
0x71A27E1A, 0xED2DDCC9, 0x66B922B8, 0xE2048717,

0x79CB44FF, 0xF55EE20C, 0x70E2005D, 0xEC75AFEA,
0x6BF8CD3B, 0xE70C6B48, 0x629F9699, 0xDE233426,
0x55B65277, 0xD139F184, 0x4D4D1FD5, 0xC8D0BD62,
0x4463D8B3, 0xC3F746C0, 0x3F7AE411, 0xBA8E03BE,
0x3611A1CF, 0xADA4CF1C, 0x29286AAD, 0xA4BB88FA,
0x20CF360B, 0x9C525458, 0x1BE5F3E9, 0x97691136,
0x12FCBF47, 0x8E0FDA94, 0x05937825, 0x8126E672,
0xFCAA0583, 0x783DA3D0, 0xF440C161, 0x73D46C8E,
0xEF678ADF, 0x6AEB286C, 0xE67E57BD, 0x5D81F5CA,
0xD915131B, 0x5498BEA8, 0xD02BDCF9, 0x4FBF7A06,
0xCBC29857, 0x475607E4, 0xC2D9A535, 0x3E6CC342,
0xB5F06E93, 0x31038C20, 0xAC972A71, 0x281A499E,
0xA7ADF72F, 0x2331157C, 0x9F44B08D, 0x1AD7DEDA,
0x965B7C6B, 0x0DEE9BB8, 0x897239C9, 0x0485A716,
0x8008C2A7, 0xFF9C60F4, 0x7B2F8E05, 0xF6B32C52,
0x72C64BE3, 0xEE49E930, 0x65DD1741, 0xE160B2EE,

0x78E75854, 0xF472FEA7, 0x71CE1CF6, 0xED59B341,
0x6AD4D190, 0xE62077E3, 0x63B38A32, 0xDF0F288D,
0x549A4EDC, 0xD015ED2F, 0x4C61037E, 0xC9FCA1C9,
0x454FC418, 0xC2DB5A6B, 0x3E56F8BA, 0xBBA21F15,
0x373DBD64, 0xAC88D3B7, 0x28047606, 0xA5979451,
0x21E32AA0, 0x9D7E48F3, 0x1AC9EF42, 0x96450D9D,
0x13D0A3EC, 0x8F23C63F, 0x04BF648E, 0x800AFAD9,
0xFD861928, 0x7911BF7B, 0xF56CDDCA, 0x72F87025,
0xEE4B9674, 0x6BC734C7, 0xE7524B16, 0x5CADE961,
0xD8390FB0, 0x55B4A203, 0xD107C052, 0x4E9366AD,
0xCAEE84FC, 0x467A1B4F, 0xC3F5B99E, 0x3F40DFE9,
0xB4DC7238, 0x302F908B, 0xADBB36DA, 0x29365535,
0xA681EB84, 0x221D09D7, 0x9E68AC26, 0x1BFBC271,
0x977760C0, 0x0CC28713, 0x885E2562, 0x05A9BBBD,
0x8124DE0C, 0xFEB07C5F, 0x7A0392AE, 0xF79F30F9,
0x73EA5748, 0xEF65F59B, 0x64F10BEA, 0xE04CAE45,
```

```
0x7F032F0D, 0xF39689FE, 0x762A6BAF, 0xEABDC418,
0x6D30A6C9, 0xE1C400BA, 0x6457FD6B, 0xD8EB5FD4,
0x537E3985, 0xD7F19A76, 0x4B857427, 0xCE18D690,
0x42ABB341, 0xC53F2D32, 0x39B28FE3, 0xBC46684C,
0x30D9CA3D, 0xAB6CA4EE, 0x2FE0015F, 0xA273E308,
0x26075DF9, 0x9A9A3FAA, 0x1D2D981B, 0x91A17AC4,
0x1434D4B5, 0x88C7B166, 0x035B13D7, 0x87EE8D80,
0xFA626E71, 0x7EF5C822, 0xF288AA93, 0x751C077C,
0xE9AFE12D, 0x6C23439E, 0xE0B63C4F, 0x5B499E38,
0xDFDD78E9, 0x5250D55A, 0xD6E3B70B, 0x497711F4,
0xCD0AF3A5, 0x419E6C16, 0xC411CEC7, 0x38A4A8B0,
0xB3380561, 0x37CBE7D2, 0xAA5F4183, 0x2ED2226C,
0xA1659CDD, 0x25F97E8E, 0x998CDB7F, 0x1C1FB528,
0x90931799, 0x0B26F04A, 0x8FBA523B, 0x024DCCE4,
0x86C0A955, 0xF9540B06, 0x7DE7E5F7, 0xF07B47A0,
0x740E2011, 0xE88182C2, 0x63157CB3, 0xE7A8D91C,

0x7E3F02E2, 0xF2AAA411, 0x77164640, 0xEB81E9F7,
0x6C0C8B26, 0xE0F82D55, 0x656BD084, 0xD9D7723B,
0x5242146A, 0xD6CDB799, 0x4AB959C8, 0xCF24FB7F,
0x43979EAE, 0xC40300DD, 0x388EA20C, 0xBD7A45A3,
0x31E5E7D2, 0xAA508901, 0x2EDC2CB0, 0xA34FCEE7,
0x273B7016, 0x9BA61245, 0x1C11B5F4, 0x909D572B,
0x1508F95A, 0x89FB9C89, 0x02673E38, 0x86D2A06F,
0xFB5E439E, 0x7FC9E5CD, 0xF3B4877C, 0x74202A93,
0xE893CCC2, 0x6D1F6E71, 0xE18A11A0, 0x5A75B3D7,
0xDEE15506, 0x536CF8B5, 0xD7DF9AE4, 0x484B3C1B,
0xCC36DE4A, 0x40A241F9, 0xC52DE328, 0x3998855F,
0xB204288E, 0x36F7CA3D, 0xAB636C6C, 0x2FEE0F83,
0xA059B132, 0x24C55361, 0x98B0F690, 0x1D2398C7,
0x91AF3A76, 0x0A1ADDA5, 0x8E867FD4, 0x0371E10B,
0x87FC84BA, 0xF86826E9, 0x7CDBC818, 0xF1476A4F,
0x75320DFE, 0xE9BDAF2D, 0x6229515C, 0xE694F4F3,

0x7D5B165B, 0xF1CEB0A8, 0x747252F9, 0xE8E5FD4E,
0x6F689F9F, 0xE39C39EC, 0x660FC43D, 0xDAB36682,
0x512600D3, 0xD5A9A320, 0x49DD4D71, 0xCC40EFC6,
0x40F38A17, 0xC7671464, 0x3BEAB6B5, 0xBE1E511A,
0x3281F36B, 0xA9349DB8, 0x2DB83809, 0xA02BDA5E,
0x245F64AF, 0x98C206FC, 0x1F75A14D, 0x93F94392,
0x166CEDE3, 0x8A9F8830, 0x01032A81, 0x85B6B4D6,
0xF83A5727, 0x7CADF174, 0xF0D093C5, 0x77443E2A,
0xEBF7D87B, 0x6E7B7AC8, 0xE2EE0519, 0x5911A76E,
0xDD8541BF, 0x5008EC0C, 0xD4BB8E5D, 0x4B2F28A2,
0xCF52CAF3, 0x43C65540, 0xC649F791, 0x3AFC91E6,
0xB1603C37, 0x3593DE84, 0xA80778D5, 0x2C8A1B3A,
0xA33DA58B, 0x27A147D8, 0x9BD4E229, 0x1E478C7E,
0x92CB2ECF, 0x097EC91C, 0x8DE26B6D, 0x0015F5B2,
0x84989003, 0xFB0C3250, 0x7FBFDCA1, 0xF2237EF6,
0x76561947, 0xEAD9BB94, 0x614D45E5, 0xE5F0E04A,

0x7C77E530, 0xF0E243C3, 0x755EA192, 0xE9C90E25,
0x6E446CF4, 0xE2B0CA87, 0x67233756, 0xDB9F95E9,
0x500AF3B8, 0xD485504B, 0x48F1BE1A, 0xCD6C1CAD,
0x41DF797C, 0xC64BE70F, 0x3AC645DE, 0xBF32A271,
0x33AD0000, 0xA8186ED3, 0x2C94CB62, 0xA1072935,
0x257397C4, 0x99EEF597, 0x1E595226, 0x92D5B0F9,
0x17401E88, 0x8BB37B5B, 0x002FD9EA, 0x849A47BD,
0xF916A44C, 0x7D81021F, 0xF1FC60AE, 0x7668CD41,
0xEADB2B10, 0x6F5789A3, 0xE3C2F672, 0x583D5405,
0xDCA9B2D4, 0x51241F67, 0xD5977D36, 0x4A03DBC9,
0xCE7E3998, 0x42EAA62B, 0xC76504FA, 0x3BD0628D,
0xB04CCF5C, 0x34BF2DEF, 0xA92B8BBE, 0x2DA6E851,
```

```
0xA21156E0, 0x268DB4B3, 0x9AF81142, 0x1F6B7F15,
0x93E7DDA4, 0x08523A77, 0x8CCE9806, 0x013906D9,
0x85B46368, 0xFA20C13B, 0x7E932FCA, 0xF30F8D9D,
0x777AEA2C, 0xEBF548FF, 0x6061B68E, 0xE4DC1321,

0x7293F8E9, 0xFE065E1A, 0x7BBABC4B, 0xE72D13FC,
0x60A0712D, 0xEC54D75E, 0x69C72A8F, 0xD57B8830,
0x5EEEEE61, 0xDA614D92, 0x4615A3C3, 0xC3880174,
0x4F3B64A5, 0xC8AFFAD6, 0x34225807, 0xB1D6BFA8,
0x3D491DD9, 0xA6FC730A, 0x2270D6BB, 0xAFE334EC,
0x2B978A1D, 0x970AE84E, 0x10BD4FFF, 0x9C31AD20,
0x19A40351, 0x85576682, 0x0ECBC433, 0x8A7E5A64,
0xF7F2B995, 0x73651FC6, 0xFF187D77, 0x788CD098,
0xE43F36C9, 0x61B3947A, 0xED26EBAB, 0x56D949DC,
0xD24DAF0D, 0x5FC002BE, 0xDB7360EF, 0x44E7C610,
0xC09A2441, 0x4C0EBBF2, 0xC9811923, 0x35347F54,
0xBEA8D285, 0x3A5B3036, 0xA7CF9667, 0x2342F588,
0xACF54B39, 0x2869A96A, 0x941C0C9B, 0x118F62CC,
0x9D03C07D, 0x06B627AE, 0x822A85DF, 0x0FDD1B00,
0x8B507EB1, 0xF4C4DCE2, 0x70773213, 0xFDEB9044,
0x799EF7F5, 0xE5115526, 0x6E85AB57, 0xEA380EF8,

0x71CFCC5E, 0xFD5A6AAD, 0x78E688FC, 0xE471274B,
0x63FC459A, 0xEF08E3E9, 0x6A9B1E38, 0xD627BC87,
0x5DB2DAD6, 0xD93D7925, 0x45499774, 0xC0D435C3,
0x4C675012, 0xCBF3CE61, 0x377E6CB0, 0xB28A8B1F,
0x3E15296E, 0xA5A047BD, 0x212CE20C, 0xACBF005B,
0x28CBBEAA, 0x9456DCF9, 0x13E17B48, 0x9F6D9997,
0x1AF837E6, 0x860B5235, 0x0D97F084, 0x89226ED3,
0xF4AE8D22, 0x70392B71, 0xFC4449C0, 0x7BD0E42F,
0xE763027E, 0x62EFA0CD, 0xEE7ADF1C, 0x55857D6B,
0xD1119BBA, 0x5C9C3609, 0xD82F5458, 0x47BBF2A7,
0xC3C610F6, 0x4F528F45, 0xCADD2D94, 0x36684BE3,
0xBDF4E632, 0x39070481, 0xA493A2D0, 0x201EC13F,
0xAFA97F8E, 0x2B359DDD, 0x9740382C, 0x12D3567B,
0x9E5FF4CA, 0x05EA1319, 0x8176B168, 0x0C812FB7,
0x880C4A06, 0xF798E855, 0x732B06A4, 0xFEB7A4F3,
0x7AC2C342, 0xE64D6191, 0x6DD99FE0, 0xE9643A4F
};

// Constants in B
unsigned long RConstB[10][16] =
{
0x24687531, 0x48D0EA62, 0x6D395F93, 0x91A1D4C4,
0xB60A49F5, 0xDA72BF26, 0xFEDB3457, 0x2343A988,
0x47AC1EB9, 0x6C1493EA, 0x907D091B, 0xB4E57E4C,
0xD94DF37D, 0xFDB668AE, 0x221EDDDF, 0x46875310,

0x6AEFC841, 0x8F583D72, 0xB3C0B2A3, 0xD82927D4,
0xFC919D05, 0x20FA1236, 0x45628767, 0x69CAFC98,
0x8E3371C9, 0xB29BE6FA, 0xD7045C2B, 0xFB6CD15C,
0x1FD5468D, 0x443DBBBE, 0x68A630EF, 0x8D0EA620,

0xB1771B51, 0xD5DF9082, 0xFA4805B3, 0x1EB07AE4,
0x4318F015, 0x67816546, 0x8BE9DA77, 0xB0524FA8,
0xD4BAC4D9, 0xF9233A0A, 0x1D8BAF3B, 0x41F4246C,
0x665C999D, 0x8AC50ECE, 0xAF2D83FF, 0xD395F930,

0xF7FE6E61, 0x1C66E392, 0x40CF58C3, 0x6537CDF4,
0x89A04325, 0xAE08B856, 0xD2712D87, 0xF6D9A2B8,
0x1B4217E9, 0x3FAA8D1A, 0x6413024B, 0x887B777C,
0xACE3ECAD, 0xD14C61DE, 0xF5B4D70F, 0x1A1D4C40,

0x3E85C171, 0x62EE36A2, 0x8756ABD3, 0xABBF2104,
```

```
0xD0279635, 0xF4900B66, 0x18F88097, 0x3D60F5C8,
0x61C96AF9, 0x8631E02A, 0xAA9A555B, 0xCF02CA8C,
0xF36B3FBD, 0x17D3B4EE, 0x3C3C2A1F, 0x60A49F50,

0x850D1481, 0xA97589B2, 0xCDDDFEE3, 0xF2467414,
0x16AEE945, 0x3B175E76, 0x5F7FD3A7, 0x83E848D8,
0xA850BE09, 0xCCB9333A, 0xF121A86B, 0x158A1D9C,
0x39F292CD, 0x5E5B07FE, 0x82C37D2F, 0xA72BF260,

0xCB946791, 0xEFFCDCC2, 0x146551F3, 0x38CDC724,
0x5D363C55, 0x819EB186, 0xA60726B7, 0xCA6F9BE8,
0xEED81119, 0x1340864A, 0x37A8FB7B, 0x5C1170AC,
0x8079E5DD, 0xA4E25B0E, 0xC94AD03F, 0xEDB34570,

0x121BBAA1, 0x36842FD2, 0x5AECA503, 0x7F551A34,
0xA3BD8F65, 0xC8260496, 0xEC8E79C7, 0x10F6EEF8,
0x355F6429, 0x59C7D95A, 0x7E304E8B, 0xA298C3BC,
0xC70138ED, 0xEB69AE1E, 0x0FD2234F, 0x343A9880,

0x58A30DB1, 0x7D0B82E2, 0xA173F813, 0xC5DC6D44,
0xEA44E275, 0x0EAD57A6, 0x3315CCD7, 0x577E4208,
0x7BE6B739, 0xA04F2C6A, 0xC4B7A19B, 0xE92016CC,
0x0D888BFD, 0x31F1012E, 0x5659765F, 0x7AC1EB90,

0x9F2A60C1, 0xC392D5F2, 0xE7FB4B23, 0x0C63C054,
0x30CC3585, 0x5534AAB6, 0x799D1FE7, 0x9E059518,
0xC26E0A49, 0xE6D67F7A, 0x0B3EF4AB, 0x2FA769DC,
0x540FDF0D, 0x7878543E, 0x9CE0C96F, 0xC1493EA0
};

/* Tables N[4][256], T[4][16][256] for multiplications MDS4x4_multiply and
MDS16x16_multiply are not listed here (due to the space). Instead, they are
created dynamically by functions Init_MDS4x4_tables and Init_MDS16x16_tables.
*/
unsigned long N[4][256], T[4][16][256];
```

12.3. Module dn.c

```
/* dn.c */
/* Unoptimized implementation of the transformation DN. */

#include "dn.h"
#include "dn_constants.h"

/*==============================================================*/
void Copy64(unsigned char* in,unsigned char* out)
{
	int i;
	for(i=0;i<c;i++) out[i] = in[i];
}
/*==============================================================*/
/* Multiplication in GF(2^8)*/
unsigned char mul(unsigned char a, unsigned char b)
{
	if (a && b)
		return Alogtable[(Logtable[a] + Logtable[b])%255];
	else
		return 0;
}
/*==============================================================*/
/* Inversion in GF(2^8)*/
unsigned char Inv(unsigned char a)
```

```
{
      if (a)
            return Alogtable[255 - Logtable[a]];
      else
            return 0;
}
/*=============================================================*/
/* Definition and print of constants in F and B*/
int Check_Const()
{
      unsigned char i, j;
      unsigned  long  CONSTA  =  0xfedc1357,  CONSTB  =  0x84736251,  CONSTC  =
0x24687531;
      int flag = 0;

      // F
      for(i = 0; i <  9; i++)   for(j = 0; j < 64; j++)
      if ( RConstF[i][j] != ( (CONSTA * (i+1)) ^ (CONSTB * (j+1)) )) flag = -
1;
      //  B
      for(i = 0; i < 10; i++)   for(j = 0; j < 16; j++)
                  if (RConstB[i][j] != CONSTC*(16*i+j+1) ) flag = -1;
      return(flag);
}
/*=============================================================*/
/* Definition of the matrix G 16x32*/
int Check_Matrix()
{
      unsigned char i, j, k, i1,j1, inv, t, flag,nasobek, a[32];
      unsigned                                                          char
      TF[16][16],F[16][16],G[16][32],G1[16][16],G2[16][16],Temp[16][16];

      a[0] = 1; // fixed
      // arbitrary different elements, different from a[0], 12..42
      for(j = 1; j < 32; j++) a[j] = j + 11;

      for(j = 0; j < 32; j++) G[0][j] = 1; //zero power
      for(j = 0; j < 32; j++)
            for(i = 1; i < 16; i++) // 1st, 2nd, ...,15th power
                  G[i][j] = mul(G[i-1][j],a[j]);

      for(i = 0; i < 16; i++) for(j = 0; j < 16; j++)
      {
            G1[i][j]=G[i][j];
            G2[i][j]=G[i][j+16];
      }

      //transformation of (G1,G2) to (I, F)
      for(i = 0; i < 16; i++)
      {
            if(G[i][i] == 0) flag = 0;
                  else flag = 1;

            if(flag != 1)
            {
                  for(i1 = i+1; i1 < 16; i1++)
                  {
                    if( (flag == 0) && (G[i1][i] != 0) )//exch. rows (i,i1)
                         {
                           flag=1;
                           for(j1 = 0; j1 < 32; j1++)
                           {t = G[i1][j1];
                                G[i1][j1] = G[i][j1]; G[i][j1] = t;
                           }
```

```
                        }
                }
            }
            if(flag == 0) return(-1);

            inv = Inv(G[i][i]);
            for(j = 0; j < 32; j++) G[i][j] = mul(G[i][j],inv);
            if(G[i][i] != 1) return(-1);
            for(i1 = 0; i1 < 16; i1++)
            {
                    nasobek = G[i1][i];
                    if( (i1 != i) && (nasobek != 0))
                    {
                            for(j1 = 0; j1 < 32; j1++)
                            G[i1][j1] = G[i1][j1] ^ mul(nasobek,G[i][j1]);
                    }
            }

    }
    for(i = 0; i < 16; i++) for(j = 0; j < 16; j++)      F[i][j]=G[i][j+16];

    // check of F
    // G2 = G1 * F
    for(i = 0; i < 16; i++) for(j = 0; j < 16; j++)
    {
            Temp[i][j]=G2[i][j];
            for(k = 0; k < 16; k++)   Temp[i][j] ^= mul(G1[i][k],F[k][j]);
    }

    flag = 0;
    for(i = 0; i < 16; i++)
    {
            for(j = 0; j < 16; j++)
            {
                    if(Temp[i][j] != 0) flag = 1;
            }
    }
    if (flag == 1) return(-1);

    // We will use transposed matrix F
    for(i = 0; i < 16; i++) for(j = 0; j < 16; j++) TF[i][j] = F[j][i];

    //check of MDS16x16
    for(i = 0; i < 16; i++)
    {
            for(j = 0; j < 16; j++)
            {
                    if(TF[i][j] != MDS16x16[i][j]) flag = 1;
            }
    }
    if (flag == 1) return(-1);
    return(0);

}
/*============================================================*/
/* Prepares tables for the function MDS4x4_multiply */
void Init_MDS4x4_tables(void)
{
    unsigned long j,x;
    for(j = 0; j < 4; j++) for(x = 0; x < 256; x++)
            {
               N[j][x] =
                     mul(MDS4x4[0][j],SubsB[(unsigned char)(x)]) ^
                  ( mul(MDS4x4[1][j],SubsB[(unsigned char)(x)]) <<  8) ^
```

```
                    ( mul(MDS4x4[2][j],SubsB[(unsigned char)(x)]) << 16) ^
                    ( mul(MDS4x4[3][j],SubsB[(unsigned char)(x)]) << 24);
            }

}

/*============================================================*/
/* Prepares tables for the function MDS16x16_multiply */
void Init_MDS16x16_tables(void)
{
       unsigned long i,j,x;
       //memset(T, 0,sizeof(T));
       for(i = 0; i < 4; i++)    for(j = 0; j < r; j++)
       {
              for(x = 0; x < 256; x++)
              {
               T[i][j][x] = mul(MDS16x16[4*i+0][j],SubsF[(unsigned char)(x)]) ^
               ( mul(MDS16x16[4*i+1][j],SubsF[(unsigned char)(x)]) <<  8) ^
               ( mul(MDS16x16[4*i+2][j],SubsF[(unsigned char)(x)]) << 16) ^
               ( mul(MDS16x16[4*i+3][j],SubsF[(unsigned char)(x)]) << 24);
              }
       }
}

/*============================================================*/
/* ExpandRK creates the array of round keys RK[1..rho][0..r-1][0..c-1]
       from input array RK[0][0..r-1][0..c-1].
*/
void ExpandRK(unsigned char RK[MAXRHO][r][c],int rho,int print)
{
       unsigned char i,j,x,m, temp[c];
       unsigned long templong;
       int k;

       //compute RK
 for(i=1;i<rho;i++)
 {
  for(j=0;j<c;j++)
  {
       templong = 0; for(m = 0;m < r; m++) templong ^= T[0][m][RK[i-1][m][j]];
       RK[i][4*0+0][j] = (unsigned char)( templong       ) & 0xFF;
       RK[i][4*0+1][j] = (unsigned char)( templong >>   8) & 0xFF;
       RK[i][4*0+2][j] = (unsigned char)( templong >>  16) & 0xFF;
       RK[i][4*0+3][j] = (unsigned char)( templong >>  24) & 0xFF;

       templong = 0; for(m = 0;m < r; m++) templong ^= T[1][m][RK[i-1][m][j]];
       RK[i][4*1+0][j] = (unsigned char)( templong       ) & 0xFF;
       RK[i][4*1+1][j] = (unsigned char)( templong >>   8) & 0xFF;
       RK[i][4*1+2][j] = (unsigned char)( templong >>  16) & 0xFF;
       RK[i][4*1+3][j] = (unsigned char)( templong >>  24) & 0xFF;

       templong = 0; for(m = 0;m < r; m++) templong ^= T[2][m][RK[i-1][m][j]];
       RK[i][4*2+0][j] = (unsigned char)( templong       ) & 0xFF;
       RK[i][4*2+1][j] = (unsigned char)( templong >>   8) & 0xFF;
       RK[i][4*2+2][j] = (unsigned char)( templong >>  16) & 0xFF;
       RK[i][4*2+3][j] = (unsigned char)( templong >>  24) & 0xFF;

       templong = RConstF[i-1][j];
       for(m = 0;m < r; m++) templong ^= T[3][m][RK[i-1][m][j]];
       RK[i][4*3+0][j] = (unsigned char)( templong       ) & 0xFF;
       RK[i][4*3+1][j] = (unsigned char)( templong >>   8) & 0xFF;
       RK[i][4*3+2][j] = (unsigned char)( templong >>  16) & 0xFF;
       RK[i][4*3+3][j] = (unsigned char)( templong >>  24) & 0xFF;
```

```
   }
  }

   //Final permutation, unoptimized
   for(i=0;i<rho;i++)
   {
        x=2;
        for(k=48;  k<c; k++) temp[k-48]  = RK[i][x][k];
        for(k=c-1;k>=16;k--) RK[i][x][k] = RK[i][x][k-16];
        for(k=15;  k>= 0;k--) RK[i][x][k] = temp[k];
        x=3;
        for(k=0;k<32;k++)
        {temp[k] = RK[i][x][k];RK[i][x][k]= RK[i][x][k+32];
               RK[i][x][k+32]= temp[k];}
        x=4;
        for(k=0;k<32;k++)
        {temp[k] = RK[i][x][k];RK[i][x][k]= RK[i][x][k+32];
        RK[i][x][k+32]= temp[k];}
        x=5;
        for(k=0;k<32;k++)
        {temp[k] = RK[i][x][k];RK[i][x][k]= RK[i][x][k+32];
        RK[i][x][k+32]= temp[k];}
        x=6;
        for(k=48;  k<c; k++) temp[k-48]  = RK[i][x][k];
        for(k=c-1;k>=16;k--) RK[i][x][k] = RK[i][x][k-16];
        for(k=15;  k>= 0;k--) RK[i][x][k] = temp[k];

        x=10;
        for(k=48;  k<c; k++) temp[k-48]  = RK[i][x][k];
        for(k=c-1;k>=16;k--) RK[i][x][k] = RK[i][x][k-16];
        for(k=15;  k>= 0;k--) RK[i][x][k] = temp[k];
        x=11;
        for(k=0;k<32;k++)
        {temp[k]   =   RK[i][x][k];RK[i][x][k]=   RK[i][x][k+32];RK[i][x][k+32]=
temp[k];}
        x=12;
        for(k=0;k<32;k++)
        {temp[k] = RK[i][x][k];RK[i][x][k]= RK[i][x][k+32];
        RK[i][x][k+32]= temp[k];}
        x=13;
        for(k=0;k<32;k++)
        {temp[k] = RK[i][x][k];RK[i][x][k]= RK[i][x][k+32];
        RK[i][x][k+32]= temp[k];}
        x=14;
        for(k=48;  k<c; k++) temp[k-48]  = RK[i][x][k];
        for(k=c-1;k>=16;k--) RK[i][x][k] = RK[i][x][k-16];
        for(k=15;  k>= 0;k--) RK[i][x][k] = temp[k];
   }

}
/*==============================================================*/

/* Function DN.
The function DN consists of rho big rounds (0 .. rho-1).
Every big round consists of r (=16) small rounds (transformations T1).
The output from previous round is the input to the next round.
I-th (i = 0 ... rho-1) big round uses r round keys RK[i][0..r-1][0..c-1].
The input : c bytes in the array indata.
The output: c bytes in the array outdata.
 */
void DN(unsigned char RK[MAXRHO][r][c],
                    int rho,
                    unsigned char indata[c],
                    unsigned char outdata[c],
```

```
                    int print)
{
unsigned char tempdata[64],tempdata2[64];

int i,j;
unsigned char k;
unsigned long temp;

ExpandRK(RK,rho,print);

Copy64(indata,tempdata);
for(i=0;i<rho;i++)
{
  for(j=0;j<r;j+=2)
  {
    for(k=0;k<c;k+=4)
    {
      temp  = N[0][tempdata[SMLPerm[j][k+0]]];
      temp ^= N[1][tempdata[SMLPerm[j][k+1]]];
      temp ^= N[2][tempdata[SMLPerm[j][k+2]]];
      temp ^= N[3][tempdata[SMLPerm[j][k+3]]];
      if (k==60) temp = temp ^ RConstB[i][j];
      tempdata2[k+0] = RK[i][j][k+0] ^ (unsigned char)( temp       ) & 0xFF;
      tempdata2[k+1] = RK[i][j][k+1] ^ (unsigned char)( temp >>   8) & 0xFF;
      tempdata2[k+2] = RK[i][j][k+2] ^ (unsigned char)( temp >>  16) & 0xFF;
      tempdata2[k+3] = RK[i][j][k+3] ^ (unsigned char)( temp >>  24) & 0xFF;
    }

    for(k=0;k<c;k+=4)
    {
      temp  = N[0][tempdata2[SMLPerm[j+1][k+0]]];
      temp ^= N[1][tempdata2[SMLPerm[j+1][k+1]]];
      temp ^= N[2][tempdata2[SMLPerm[j+1][k+2]]];
      temp ^= N[3][tempdata2[SMLPerm[j+1][k+3]]];
      if (k==60) temp = temp ^ RConstB[i][j+1];
      tempdata[k+0] = RK[i][j+1][k+0] ^ (unsigned char)( temp       ) & 0xFF;
      tempdata[k+1] = RK[i][j+1][k+1] ^ (unsigned char)( temp >>   8) & 0xFF;
      tempdata[k+2] = RK[i][j+1][k+2] ^ (unsigned char)( temp >>  16) & 0xFF;
      tempdata[k+3] = RK[i][j+1][k+3] ^ (unsigned char)( temp >>  24) & 0xFF;
    }
  }
}
Copy64(tempdata,outdata);
}
```

12.4. Module hdn.h

```
/*  hdn.h */
#ifndef __HDN_H__
#define __HDN_H__ 1

#ifdef __cplusplus
extern "C" {
#endif

#include "dn.h"

extern unsigned char SMLPerm[r][c];
extern unsigned char IV_HDN[c];
extern unsigned char CONST0[c];
extern unsigned char CONST1[c];
```

```
typedef struct
{
       // rk is input/output array, the output is stored in rk[0][0][0..63]
    unsigned char rk[MAXRHO][r][c];
    unsigned long hbits, lbits;
    unsigned long mlen;
       int rho; // expected values 1 ... MAXRHO
} HDN_CTX;

int Init_HDN(HDN_CTX *ctx,int rho);
int Update_HDN( HDN_CTX* ctx, unsigned char* vdata, unsigned long data_len );
int Final_HDN( HDN_CTX* ctx );
int Final_HDN_2( HDN_CTX* ctx );

#ifdef __cplusplus
}
#endif //#ifdef __cplusplus
#endif //#ifndef __HDN_H__
```

12.5. Module hdn.c

```
/* hdn.c */
/*  Unoptimized implementation of HDN.
In this implementation we suppose that the total length of hashed data is
less then 2^64. If not, easily change the function Update_HDN and Final_HDN.
*/

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

#include "hdn.h"
#include "dn.h"
/*============================================================*/
// Initializes hash context
int Init_HDN( HDN_CTX* ctx, int rho )
{
       int i;

       ctx->rho    = rho;
       ctx->lbits = 0;
       ctx->hbits = 0;
       ctx->mlen  = 0;
       //initializing value
       for(i=0;i<c;i++) ctx->rk[0][0][i] = IV_HDN[i];
       return 0;
}
/*============================================================*/
// Processes one complete data block
static void Process_One_Block_HDN( HDN_CTX* ctx )
{
       unsigned char outdata[c];
       DN(ctx->rk,ctx->rho,CONST0,outdata,0);
       Copy64(outdata,ctx->rk[0][0]);
}
/*============================================================*/
/* This function reads data_len bytes from "vdata" and processes complete
blocks of 960 bytes. If the block is incomplete, data is stored into context
for later processing. The temporary hash value is stored in the context also.
In this implementation we suppose that the total length of hashed data is less
then 2^64.
```

```
*/
int Update_HDN( HDN_CTX* ctx, unsigned char* vdata, unsigned long data_len )
{
   unsigned char* data = vdata;
   unsigned long use, low_bits;

   ctx->hbits += data_len >> 29;
   low_bits = data_len << 3;
   ctx->lbits += low_bits;
   if ( ctx->lbits < low_bits ) { ctx->hbits++; }

   use = 960 - ctx->mlen; if (use > data_len) use = data_len;
   memcpy( ctx->rk[0][1] + ctx->mlen, data, use );
   ctx->mlen += use;
   data_len -= use;
   data += use;

   while ( ctx->mlen == 960 )
   {
      Process_One_Block_HDN( ctx );
      use = 960; if (use > data_len) use = data_len;
      memcpy( ctx->rk[0][1], data, use );
      ctx->mlen = use;
      data_len -= use;
      data += use;
   }
   return 0;
}
/*==============================================================*/
/* This function processes the last data block. It pads 16 bytes of the data
length. In some cases the padding creates a new data block. In this
implementation we suppose that the total length of hashed data is less then
2^64. If not, easily change the function Update_HDN and Final_HDN.
*/
int Final_HDN( HDN_CTX* ctx )
{
   if ( ctx->mlen < 960-16 )
   {
       ctx->rk[0][1][ ctx->mlen ] = 0x80; ctx->mlen++;
      memset( ctx->rk[0][1] + ctx->mlen, 0x00, 960-8 - ctx->mlen );
   }
   else
   {
      ctx->rk[0][1][ ctx->mlen ] = 0x80;
      ctx->mlen++;
      memset( ctx->rk[0][1] + ctx->mlen, 0x00, 960 - ctx->mlen );
      Process_One_Block_HDN( ctx );
      memset( ctx->rk[0][1], 0x00, 960-8 );
   }
   ctx->rk[0][r-1][56] = (unsigned char)((ctx->hbits >> 24) & 0xFF);
   ctx->rk[0][r-1][57] = (unsigned char)((ctx->hbits >> 16) & 0xFF);
   ctx->rk[0][r-1][58] = (unsigned char)((ctx->hbits >> 8) & 0xFF);
   ctx->rk[0][r-1][59] = (unsigned char)(ctx->hbits & 0xFF);

   ctx->rk[0][r-1][60] = (unsigned char)((ctx->lbits >> 24) & 0xFF);
   ctx->rk[0][r-1][61] = (unsigned char)((ctx->lbits >> 16) & 0xFF);
   ctx->rk[0][r-1][62] = (unsigned char)((ctx->lbits >> 8) & 0xFF);
   ctx->rk[0][r-1][63] = (unsigned char)(ctx->lbits & 0xFF);
   Process_One_Block_HDN( ctx );
   return 0;
}
/*==============================================================*/
/*
```

```
This function provides the final processing by the oracle g. The 64 bytes long
input rk[0][0] is padded by 960 zero bytes. Then DN is called with the
"plaintext" CONST1. The output is stored in ctx->rk[0][0].
*/
int Final_HDN_2( HDN_CTX* ctx )
{
	unsigned char outdata[c];
	// padding
	memset( ctx->rk[0][1], 0x00, 960);
	DN(ctx->rk,ctx->rho,CONST1,outdata,0);
	Copy64(outdata,ctx->rk[0][0]);
	return 0;
}
/*=============================================================*/
```

12.6. Module main_test_definition_DN_and_HDN.c

```
/* main_test_definition_DN_and_HDN.c */

/*Tests DN and HDN for the number of rounds rho = 1...MAXRHO (10).*/

#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <basetsd.h>
#include  <time.h>

#include "hdn.h"
#include "dn.h"

extern unsigned char SubsB[256];
extern unsigned char SubsF[256];
extern unsigned char SMLPerm[r][c];
extern unsigned char MDS4x4[4][4];
extern unsigned char MDS16x16[r][r];
extern unsigned char IV_HDN[c];
extern unsigned char CONST0[c];
extern unsigned char CONST1[c];
extern unsigned long RConstB[10][16];
extern unsigned long RConstF[9][64];

unsigned char rk[MAXRHO][r][c];
/*=====================================================*/
/* Functions StartTimer() and StopTimer() measure the time in between their
callings */
FILETIME cre, ex, krn, usr, usr2;
void StartTimer()
{
	HANDLE hThread;
	hThread = GetCurrentThread();
	GetThreadTimes( hThread, &cre, &ex, &krn, &usr );
}
double StopTimer()
{
	double delta;
	__int64 i,i2,res;
	HANDLE hThread;
	hThread = GetCurrentThread();
```

```
       GetThreadTimes( hThread, &cre, &ex, &krn, &usr2 );
       i = usr.dwHighDateTime;
       i = i <<32;
       i+= usr.dwLowDateTime;

       i2 = usr2.dwHighDateTime;
       i2 = i2 <<32;
       i2+= usr2.dwLowDateTime;

       res = i2 - i;
       res/=1000;
       delta = (double)res;
       delta/=10000;
       return delta;
}
/*============================================================*/
/* Test values of DN for the number of rounds 1 .. MAXRHO (10)
   plaintext: CONST0
   key (1024 bytes): "abc", 0x80, 1004 zero bytes, 16 bytes of the length
*/

unsigned char DN_abc_CONST0[MAXRHO][c] =
{
0x8F,0xB3,0xE2,0x25, 0xC5,0x62,0x3E,0xD9, 0xAC,0x62,0x0F,0x5E, 0x03,0x41,0x82,0x20,
0xBB,0xC1,0x63,0xB1, 0x70,0x06,0x44,0xDA, 0x5A,0x87,0x38,0x64, 0x81,0x59,0xD9,0x8A,
0x88,0xCC,0x73,0x0E, 0x6B,0x5C,0xCE,0x44, 0x63,0x53,0xED,0x1B, 0x35,0xC0,0x0C,0x6D,
0xFE,0x51,0xE1,0x15, 0x75,0xD4,0xA7,0x3B, 0x05,0xF3,0x85,0xF0, 0x02,0x21,0x7C,0x95,

0x8B,0x96,0x50,0x0D, 0xBF,0x8C,0xD9,0x5D, 0x21,0x58,0x7F,0x56, 0x2A,0xA2,0x8A,0x0C,
0x6D,0xF7,0x69,0x71, 0x25,0x0A,0x40,0x1D, 0x1C,0xF8,0x97,0x3A, 0xDB,0x1F,0x93,0x3B,
0x9F,0x1E,0x9F,0xAD, 0x06,0xC5,0xC8,0x34, 0x63,0xD2,0x9E,0xF3, 0xF1,0xDD,0x9E,0x91,
0x75,0x7E,0xC3,0x09, 0x79,0x93,0x96,0xD4, 0x45,0x8E,0xB5,0x74, 0x46,0xAC,0x46,0x80,

0x12,0x64,0xDE,0xED, 0xCC,0x85,0x8F,0xC4, 0x7C,0x7E,0xC5,0x4B, 0xCF,0xF9,0x49,0x3D,
0x10,0x61,0x02,0x30, 0xDA,0xD0,0x76,0x8B, 0x55,0xE8,0x50,0xE7, 0x46,0x82,0xBB,0x15,
0xA3,0x05,0xF2,0xAB, 0xFF,0x87,0xB3,0x82, 0x02,0x23,0x40,0x31, 0x23,0x07,0x83,0xAD,
0x57,0xA4,0xB6,0x96, 0x8D,0x20,0x8B,0x63, 0x29,0xD5,0xC0,0x77, 0x56,0x47,0xB8,0x55,

0x03,0xFA,0x91,0xBB, 0x43,0x1F,0xC2,0x48, 0xF6,0xBC,0x36,0x12, 0xCE,0x44,0xB5,0x44,
0x31,0xC3,0xB5,0x1D, 0x5E,0x70,0x09,0xF8, 0x55,0x2C,0x93,0xF9, 0xC9,0x92,0x46,0xF8,
0x49,0x5F,0xC2,0x3F, 0x72,0x60,0xB4,0xE5, 0x6B,0x02,0x8D,0x49, 0x9F,0x7E,0xDA,0x85,
0xFC,0xBC,0x4F,0x15, 0x48,0xCD,0x67,0x60, 0x29,0x05,0xF3,0x14, 0x48,0xFA,0x51,0xC1,

0x6E,0x82,0x49,0xBA, 0x28,0xC0,0x1A,0x7B, 0x73,0x3D,0x66,0xAC, 0x22,0x04,0x4F,0x84,
0x5C,0xBB,0x12,0x5F, 0x91,0x2F,0xAF,0x6B, 0xCC,0x70,0xDE,0x30, 0xAE,0xD4,0x37,0x36,
0xBB,0x55,0xBC,0x43, 0x8B,0x29,0xDC,0xAA, 0x45,0x11,0xCC,0xA1, 0xA1,0x00,0xC8,0xBD,
0x79,0x74,0x52,0x5C, 0xE4,0xD8,0xFF,0x4F, 0x51,0xE2,0x7A,0x70, 0xFE,0x1B,0xC4,0x5C,

0xD5,0x82,0xEF,0xC3, 0x90,0x1F,0x6F,0xAA, 0xC2,0x2D,0x84,0x06, 0x13,0x5A,0xC8,0x17,
0xE1,0xFB,0xB4,0x46, 0x04,0x78,0xA0,0xEF, 0xA0,0xDD,0xBE,0xEF, 0x3B,0x95,0x51,0x14,
0x9C,0x86,0x6E,0x85, 0x1C,0xCD,0x12,0xEA, 0xAB,0x24,0x93,0x67, 0x78,0x9A,0x34,0x6F,
0xAE,0x5B,0x25,0xD9, 0xC7,0x56,0xA4,0xE4, 0x46,0x7A,0x0D,0x00, 0x85,0x7B,0x6F,0xC9,

0xE9,0x31,0xCF,0xC2, 0xA2,0xF5,0x1F,0xE6, 0x36,0x63,0x11,0x6B, 0xBD,0xE5,0x57,0xAB,
0x76,0x7C,0x97,0xBF, 0xEC,0x73,0x78,0xF3, 0x3A,0x92,0xA7,0x62, 0xA4,0xCE,0x2A,0x61,
0x38,0x3F,0x28,0x1F, 0x17,0x1C,0x2B,0xD6, 0x9C,0x49,0x7F,0x20, 0x17,0x33,0xCE,0x4A,
0x4B,0x54,0x17,0x26, 0x54,0x9D,0xB1,0xCA, 0x58,0x66,0x38,0x88, 0x82,0x0A,0xC2,0xEA,

0xC7,0x82,0x98,0x97, 0xEC,0x28,0xC9,0x25, 0xE1,0xCB,0x62,0x42, 0x0E,0x36,0xAE,0x10,
0xAC,0x2B,0xF3,0x88, 0x77,0xA7,0x73,0x1D, 0x7A,0x37,0x94,0xC8, 0x26,0x18,0x2D,0x9D,
0xB4,0xBF,0x4A,0xE5, 0x02,0xC6,0x56,0x8E, 0x3A,0xE2,0x9A,0x44, 0xC6,0xA6,0x16,0x34,
0x17,0x77,0x7D,0x7A, 0xDE,0xE0,0xA2,0xBA, 0xC5,0xF7,0xE2,0xB2, 0x31,0x81,0x20,0xFB,

0x48,0x23,0x84,0xB2, 0x41,0xDC,0x1C,0x6B, 0xC5,0xAE,0x55,0x8D, 0x48,0xC1,0x29,0xFD,
0x89,0x6F,0x47,0xE6, 0x8A,0x88,0x83,0x96, 0x63,0x33,0xF1,0x71, 0xD7,0xEF,0x85,0xA0,
0xC4,0x25,0x81,0x3A, 0xEA,0xC8,0x80,0x23, 0x6E,0x17,0xA2,0x0C, 0x60,0xFE,0xF0,0xD6,
0x90,0xCA,0x3A,0x42, 0x38,0x04,0xFF,0xFA, 0xDE,0x05,0x28,0xAD, 0x83,0x8C,0xBA,0x3F,

0x69,0x9E,0xDA,0xC8, 0x3A,0x7D,0x6D,0x93, 0xBC,0x22,0xE3,0x42, 0xB9,0x57,0xC4,0x43,
```

```
0x3F,0x74,0x8E,0xD1, 0x67,0xC8,0x27,0x4B, 0x5F,0x38,0x60,0x13, 0x79,0x64,0x33,0xEB,
0xF9,0x26,0x65,0x5A, 0x04,0x1E,0x3F,0x50, 0xEB,0xB2,0x3C,0x7F, 0x0E,0x51,0x1D,0x7C,
0x74,0x5A,0x99,0x98, 0x92,0xED,0x75,0xB1, 0xC1,0x7E,0x87,0xB4, 0x66,0x58,0xD5,0x54
};
/*==========================================================*/
/* Test values of DN for the number of rounds 1 .. MAXRHO (10).
   plaintext = CONST1
   key (1024 bytes): defined in the documentation
*/
unsigned char DN_abc_CONST1[MAXRHO][c] =
{
0xD9,0x9F,0xFD,0xFD, 0x2A,0x6E,0x89,0x07, 0xA3,0x05,0x10,0xC9, 0x87,0x29,0x4A,0x86,
0x99,0x40,0xEF,0x84, 0xF0,0x1B,0x1B,0x4D, 0xE7,0x73,0x63,0x1D, 0x24,0x12,0x78,0x84,
0xEE,0xCB,0xCB,0xBE, 0x9D,0xC5,0x2D,0x98, 0x1A,0xC9,0xD4,0x36, 0x77,0x14,0x3C,0x85,
0xD6,0x67,0xF3,0x8F, 0x9A,0xB8,0xD0,0x38, 0x21,0xDE,0x9D,0xB2, 0xEC,0x65,0x1A,0xBE,

0xB1,0xB5,0x3A,0x8C, 0x25,0x07,0xF5,0xAC, 0x4B,0xA0,0x04,0xBC, 0x51,0x88,0xE5,0xA7,
0x4B,0x03,0x71,0xE4, 0x6B,0xB7,0x77,0xB7, 0x99,0xC5,0x50,0xBC, 0x9B,0xF4,0xC5,0x27,
0xDA,0xFC,0x33,0x15, 0x84,0xCC,0x80,0x69, 0x15,0x3A,0x52,0x05, 0x17,0xCA,0xAD,0x86,
0xCB,0x43,0xEC,0xA2, 0x47,0x17,0x23,0x5A, 0x03,0x55,0xC4,0x96, 0xE7,0xD5,0x80,0x0C,

0xD0,0xA0,0x17,0xE5, 0xF1,0x3D,0x84,0xCC, 0x59,0x9D,0xFA,0x39, 0xA0,0x98,0x90,0xD2,
0x5D,0x3E,0x82,0xBE, 0xE5,0xD4,0xDF,0x24, 0x73,0x27,0xE3,0xE2, 0x37,0x43,0x92,0x85,
0x45,0xC2,0x55,0xC4, 0x0D,0x92,0xBC,0x72, 0xC7,0xD6,0xC1,0x16, 0xFD,0x13,0xA6,0xE7,
0x49,0x79,0xFD,0xFC, 0x72,0x74,0x3D,0xF7, 0x7B,0x91,0x16,0x19, 0xC7,0xC9,0x66,0x2D,

0x50,0x4A,0x88,0x2A, 0x17,0xAD,0x8C,0xE1, 0xF2,0x63,0xA3,0xBA, 0x6C,0x36,0xC7,0x6A,
0xDC,0x4E,0xA5,0x30, 0xAB,0x85,0xDC,0xDD, 0xE0,0x26,0xFB,0xC8, 0x37,0x17,0x1C,0xFD,
0x6C,0x5A,0x99,0xA3, 0x78,0x95,0xAA,0xF1, 0x26,0xCE,0xB4,0x0F, 0x9C,0x95,0xB2,0xB9,
0xBB,0x2D,0x6E,0x57, 0x58,0xD3,0x1C,0xD4, 0x7C,0x9E,0xEE,0x9A, 0x83,0xBE,0x07,0xB7,

0xF9,0x3A,0x44,0xFA, 0xCC,0x23,0x6F,0xCA, 0xF3,0xA3,0x90,0xD6, 0x47,0xFC,0x14,0xDB,
0x94,0xAA,0xC9,0xE5, 0xE3,0x0E,0x51,0xA0, 0xBB,0xF2,0xF1,0x54, 0xB7,0x11,0xD1,0xB8,
0x32,0xE1,0x5F,0x78, 0x4A,0x62,0x68,0x6E, 0xDA,0xFF,0x3E,0x1B, 0x59,0xAA,0xF5,0x61,
0x2D,0x64,0x0E,0xFB, 0x48,0x57,0x3A,0x95, 0xC9,0xA2,0xCF,0x69, 0xB9,0xA9,0x58,0x38,

0x33,0xF7,0x01,0xF0, 0xF5,0x27,0x5F,0xEB, 0xF6,0x80,0xD9,0x68, 0x63,0x87,0x88,0xE2,
0x6B,0x7B,0x29,0xB3, 0x76,0x6A,0x06,0x28, 0x9F,0x28,0x59,0x8A, 0x60,0xD2,0x91,0xAA,
0xD4,0xE8,0x0B,0x52, 0x5B,0xB3,0xF8,0x3E, 0x57,0x10,0x95,0x51, 0x37,0x26,0x58,0x70,
0xF8,0x11,0x90,0xD8, 0x7A,0x9B,0xD1,0x46, 0xAD,0x0A,0xEA,0x0A, 0xD2,0xE2,0x2D,0x20,

0x06,0xF8,0x9B,0xD9, 0x78,0xF9,0x88,0x6E, 0x7F,0xDD,0xC5,0x15, 0xA7,0x03,0xC7,0x48,
0x03,0xC8,0x0D,0x0C, 0x05,0xBA,0x49,0x13, 0x24,0x34,0xE3,0x16, 0xED,0x03,0xB2,0xDC,
0x5F,0xAC,0x20,0x24, 0x2E,0xC9,0xF8,0x3E, 0x72,0xB3,0x6A,0xFE, 0xA1,0xDF,0xAB,0xC7,
0xAE,0x90,0xE3,0x6B, 0xA3,0xEA,0x1B,0x14, 0x60,0xCE,0xEE,0xBB, 0xBF,0xF9,0xD1,0x60,

0xE0,0xAD,0xD3,0x17, 0x03,0x8A,0x22,0x16, 0x30,0x77,0xFA,0x38, 0xDC,0x5E,0xAD,0x8C,
0x8A,0xB1,0xD2,0x76, 0x2E,0x7C,0xAD,0xBA, 0xC8,0x21,0x14,0xC8, 0xF2,0x85,0x7E,0xCF,
0x88,0xAD,0x58,0x14, 0xF4,0xA3,0x0C,0xA6, 0x83,0x4C,0x8B,0xFB, 0x27,0xB0,0x1A,0xCE,
0x5E,0x17,0xC0,0xB6, 0xBC,0x57,0x7A,0x18, 0xB9,0xF7,0x50,0xFF, 0x1C,0x43,0x04,0xE4,

0xD7,0x5C,0xA2,0xAA, 0xE3,0x4A,0xEA,0x2F, 0x2A,0xE2,0x35,0xC4, 0xE3,0x39,0x24,0x6B,
0x84,0x45,0xA3,0xC1, 0xEE,0xD0,0xC2,0x42, 0xE7,0xE2,0xF3,0x7B, 0xBC,0xC9,0x19,0x52,
0x84,0xCC,0x57,0x7D, 0x8B,0x20,0x73,0x17, 0x6C,0x1B,0x5A,0x89, 0x31,0xDC,0xBB,0xFC,
0x2D,0xF4,0xF3,0x2D, 0xBF,0xF8,0x1A,0xFB, 0x47,0xE1,0xF0,0x20, 0xD4,0x30,0x0A,0x44,

0xB4,0x04,0xF6,0xF0, 0x41,0x8B,0xC3,0x10, 0x4E,0x82,0x61,0x2F, 0x4C,0x82,0x08,0x02,
0x38,0xA4,0xE7,0xAA, 0x55,0x14,0x9E,0xB4, 0x15,0x59,0x64,0x99, 0x85,0xFC,0x5F,0x56,
0x20,0xD2,0xDE,0x41, 0x0F,0x95,0x10,0x19, 0x70,0x5B,0x4A,0x1B, 0x95,0x59,0xBA,0xE2,
0x86,0x92,0xC4,0x4B, 0x4E,0xAB,0x11,0x61, 0x2E,0x0E,0xEE,0xAE, 0x70,0x70,0xC6,0xFA
};

/*==========================================================*/
/* (mega)test values of DN for the number of rounds 1 .. MAXRHO (10),
plaintext and key defined in the documentation
*/
unsigned char DN_mega[MAXRHO][c]=
{
0x18,0xD3,0x64,0xE2, 0x0B,0xAB,0x2D,0x78, 0xE9,0x40,0x25,0x30, 0xB5,0xC5,0x39,0x7F,
0xF6,0x5A,0x12,0xC8, 0x8B,0x1D,0x55,0x1F, 0xC5,0x38,0x21,0x0C, 0x88,0x71,0x49,0x15,
```

```
0x1F,0x9E,0x5A,0xB5, 0xD0,0xFD,0x85,0x36, 0x41,0x33,0xB9,0x68, 0x8A,0x8C,0x61,0xE8,
0x9B,0x3B,0xE7,0xF8, 0xA8,0x3F,0xCD,0x8D, 0xB7,0xEA,0x71,0x74, 0x90,0x91,0x51,0xE2,

0x3A,0x72,0x44,0x05, 0xFB,0xF0,0xD2,0x4D, 0x85,0x80,0x91,0xD3, 0x7F,0x99,0xC9,0xB7,
0x59,0x36,0xA6,0x74, 0x1A,0x20,0x86,0x3D, 0x51,0x30,0x32,0xE4, 0x2D,0xB8,0xF4,0x13,
0xF6,0x21,0x8F,0x93, 0x0A,0x37,0xC8,0x8F, 0x97,0x38,0x61,0xB9, 0xB6,0xBF,0x3F,0x1D,
0xE7,0xA7,0xFF,0xAF, 0x97,0x95,0xE7,0x15, 0x59,0x59,0xE3,0xBF, 0xA1,0x7E,0x67,0xEB,

0x85,0x13,0xB2,0x60, 0xD5,0xAE,0x42,0xFF, 0xE0,0x9B,0xE3,0x7E, 0x43,0x1C,0x5B,0xC1,
0x01,0xC3,0xD8,0x6F, 0x4F,0x99,0x6C,0x2B, 0x23,0xD3,0x99,0xE2, 0x00,0xD6,0x10,0x25,
0xFD,0x24,0x36,0x64, 0x42,0x00,0x0A,0x50, 0x83,0x5E,0xE4,0x47, 0xC7,0x84,0xF4,0x83,
0xFF,0xAC,0x36,0x7A, 0x54,0x34,0x6C,0x35, 0x75,0x74,0x02,0x0B, 0x74,0x00,0x09,0x30,

0x40,0xC2,0x9F,0x25, 0xE9,0x3A,0x1B,0x8A, 0xEE,0xE9,0x13,0x6F, 0x6B,0x6A,0x8B,0xF5,
0x19,0xE5,0xB6,0xB7, 0x83,0x96,0xF1,0xBF, 0x43,0x91,0xA8,0xBE, 0xD1,0x15,0xB3,0xCB,
0xCE,0x66,0xFA,0xE4, 0x64,0x84,0xD0,0x20, 0x32,0xFC,0xD4,0x51, 0xC6,0xAC,0xCE,0x69,
0x2E,0x20,0xA4,0x75, 0x49,0x79,0x35,0xE9, 0x47,0xC5,0xEE,0x03, 0x5A,0xC8,0xD9,0xDA,

0x47,0xA3,0xBD,0x8C, 0x8E,0xA2,0x13,0x03, 0x07,0xBA,0xBB,0x5C, 0xD8,0x91,0x8A,0x1F,
0xD6,0xC6,0x4D,0x23, 0x31,0x19,0xBB,0x07, 0x40,0x95,0xEE,0x36, 0xC2,0xA7,0xD7,0x5F,
0x20,0x04,0x09,0x20, 0xF5,0x20,0x9C,0x58, 0xC8,0x14,0xE7,0x49, 0xA1,0x23,0x2B,0x57,
0x49,0xEE,0x0A,0xE6, 0xD8,0xC8,0x54,0xFF, 0xA3,0x26,0x08,0x70, 0x20,0x60,0xAB,0xB9,

0xF0,0xD1,0x41,0xDE, 0xBA,0x3C,0xA3,0x3E, 0xBB,0x0F,0x44,0x5B, 0xA7,0x0B,0x32,0xFB,
0xB2,0x97,0xB6,0x7B, 0xD4,0x64,0xDD,0xC9, 0x16,0x98,0x21,0x72, 0xB1,0xE4,0xFB,0xEE,
0xD1,0x21,0xBF,0x18, 0x1B,0x5F,0xF5,0x36, 0xBF,0xFF,0xD6,0x94, 0x09,0x03,0x67,0xEE,
0x0B,0x67,0x36,0x02, 0xBC,0x45,0x05,0xF3, 0x85,0xA5,0x4A,0x2B, 0x75,0xEB,0x07,0xE9,

0x28,0xF3,0xF8,0xA2, 0xDA,0x05,0xE5,0x19, 0x9D,0x06,0xA6,0x35, 0xB3,0xA5,0xA4,0x50,
0x16,0x12,0x2E,0xD4, 0x81,0x66,0x0E,0x91, 0x70,0xD4,0x08,0xA0, 0x13,0x30,0xBE,0xFA,
0xA1,0x70,0xDA,0x7E, 0xE9,0x8D,0xD2,0xC4, 0x61,0x81,0x99,0xD1, 0x2A,0xE5,0x91,0xE6,
0x24,0xD9,0x3D,0xC3, 0x70,0x61,0x3F,0xF7, 0xB3,0xBB,0x46,0xE4, 0x0B,0x90,0x71,0x53,

0x06,0x19,0x4B,0xFB, 0x19,0x37,0xB9,0x24, 0xF6,0xBB,0x9F,0xD1, 0xFA,0xE9,0xE8,0x66,
0x6E,0x38,0x4A,0x25, 0xC0,0x19,0xAC,0x78, 0x10,0x2F,0x7A,0x16, 0x5F,0x80,0x8C,0x71,
0x51,0xBC,0x84,0xEC, 0xD5,0xAD,0x55,0x00, 0x7A,0x22,0x3B,0x09, 0xFD,0x7A,0xE3,0x9D,
0x32,0x79,0xC4,0xC9, 0xD0,0xCB,0xB2,0xA5, 0xEA,0x53,0x5E,0x25, 0x7E,0x68,0x33,0xAE,

0x00,0x34,0xC0,0x75, 0x17,0xF3,0x75,0xAA, 0x4E,0x66,0x31,0xA9, 0x25,0x29,0xAD,0x10,
0x7C,0xE8,0x79,0xEB, 0xD9,0x7C,0x1E,0x97, 0xF7,0x72,0x6A,0x11, 0x06,0xC3,0xC2,0x05,
0x5C,0xAA,0x19,0x64, 0x99,0x72,0x9C,0xD9, 0xD0,0xBE,0xB5,0x1C, 0xD5,0x0B,0xB7,0x11,
0x18,0x81,0x3A,0xD8, 0xC3,0x52,0x79,0x90, 0x47,0x05,0x20,0x24, 0x13,0x15,0x58,0x99,

0xB7,0x83,0x89,0xA5, 0xB9,0x58,0x68,0x46, 0xCD,0x4B,0x47,0xB2, 0x47,0xD0,0x0F,0xD3,
0xD6,0x4D,0x04,0x5E, 0xEA,0x14,0x79,0x62, 0x79,0xCC,0xC6,0xFA, 0xE3,0x03,0xDA,0x20,
0x38,0x47,0xF9,0x40, 0x16,0x6F,0x68,0x57, 0xF4,0xB6,0x3B,0x48, 0x63,0x94,0x64,0x10,
0xB7,0xE2,0xF5,0xE6, 0xED,0x2B,0x03,0x9F, 0x3A,0x75,0x9C,0x9F, 0xAA,0xCF,0x87,0x15
};
/*============================================================*/
/* Test values of HDN("abc") for the number of rounds 1 .. MAXRHO (10) */
unsigned char HDN_abc[MAXRHO][c] =
{
0xD9,0x9F,0xFD,0xFD, 0x2A,0x6E,0x89,0x07, 0xA3,0x05,0x10,0xC9, 0x87,0x29,0x4A,0x86,
0x99,0x40,0xEF,0x84, 0xF0,0x1B,0x1B,0x4D, 0xE7,0x73,0x63,0x1D, 0x24,0x12,0x78,0x84,
0xEE,0xCB,0xCB,0xBE, 0x9D,0xC5,0x2D,0x98, 0x1A,0xC9,0xD4,0x36, 0x77,0x14,0x3C,0x85,
0xD6,0x67,0xF3,0x8F, 0x9A,0xB8,0xD0,0x38, 0x21,0xDE,0x9D,0xB2, 0xEC,0x65,0x1A,0xBE,

0xB1,0xB5,0x3A,0x8C, 0x25,0x07,0xF5,0xAC, 0x4B,0xA0,0x04,0xBC, 0x51,0x88,0xE5,0xA7,
0x4B,0x03,0x71,0xE4, 0x6B,0xB7,0x77,0xB7, 0x99,0xC5,0x50,0xBC, 0x9B,0xF4,0xC5,0x27,
0xDA,0xFC,0x33,0x15, 0x84,0xCC,0x80,0x69, 0x15,0x3A,0x52,0x05, 0x17,0xCA,0xAD,0x86,
0xCB,0x43,0xEC,0xA2, 0x47,0x17,0x23,0x5A, 0x03,0x55,0xC4,0x96, 0xE7,0xD5,0x80,0x0C,

0xD0,0xA0,0x17,0xE5, 0xF1,0x3D,0x84,0xCC, 0x59,0x9D,0xFA,0x39, 0xA0,0x98,0x90,0xD2,
0x5D,0x3E,0x82,0xBE, 0xE5,0xD4,0xDF,0x24, 0x73,0x27,0xE3,0xE2, 0x37,0x43,0x92,0x85,
0x45,0xC2,0x55,0xC4, 0x0D,0x92,0xBC,0x72, 0xC7,0xD6,0xC1,0x16, 0xFD,0x13,0xA6,0xE7,
0x49,0x79,0xFD,0xFC, 0x72,0x74,0x3D,0xF7, 0x7B,0x91,0x16,0x19, 0xC7,0xC9,0x66,0x2D,

0x50,0x4A,0x88,0x2A, 0x17,0xAD,0x8C,0xE1, 0xF2,0x63,0xA3,0xBA, 0x6C,0x36,0xC7,0x6A,
0xDC,0x4E,0xA5,0x30, 0xAB,0x85,0xDC,0xDD, 0xE0,0x26,0xFB,0xC8, 0x37,0x17,0x1C,0xFD,
0x6C,0x5A,0x99,0xA3, 0x78,0x95,0xAA,0xF1, 0x26,0xCE,0xB4,0x0F, 0x9C,0x95,0xB2,0xB9,
0xBB,0x2D,0x6E,0x57, 0x58,0xD3,0x1C,0xD4, 0x7C,0x9E,0xEE,0x9A, 0x83,0xBE,0x07,0xB7,
```

```
0xF9,0x3A,0x44,0xFA, 0xCC,0x23,0x6F,0xCA, 0xF3,0xA3,0x90,0xD6, 0x47,0xFC,0x14,0xDB,
0x94,0xAA,0xC9,0xE5, 0xE3,0x0E,0x51,0xA0, 0xBB,0xF2,0xF1,0x54, 0xB7,0x11,0xD1,0xB8,
0x32,0xE1,0x5F,0x78, 0x4A,0x62,0x68,0x6E, 0xDA,0xFF,0x3E,0x1B, 0x59,0xAA,0xF5,0x61,
0x2D,0x64,0x0E,0xFB, 0x48,0x57,0x3A,0x95, 0xC9,0xA2,0xCF,0x69, 0xB9,0xA9,0x58,0x38,

0x33,0xF7,0x01,0xF0, 0xF5,0x27,0x5F,0xEB, 0xF6,0x80,0xD9,0x68, 0x63,0x87,0x88,0xE2,
0x6B,0x7B,0x29,0xB3, 0x76,0x6A,0x06,0x28, 0x9F,0x28,0x59,0x8A, 0x60,0xD2,0x91,0xAA,
0xD4,0xE8,0x0B,0x52, 0x5B,0xB3,0xF8,0x3E, 0x57,0x10,0x95,0x51, 0x37,0x26,0x58,0x70,
0xF8,0x11,0x90,0xD8, 0x7A,0x9B,0xD1,0x46, 0xAD,0x0A,0xEA,0x0A, 0xD2,0xE2,0x2D,0x20,

0x06,0xF8,0x9B,0xD9, 0x78,0xF9,0x88,0x6E, 0x7F,0xDD,0xC5,0x15, 0xA7,0x03,0xC7,0x48,
0x03,0xC8,0x0D,0x0C, 0x05,0xBA,0x49,0x13, 0x24,0x34,0xE3,0x16, 0xED,0x03,0xB2,0xDC,
0x5F,0xAC,0x20,0x24, 0x2E,0xC9,0xF8,0x3E, 0x72,0xB3,0x6A,0xFE, 0xA1,0xDF,0xAB,0xC7,
0xAE,0x90,0xE3,0x6B, 0xA3,0xEA,0x1B,0x14, 0x60,0xCE,0xEE,0xBB, 0xBF,0xF9,0xD1,0x60,

0xE0,0xAD,0xD3,0x17, 0x03,0x8A,0x22,0x16, 0x30,0x77,0xFA,0x38, 0xDC,0x5E,0xAD,0x8C,
0x8A,0xB1,0xD2,0x76, 0x2E,0x7C,0xAD,0xBA, 0xC8,0x21,0x14,0xC8, 0xF2,0x85,0x7E,0xCF,
0x88,0xAD,0x58,0x14, 0xF4,0xA3,0x0C,0xA6, 0x83,0x4C,0x8B,0xFB, 0x27,0xB0,0x1A,0xCE,
0x5E,0x17,0xC0,0xB6, 0xBC,0x57,0x7A,0x18, 0xB9,0xF7,0x50,0xFF, 0x1C,0x43,0x04,0xE4,

0xD7,0x5C,0xA2,0xAA, 0xE3,0x4A,0xEA,0x2F, 0x2A,0xE2,0x35,0xC4, 0xE3,0x39,0x24,0x6B,
0x84,0x45,0xA3,0xC1, 0xEE,0xD0,0xC2,0x42, 0xE7,0xE2,0xF3,0x7B, 0xBC,0xC9,0x19,0x52,
0x84,0xCC,0x57,0x7D, 0x8B,0x20,0x73,0x17, 0x6C,0x1B,0x5A,0x89, 0x31,0xDC,0xBB,0xFC,
0x2D,0xF4,0xF3,0x2D, 0xBF,0xF8,0x1A,0xFB, 0x47,0xE1,0xF0,0x20, 0xD4,0x30,0x0A,0x44,

0xB4,0x04,0xF6,0xF0, 0x41,0x8B,0xC3,0x10, 0x4E,0x82,0x61,0x2F, 0x4C,0x82,0x08,0x02,
0x38,0xA4,0xE7,0xAA, 0x55,0x14,0x9E,0xB4, 0x15,0x59,0x64,0x99, 0x85,0xFC,0x5F,0x56,
0x20,0xD2,0xDE,0x41, 0x0F,0x95,0x10,0x19, 0x70,0x5B,0x4A,0x1B, 0x95,0x59,0xBA,0xE2,
0x86,0x92,0xC4,0x4B, 0x4E,0xAB,0x11,0x61, 0x2E,0x0E,0xEE,0xAE, 0x70,0x70,0xC6,0xFA
};
/*==========================================================*/
/* (mega)test values of HDN for the number of rounds 1 .. MAXRHO (10) defined
in the documentation */
unsigned char HDN_mega[MAXRHO][c]=
{
0x24,0x3D,0xC5,0x89, 0xD6,0xA7,0x43,0x09, 0x25,0x1E,0x1D,0xF0, 0xBB,0xF3,0x93,0x89,
0xCD,0xDE,0x0E,0x0A, 0x29,0x07,0xAF,0x35, 0x70,0x96,0x1D,0xAB, 0x5C,0x83,0x3C,0xB5,
0x1F,0x5F,0x6A,0x66, 0x53,0x55,0xFE,0x15, 0xA4,0x92,0xBF,0x62, 0xC8,0xC3,0xCB,0x24,
0x37,0x8D,0x5C,0x62, 0x10,0x50,0x91,0xC1, 0xE0,0x31,0xB5,0x85, 0x4D,0x33,0x62,0x0E,

0xD2,0x18,0x4B,0x7A, 0x0E,0xC7,0xAD,0xB6, 0x72,0x51,0x81,0x50, 0xD7,0x96,0x16,0xA0,
0x35,0x07,0x2B,0xE4, 0x95,0x1E,0x32,0x62, 0xEE,0x84,0x38,0x49, 0x4A,0x6A,0x57,0x87,
0x05,0x94,0x1E,0x9E, 0xCB,0xAB,0x68,0x47, 0xE2,0x80,0x17,0xCC, 0xA5,0x10,0x53,0x69,
0xB9,0x44,0x83,0xCB, 0x59,0x35,0x1B,0x50, 0x32,0x30,0x1E,0xC2, 0x3A,0xBB,0x6B,0x03,

0x11,0x73,0x1A,0xB9, 0xFA,0x98,0xA6,0x58, 0x99,0xA2,0x9C,0x83, 0xE9,0xEA,0xF2,0x82,
0xF1,0x90,0x39,0x99, 0xC5,0x19,0x26,0x5D, 0xE9,0x59,0x23,0x47, 0xCB,0xEF,0x34,0xFC,
0x5A,0x4C,0xDD,0xDA, 0x10,0x61,0xDE,0x19, 0x96,0x9F,0x43,0x24, 0x5F,0x90,0x4E,0x0D,
0xB3,0x11,0xB2,0xA8, 0xAA,0x49,0x36,0x17, 0xC9,0x0B,0xFB,0x57, 0x72,0xFE,0x2A,0x43,

0xCF,0x7B,0x80,0x43, 0x5B,0xE0,0x49,0x25, 0x41,0xAA,0xB4,0xF0, 0x65,0x10,0x44,0x10,
0xE8,0xF0,0x1A,0x56, 0x67,0xD7,0x53,0xB2, 0xB3,0x1A,0x36,0x12, 0xE7,0x0A,0xD7,0x7B,
0x39,0xF7,0x7F,0x61, 0x28,0xCC,0xC6,0xA7, 0x3E,0xD1,0x17,0xEF, 0x09,0x6A,0xAC,0x92,
0x58,0x45,0x0B,0xBA, 0xB3,0x1B,0x6F,0x04, 0xF3,0x7F,0x78,0x73, 0xDA,0x32,0x98,0x95,

0xE9,0x04,0x84,0xD3, 0x93,0x8A,0x2D,0x15, 0x65,0xE6,0x20,0xE8, 0xDB,0x20,0x77,0xC8,
0x27,0x7F,0x01,0xFC, 0x5E,0x7C,0xF8,0xD1, 0xFE,0x8A,0x3B,0x26, 0x55,0x02,0xB7,0xB9,
0xBB,0x4F,0x2B,0xA1, 0xBC,0xE0,0xD1,0xAB, 0x57,0x7C,0x5D,0x8A, 0x9D,0xDF,0x74,0x7A,
0x81,0x9A,0x43,0x45, 0x82,0x77,0xC3,0xDF, 0x13,0xDE,0xAA,0x40, 0xF3,0x1B,0x6F,0x64,

0x6B,0x2E,0x4A,0x15, 0x41,0x96,0xC7,0x59, 0xF4,0x46,0x07,0x25, 0x0F,0xBF,0x4C,0x59,
0x51,0x5C,0x8F,0x8F, 0x78,0x6F,0xBA,0x83, 0x86,0xBF,0x9F,0x61, 0x3D,0xEB,0xA3,0x98,
0xFB,0x20,0x46,0x31, 0xAA,0x1B,0x4C,0x3A, 0x42,0x30,0x07,0x7C, 0x30,0xED,0xCD,0x01,
0x1C,0x2A,0xA8,0x7E, 0x71,0xEA,0x10,0x10, 0xCF,0xEF,0x7E,0xE6, 0x31,0xCD,0x99,0x90,

0x5B,0xE6,0x45,0xA4, 0x1B,0xCC,0xF6,0x75, 0xCD,0xFD,0xB8,0x4D, 0x53,0xC3,0x37,0x60,
0x43,0x1A,0x4B,0xD8, 0xAA,0x03,0xD1,0xA1, 0x3B,0x9F,0x99,0x6C, 0x96,0xF6,0x41,0x67,
0x8B,0x57,0xD1,0x27, 0x5A,0x1E,0xC2,0x77, 0x3F,0xC5,0x3F,0xA0, 0x59,0x15,0xEC,0xB2,
0x83,0xE7,0xAA,0xBE, 0x3B,0x65,0xB0,0x69, 0xB5,0x4F,0xB3,0x51, 0x19,0xC1,0x5F,0xD0,
```

```
0xB8,0xAD,0x7C,0x8C, 0x01,0xAF,0xFB,0xAE, 0xD8,0xB9,0xA4,0x1E, 0x5E,0xFC,0x23,0x10,
0x57,0xA3,0xAC,0x28, 0x08,0xAE,0x5A,0x41, 0x67,0x7A,0x5F,0xF5, 0x2F,0x32,0x5E,0xC7,
0xFF,0x0A,0xDA,0xA9, 0x1A,0xB9,0x84,0x55, 0x5E,0x33,0xF5,0x6D, 0x3B,0x27,0x06,0x22,
0x76,0x47,0x84,0xBF, 0x32,0x76,0x0A,0x7A, 0x5F,0xEF,0xB7,0xAD, 0xE3,0x3C,0xBC,0x19,

0x8C,0x2F,0xE1,0x95, 0xBD,0x87,0xB9,0xD6, 0xB4,0x28,0xCF,0xEF, 0x8F,0x87,0x18,0x2B,
0xA9,0xF3,0x09,0x55, 0xA7,0x57,0xD7,0xC1, 0x83,0xA5,0xAF,0x6B, 0x7D,0x79,0x2B,0x5E,
0x88,0x05,0xCB,0x5A, 0x6C,0xB1,0x7F,0xDB, 0xB3,0x65,0x9E,0xAA, 0x1B,0xE2,0xA0,0xB8,
0x6F,0xE9,0xA3,0x73, 0x72,0x32,0xE6,0xD0, 0xFC,0xF1,0x6A,0x67, 0x0E,0x3A,0xDF,0x94,

0x37,0x55,0x09,0xFC, 0xAB,0xB8,0x78,0x36, 0x50,0x0F,0x32,0xD6, 0xD6,0x15,0x00,0x76,
0x26,0x9E,0x93,0xF5, 0xAA,0xF1,0xE6,0xBB, 0x34,0xD0,0x34,0x44, 0xE0,0xDF,0x4C,0x8B,
0x11,0x6B,0x35,0xF8, 0xE3,0xED,0x65,0x05, 0x46,0x4C,0xB0,0xE3, 0x4B,0x64,0x92,0x8A,
0xE9,0xED,0xB9,0x98, 0x65,0xA4,0xD5,0x25, 0x89,0x23,0x6A,0x9A, 0x48,0xA7,0x76,0x01
};
/*==========================================================*/
/* In this test the function DN encrypts the plaintext CONST0. The content of
the key in RK corresponds to the state of RK after hashing of the string "abc"
by HDN.
*/
int test_abc_DN_CONST0(int rho)
{
	unsigned char indata[c];
	unsigned char outdata[c];
	int i,j,x;
	for(i=0;i<MAXRHO;i++)  for(j=0;j<r;j++)  for(x=0;x<c;x++)  rk[i][j][x]  =
0x00;
	// IV
	for(x=0;x<c;x++)    rk[0][0][x] = IV_HDN[x];
	// as in hashing of "abc"
	rk[0][1][0] = 'a';
	rk[0][1][1] = 'b';
	rk[0][1][2] = 'c';
	rk[0][1][3] = 0x80;
	// padding the length
	rk[0][r-1][63] = 0x18;
	// oracle f
	for(x=0;x<c;x++)    indata[x] = CONST0[x];
	DN(rk,rho,indata,outdata,0);
	if ( memcmp( outdata, DN_abc_CONST0[rho-1], 64 ) == 0 )
		return 0;
	else
		return -1;
}
/*==========================================================*/
/* In this test the function DN encrypts the plaintext CONST1.  The key is
equal to the results of the first oracle f when hashing "abc". This is the
value DN_abc_CONST0[rho-1]), padded by zero bytes. This tests how DN behaves
in the final operation (oracle g) of HDN.
*/
int test_abc_DN_CONST1(int rho)
{
	unsigned char indata[c];
	unsigned char outdata[c];
	int i,j,x;
	// oracle g - padding by zero bytes
	for(i=0;i<MAXRHO;i++) for(j=0;j<r;j++) for(x=0;x<c;x++)
		 rk[i][j][x] = 0x00;
	// the initialization value is DN_abc_CONST0[rho-1][0..63]
	for(x=0;x<c;x++)    rk[0][0][x] = DN_abc_CONST0[rho-1][x];
	// oracle g: plaintext CONST1
	for(x=0;x<c;x++)    indata[x] = CONST1[x];
	DN(rk,rho,indata,outdata,0);
	if ( memcmp( outdata, DN_abc_CONST1[rho-1], 64 ) == 0 )
		return 0;
	else
```

```
		return -1;
}
/*=============================================================*/
/* This is DN mega-test, described in the Appendix E. */
int test_mega_DN(int rho)
{
	unsigned char indata[c];
	unsigned char outdata[c];
	int i,j,x;

	// RK[0] and entry for the first encryption
	for(j=0;j<r;j++) for(x=0;x<c;x++) rk[0][j][x] = 0;
	for(x=0;x<c;x++)	indata[x] = 0;

	// 100x
	for(i=0;i<100;i++)
	{
		DN(rk,rho,indata,outdata,0);
		for(x=0;x<c;x++)	indata[x] = outdata[x];

		for(j=0;j<r;j++)
		{
			DN(rk,rho,indata,outdata,0);
			for(x=0;x<c;x++)	rk[0][j][x] = outdata[x];
		}

	}

	if ( memcmp( outdata, DN_mega[rho-1], 64 ) == 0 )
		return 0;
	else
		return -1;
}
/*=============================================================*/
/* Test HDN("abc") */
int test_abc_HDN(int rho)
{
	HDN_CTX ctx;
	Init_HDN (&ctx,rho);
	Update_HDN(&ctx, "abc", 3);
	Final_HDN(&ctx);

	// intermediate value has to be equal to DN_abc_CONST0
	if ( memcmp( ctx.rk[0][0], DN_abc_CONST0[rho-1], 64 ) != 0 ) return -1;
	Final_HDN_2(&ctx);
	// check
	if ( memcmp( ctx.rk[0][0], HDN_abc[rho-1], 64 ) != 0 ) return -1;
	// it has to be HDN_abc ==  DN_abc_CONST1
	if ( memcmp( ctx.rk[0][0], DN_abc_CONST1[rho-1], 64 ) != 0 ) return -1;
	return 0;
}
/*=============================================================*/
/* This is mega-test for HDN, described in the Appendix E. */
int test_mega_HDN(int rho)
{
	unsigned char *data;
	int actual_len;
	unsigned char* actual_ptr;
	HDN_CTX ctx;
	int i,x;

	data = malloc(65000);

	actual_ptr = data;
```

```
	actual_len = 0;

	data[0] = 'a';		data[1] = 'b';		data[2] = 'c';
	actual_ptr+=3;
	actual_len+=3;

	for(i=0;i<100;i++)
	{
		Init_HDN (&ctx,rho);
		Update_HDN(&ctx, data, actual_len);
		Final_HDN(&ctx);
		Final_HDN_2(&ctx);

		// intermediate check of hash "abc"
		if(i == 0)
		if ( memcmp( ctx.rk[0][0], HDN_abc[rho-1], 64 ) != 0 ) return -1;

		for(x=0;x<c;x++)
		{
			*actual_ptr = ctx.rk[0][0][x];
			actual_ptr++;
			actual_len++;
		}
	}
	free(data);

	if ( memcmp( ctx.rk[0][0], HDN_mega[rho-1], 64 ) == 0 ) return 0;
	else return -1;
}
/*============================================================*/
int speed_test_HDN(unsigned long M, int rho)
{
	// hash M megabytes
	unsigned char buff[10000];
	unsigned long i;
	HDN_CTX ctx;
	memset(buff, 'a', 10000);
	Init_HDN (&ctx, rho);
	for (i = 0; i< 100*M; i++) Update_HDN(&ctx, buff,10000L);
	Final_HDN(&ctx);
	Final_HDN_2(&ctx);
	return 0;
}
/*============================================================*/
int main( void )
{
	int i,j,rho, print = 0;

	double		cas1, dt, speed;
	unsigned long delka;

	//initialize multiplication tables
	Init_MDS4x4_tables();
	Init_MDS16x16_tables();

	// check constants
	if ( Check_Const() == 0 )
		printf("Check_const: OK\n");
	else
		printf("Check_const: Failed\n");

	// check matrix
	if ( Check_Matrix() == 0 )
		printf("Check matrix: OK\n");
```

```
	else
		printf("Check matrix: Failed\n");

	// DN test, CONST0
	for(i=1;i<MAXRHO+1;i++)
	{
		if ( test_abc_DN_CONST0(i) == 0 )
			printf("test_abc_DN_CONST0(%d) OK\n",i);
		else
			printf("test_abc_DN_CONST0(%d) Failed\n",i);
	}

	// DN test, CONST1
	for(i=1;i<MAXRHO+1;i++)
	{
		if ( test_abc_DN_CONST1(i) == 0 )
		printf("test_abc_DN_CONST1(%d) OK\n",i);
		else
			printf("test_abc_DN_CONST1(%d) Failed\n",i);
	}

	// DN mega-test
	for(i=1;i<MAXRHO+1;i++)
	{
		if ( test_mega_DN(i) == 0 )
		printf("test_mega_DN(%d) OK\n",i);
		else
			printf("test_mega_DN(%d) Failed\n",i);
	}

	// HDN mega-test
	for(i=1;i<MAXRHO+1;i++)
	{
		if ( test_mega_HDN(i) == 0 )
			printf("test_mega_HDN(%d) OK\n",i);
		else
			printf("test_mega_HDN(%d) Failed\n",i);
	}

	// HDN, "abc"
	for(i=1;i<MAXRHO+1;i++)
	{
		if ( test_abc_HDN(i) == 0 )
			printf("test_abc_HDN(%d) OK\n",i);
		else
			printf("test_abc_HDN(%d) Failed\n",i);
	}

	//speed test
	printf( "\nspeed test - ...." );
	delka = 10;dt = 1.0*(double)(delka);
	for(rho = 1; rho <MAXRHO+1; rho++)
	{
		StartTimer();
		j = speed_test_HDN(delka, rho );
		cas1 = StopTimer();speed = dt/cas1;
		printf("\n   Length:   %4.0f   MB.   Speed   of   HDN-%d:   %f
		MByte/s",dt,rho,speed);
	}

	printf( "\nEnd of test. Push ENTER.\n" );
	getch();
	return 0;
}
```

13. Appendix E: Test vectors for DN(512, 8192) and HDN(512, 8192)

The test vectors are defined for ρ = 1 to 10 rounds in order to verify the implementations correctness. The corresponding test variables are shown in arrays of 10 elements. The variable ρ is denoted as *rho* in the source code, its maximal value as MAXRHO. HDN basic test is the hash code of "abc" string. During the hashing, transformation DN with CONST0 on input is called and with CONST1 on input within the final modification. The result is the HDN hash code. Transformation DN with CONST0 and CONST1 input constants are tested with input key arrays that are produced during the hashing of "abc" string with HDN. Thus, the test vectors for DN are the test vectors for the inner states of HDN hashing the string "abc". In total, we have these test vectors

- string DN_abc_CONST0, the result of the first transformation DN during string "abc" hashing,
- string DN_abc_CONST1, the result of DN with constant CONST1 within the final modification during string "abc" hashing,
- string HDN_abc, the final result of string "abc" hashing using HDN.

DN_abc_CONST1 and HDN_abc have to be identical. The strings DN_abc_CONST0 and DN_abc_CONST1 are compared with the strings obtained.

Moreover, so-called mega-tests are defined for both functions, DN and HDN where these function are called many times with different input data. Their description is to follow.

13.1. DN test vectors

13.1.1. DN_abc_CONST0

Plaintext: CONST0
Key:
rk[0][0] = IV,
rk[0][1][0] = 0x61; // 'a';
rk[0][1][1] = 0x62; // 'b';
rk[0][1][2] = 0x63; // 'c';
rk[0][1][3] = 0x80; // padding
all the remaining bytes rk[0][][] are 0x00 except the last byte (the length of bit string "abc" is 24 = 0x18): rk[0][15][63] = 0x18;
Ciphertext: see array DN_abc_CONST0[MAXRHO][*c*], where the result is stored (*c* = 64 bytes of ciphertext) for 1 to 10 rounds.

13.1.2. DN_abc_CONST1

Plaintext: CONST1
Key:
rk[0][0] = DN_abc_CONST0,
all the remaining bytes rk[0][][] are 0x00

Ciphertext: see array DN_abc_CONST1[MAXRHO][*c*], where the result is stored (*c* = 64 bytes of ciphertext) for 1 to 10 rounds.

13.1.3. DN mega-test

Both, the plaintext and round keys RK[0][0..15][0..63] are set to zeros during the initialization of this mega-test.
The following loop is repeated 100 times:

- With the current setting encrypt the plaintext. The plaintext is overwritten with the resulting 64 bytes.
- With the current setting encrypt the plaintext. Round key RK[0][0] is overwritten with the resulting 64 bytes.
- With the current setting encrypt the plaintext. Round key RK[0][1] is overwritten with the resulting 64 bytes.
- …
- With the current setting encrypt the plaintext. Round key RK[0][15] is overwritten with the resulting 64 bytes.

In total, 1700 (100*17) encryptions are engaged. The test value is the result of the last operation. The test values are stored in array DN_mega[MAXRHO][*c*], with the resulting *c* = 64 bytes of ciphertext for 1 to 10 rounds.

13.2. HDN test vectors

13.2.1. Test value HDN("abc")

The test values for 1 to 10 rounds are stored in array HDN_abc. They have to be identical to values in DN_abc_CONST1. Moreover, the program checks if array DN_abc_CONST0 is obtained as the inner state during the compression of the first block

13.2.2. HDN mega-test

Function HDN is called 100 times iteratively during this test. First, 64 byte hash code HDN("abc") is computed and padded to string "abc". The resulting 3 + 64 bytes are hashed again and padded again to its input. This procedure is repeated for 100 times. The final output is HDN("abc" || HDN("abc") || HDN("abc" || HDN("abc")) || ......))))))...). The test values are stored in array HDN_mega for 1 to 10 rounds.

13.3. DN a HDN testing program

Testing module main_test_definition_DN_and_HDN.c computes the entire test vectors for functions DN and HDN, the program speed is determined at the end; all for 1 to 10 rounds.

Note. Source codes are available on http://cryptography.hyperlink.cz/.