KUPY-NEEV HASH FUNCTION

Khushboo Bussi[1]

Department of Mathematics
University of Delhi
Delhi-110 007
India
e-mail: khushboobussi7@gmail.com

Dhananjoy Dey
Manoj Kumar

Scientific Analysis Group
DRDO, Metcalfe House Complex
Delhi-110 054
India
e-mails: dhananjoydey@sag.drdo.in
manojkumar@sag.drdo.in

B.K. Dass

Department of Mathematics
University of Delhi
Delhi-110 007
India
e-mail: dassbk@rediffmail.com

Abstract. Kupyna has been approved as a new cryptographic standard hash function of Ukrain in 2015 (Ukrainian standard DSTU 7564:2014). It is built on the transformations of Kalyna block cipher (Ukrainian standard DSTU 7624:2014). The design of compression function of Kupyna is nearly identical with Grøstl that makes it vulnerable to the similar attacks those were introduced on Grøstl. It is adapted for its highly secured design and its efficiency but lately Mendel et al. [4] and Zou et al. [14] mounted rebound attack on compression function of Kupyna-256 and reduced round collision attack on Kupyna hash function respectively. These attacks are applied on Kupyna because of its permutations based on AES. In this paper we propose Kupy-Neev hash function in which we change those permutations with the permutations based on *Neeva Hash* [1].

Keywords: Davis-Meyer mode, Grøstl, Kupyna, Neeva hash function, Rebound attack.

[1]Corresponding author.

1. Introduction

Kupyna is announced as a new standard cryptographic hash function for Ukrain in 2015. It has been chosen over GOST R 34.11-94 [6] which was used earlier for all security applications in nationwide before 2015. GOST R 34.11-94 was not efficient enough and couldn't provide required security hence a new hash standard was needed. Kupyna is quite efficient and gives high security. Due to these features, it had been started using in every practical scenario for cryptographic hash function. Kupyna uses Davis-Meyer compression function based on Even-Mansour scheme and its internal permutations are similar to the transformations of the Kalyna block cipher [11].

The compression functions of Kupyna and Grøstl [5] are very much similar which allow us to use the same cryptanalytic tools used in the analysis of Grøstl. Grøstl was chosen as one of the five finalists of SHA-3 competition because of its unique design and efficient software implementation. It is based on wide trail strategy. Mendel[10], [9] and Jean[7] had applied rebound attacks and collision attack on 5 rounds of Grøstl due to similarity of its permutations with AES and also applied collision attack on 5 rounds of Grøstl. The same attack strategy is used for Kupyna which makes it vulnerable to use in near future. This is the main reason to find an alternate of Kupyna. Thus, we propose a Kupy-Neev hash function[2] which has a different set of permutations and can provide better security.

This paper is organized in the following manner: In Section 2, Kupyna hash function is discussed briefly, we propose our Kupy-Neev hash function in Section 3 and in Section 4, the analysis of the proposed scheme is shown.

2. Kupyna hash function

Kupyna is an iterated hash function approved as Ukrain standard hash function. There are two permutations T^{+} and $T^{\oplus}$ used in the compression function of Kupyna which is very much similar to AES. In the following subsection we define the design of this hash function and the underlying permutations.

2.1. The hash function

Kupyna has two variants, Kupyna-256 and Kupyna-512 which give output of length 256 and 512 respectively. We will discuss here Kupyna-256 only. The input message M is first padded and then divided into the blocks $m_1, m_2, \ldots, m_k$ of length 512-bit. Here, Message is processed by $f(h_{i-1}, m_i)$ iteratively till all the message blocks of 512-bit get exhausted where f is the compression function and h_i is the chaining variable for $1 \leq i \leq k$. Once the updated 512-bit register h_t is obtained and then output transformation $\Omega(h_t)$ to calculate the final hash h. So,

$$\begin{aligned} h_0 &= \mathcal{IV} \\ h_i &= f(h_{i-1}, m_i) \quad \text{for } 1 \leq i \leq k \\ h &= \Omega(h_t) \end{aligned}$$

[2] *Kupy-Neev* is the proposed hash function in which the mode of iteration is same as Kupyna but underlying compression function is based on Neeva hash function [1]

The compression function f is based on two permutations T^{+} and $T^{\oplus}$ and is defined as follows:

$$f(h_{i-1}, m_i) = T^{\oplus}(h_{i-1} \oplus m_i) \oplus T^{+}(m_i) \oplus h_{i-1}$$

The output transformation Ω applied on h_t to find the final hash value h of size 256 where $trunc_n(x)$ is taking 'n' most significant bits of x.

$$\Omega(h_t) = trunc_{256}(T^{\oplus}(h_t) \oplus h_t)$$

The following figure is taken from the reference [12] shows the design of Kupyna hash function.

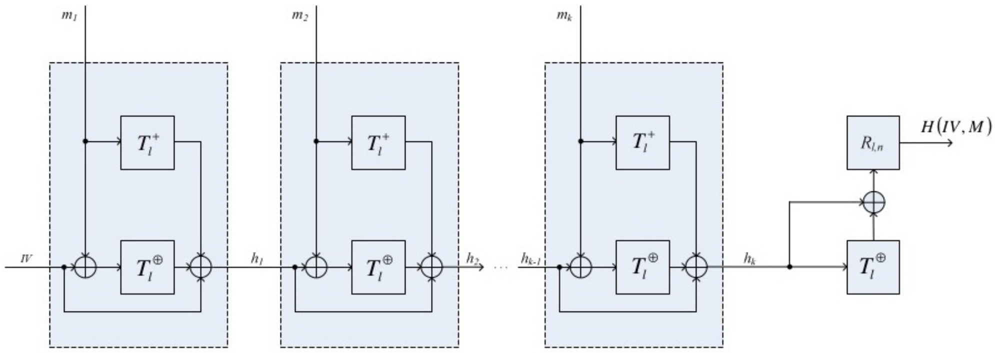

Figure 1: Kupyna hash function

2.2. The permutations T^{+} and $T^{\oplus}$

The structure of both permutations T^{+} and $T^{\oplus}$ is very similar to each other. In Kupyna-256 permutation updates on 8×8 state of 64 bytes in 10 rounds. In each round, the round transformations updates the state by means of the sequence of transformations viz., AddConstant, SubBytes, RotateBytes and MixBytes [4].

$$\textbf{MixBytes} \circ \textbf{RotateBytes} \circ \textbf{SubBytes} \circ \textbf{AddConstant}(\cdot)$$

The permutations are explained in detail with all transformations in the main document of standardization of Kupyna [11]. We describe it briefly in the following manner:

(i) **AddConstant:** In this transformation, the state is added by round constants. The column-wise modular addition mod 2^{64} is done with round constant which provides non-linearity in T^{+} but not in $T^{\oplus}$ (as round constant is xored column wise to the state). This is the only difference in both of the permutations. The round constants for $T^{\oplus}$ are defined as follows for round r, $1 \leq r \leq 10$ and column j, $1 \leq j \leq 8$:

$$\omega_j^{(r)} = ((j \ll 4) \oplus r, 00, 00, 00, 00, 00, 00, 00)^T$$

The round-independent constants for T^{+} for column j are given by:

$$\zeta_j^{(r)} = (F3, F0, F0, F0, F0, F0, F0, (7-j) \ll 4)^T.$$

(ii) **SubBytes:** In this transformation, the S-boxes are applied to each byte of the state. S-box of 8×8 (defined in [12]) is used for decent cryptographic properties. The detailed specifications is given in the main document. This step is same for both of the permutations T^+ and $T^\oplus$ but it is the only non-linear component in $T^\oplus$.

(iii) **RotateBytes:** In this transformation, the rows are cyclically shifted in such a manner that row j is shifted rightward by j byte positions, $0 \leqslant j \leqslant 8$. Even this transposition is same in both of the permutations.

(iv) **MixBytes:** In this transformation, the columns of the state get operated. A 8×8 circulant MDS matrix over $\mathbb{F}_{2^8}$. The coefficient of the matrix is chosen in such a way that the branch number of the MixBytes is 9 [12]. This step is also same for both the permutations T^+ and $T^\oplus$.

Permutations T^+ and $T^\oplus$ are almost same with all the transformations except AddConstants. This is the only difference in both of the permutations and this variation will make the structure different from AES and can resist all the direct attacks of AES like rebound and many more [4], [14], [9], [7].

3. Proposed Design: Kupy-Neev Hash

In this section, we present Kupy-Neev hash in which compression function is changed, keeping the mode of iteration of Kupyna intact. This is done because the majority of attacks applied on Kupyna [4, 14] are basically targeting the compression function, precisely its permutations i.e., T^+ and $T^\oplus$. So, in the proposed design we have changed those permutations with N and N^*. In this design, message is divided into the blocks of 512-bit and it has two more inputs i.e., initial variable and a counter. It will be called KN-hash now onwards. Now, it will be discussed in detail.

3.1. Padding

The padding procedure for the KN-hash is same as in Kupyna hash function. It takes a message m of b bits as an input. Each message follows an unambiguous padding rule irrespective of its length. Append 1 to the end of the message and add d '0' bits in such a way that d will be the smallest non-negative solution of the congruence relation $b+1+d+96 \equiv 0 \mod 512$, i.e., $b+1+d \equiv 416 \mod 512$ where the last 96-bit represents b in bits. The maximum length of the message is restricted up to $2^{96}-1$ bits.

3.2. Parsing

Once message m is padded, it will be divided into the 't' blocks of 512-bit length.

$$m' = m_1||m_2||\cdots||m_t,$$

where m' is the padded message.

3.3. Initial values

In the proposed scheme we need to have an initial value $\mathcal{IV}$ and a counter C_0 of lengths 512-bit each.

$$\mathcal{IV} = 1 \ll 510, \quad \text{i.e., } \mathcal{IV} = 2^{510}$$

The counter C_0 is

$$C_0 = \underbrace{000\cdots000}_{127-times}1.$$

3.4. Hash construction

An initial value $\mathcal{IV}$, a counter C_0 and the first message block m_1, each of 512-bit are taken as an input of compression function f and processed as following:

$$\begin{aligned} h_0 &= \mathcal{IV} \\ C_i &= C_{i-1} \boxplus_{512} 1 & \text{for } 1 \le i \le t \\ h_i &= f(h_{i-1}, m_i, C_{i-1}) & \text{for } 1 \le i \le t \\ h &= \Omega(h_t) \\ \Omega(h_t) &= trunc_{256}(N(h_t) \oplus h_t) \end{aligned}$$

where N is the permutation in KN-hash and h is hash digest of 512-bit which will be used as a initial value for next message block. The hash construction of KN-hash is almost same as hash design of Kupyna. The only difference is the extra input, i.e., a counter C_0 which wasn't there initially.

3.5. Compression function

Suppose we have a message, m and we need to evaluate KN-hash of this message. Firstly, it is to be padded and divided into the 't' blocks of 512-bit, i.e., $m_1||m_2||\cdots||m_t$. Message block m_1 xored with $\mathcal{IV}$, fed to permutation N. Also, m_1 xored with C_0, fed to N^*. After applying permutations N and N^* on those 512-bit registers, the output of N and N^* are xored with initial value as in the case Davis-Meyer construction [3]. This 512-bit output will be used as chaining variable for next message block and the whole process will be repeated. It will happen till all the message blocks get exhausted. The final 512-bit output h_t, after all message blocks are used up, is xored with $N(h_t)$ to get an updated register of 512-bit. The most significant '256'-bit of this register is taken as a hash digest of the message m. The following figure represents the compression function for the proposed scheme.

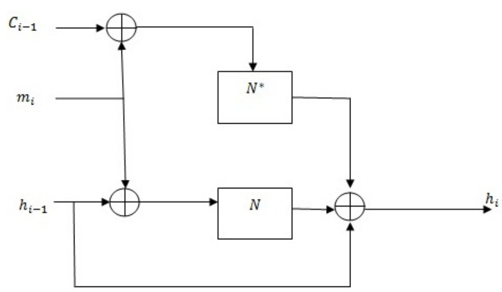

Figure 2: Kupy-Neev compression function

3.5.1. The permutations N and N^*

The N and N^* permutations act on 512-bit register. Both of these permutations are almost same but differs in Feistel type transformation. We will now describe the structure of N and N^*.

The permutation N is basically same as the permutation used in the *Neeva-hash* [1]. In N, firstly Present S-box (4×4) is applied on the register in parallel, then the register is divided in sixteen 32-bit words on which Feistel structure is applied on every 128-bit in parallel. In this Feistel type structure, the first, second and third word are updated by xoring with the fourth word and keeping the fourth word unchanged. This unbalanced Feistel structure is applied in parallel on four 128-bit strings of the register to provide word wise diffusion. After this a 16-bit left rotation is given and round constants are added under modulo 2^{32} to each round. The whole set of these transformations is named as u. It is applied 32 times for one application of N. Mathematically, we can write

$$\begin{aligned} u(x) &= (rotl_{16}(F(S(x)))) \boxplus_{2^{32}} RC_j \\ N(x) &= \underbrace{u \circ u \cdots \circ u}_{32-times}(x), \end{aligned}$$

where S is the Present S-box (4×4) acting in parallel, F is the Feistel structure defined above, $rotl_{16}$ represents the 16-bit left rotation and RC_j, $0 \leq j \leq 31$ are the 32 round constants written in the end of the paper. The permutation N is described in Figure 3.

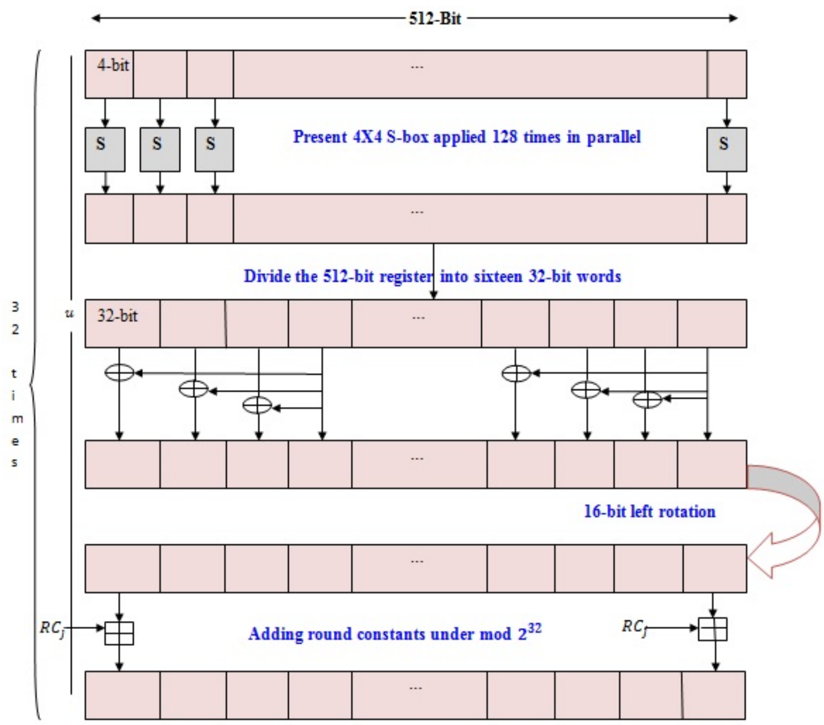

Figure 3: N Permutation

The permutation N^* is same as N except one step i.e., the Feistel type transformation is changed. Message block, m_1 is xored with C_0 and fed to N^*. After S-box application, the register is divided in sixteen 32-bit words and Feistel structure is applied on every 128-bit of 512-bit of register in parallel. In the four words structure, the second and fourth words are same and we just shift their places to first and third respectively whereas the first and third words are xored with second and fourth respectively then move to fourth and second place after a round. This balanced Feistel will act on the whole register four times in parallel. Again a 16-bit left rotation and modular addition are applied in every round till 32 rounds. The whole set of these transformation is named as v. N^* comprises of 32 times of v. Mathematically,

$$v(x) = (rotl_{16}(F'(S(x)))) \boxplus_{2^{32}} RC'_j$$

$$N^*(x) = \underbrace{v \circ v \cdots \circ v}_{32-times}(x)$$

where S is the Present S-box (4×4) acting in parallel, F' is the Feistal structure defined above, $rotl_{16}$ represents the 16-bit left rotation and RC'_j, $0 \leq j \leq 31$ are the 32 round constants written in the end of the paper. The permutation N^* is described in Figure 4.

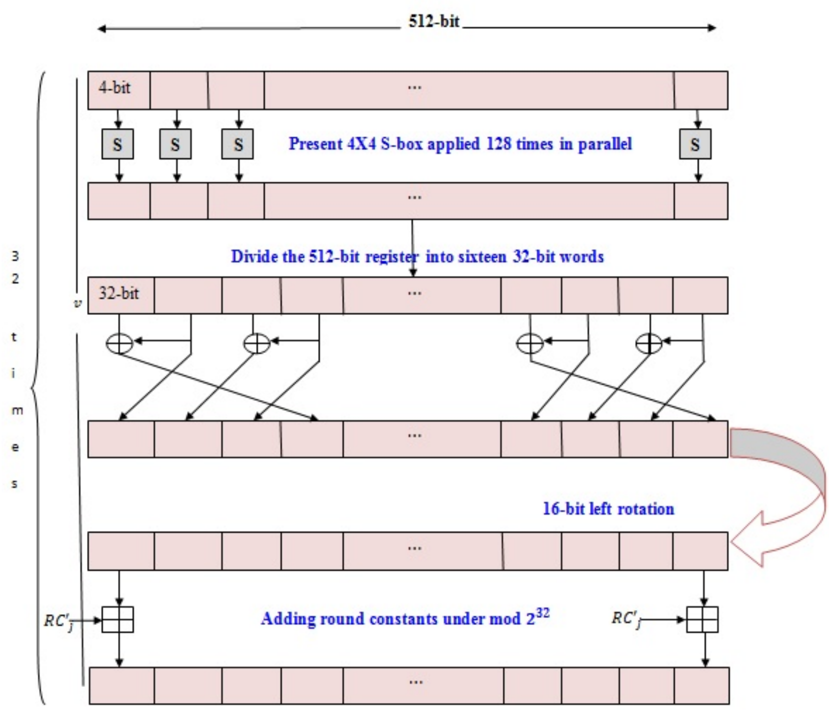

Figure 4: N^* Permutation

The step wise presentation of the proposed design is given in Algorithm 1.

input : $m_1, m_2, \ldots, m_t$
for $i = 1$ *to* t **do**
  $A_i \leftarrow h_{i-1} \oplus m_i$, $B_i \leftarrow C_{i-1} \oplus m_i$, where $h_0 = \mathcal{IV}$, C_0 *is counter*
  for $j = 0$ *to* 31 **do**
    $D_i^{(j)} \leftarrow S^{(j)}(A_i)$, $E_i^{(j)} \leftarrow S^{(j)}(B_i)$, $S^{(j)}$ represents application of Present S-box 128 times in parallel
    $D_i^{(j)} = d_0^{(j)} \| d_1^{(j)} \| \cdots \| d_{15}^{(j)}$, $d_m^{(j)} \rightarrow$ 32-bit words, $0 \leq m \leq 15$
    $E_i^{(j)} = e_0^{(j)} \| e_1^{(j)} \| \cdots \| e_{15}^{(j)}$, $e_m^{(j)} \rightarrow$ 32-bit words, $0 \leq m \leq 15$
    for $k = 0$ *to* 3 **do**
      for $\ell = 0$ *to* 2 **do**
        $d_{4k+\ell}^{(j)} \leftarrow d_{4k+\ell}^{(j)} \oplus d_{4k+3}^{(j)}$,
        $D_i^{(j)} \leftarrow d_0^{(j)} \| d_1^{(j)} \| \cdots \| d_{15}^{(j)}$,
      end
    end
    for $k = 0$ *to* 3 **do**
      for $\ell = 0$ *to* 1 **do**
        $e_{4k+2\ell}^{(j)} \leftarrow e_{4k+2\ell+1}^{(j)}$,
        $e_{4k+2\ell+1}^{(j)} \leftarrow e_{4k-2\ell+2}^{(j)} \oplus e_{4k-2\ell+3}^{(j)}$
        $E_i^{(j)} \leftarrow e_0^{(j)} \| e_0^{(j)} \| \cdots \| e_{15}^{(j)}$
      end
    end
    $D_i^{(j)} \leftarrow rotl_{16}(D_i^{(j)})$, $E_i^{(j)} \leftarrow rotl_{16}(E_i^{(j)})$
    $D_i^{(j)} \leftarrow D_i^{(j)} \boxplus_{2^{32}} RC_j$, $E_i^{(j)} \leftarrow E_i^{(j)} \boxplus_{2^{32}} RC'_j$
  end
  $h_i \leftarrow D_i^{(31)} \oplus E_i^{(31)} \oplus h_{i-1}$
end
return $A_t \leftarrow D_t^{(31)}$, $B_t \leftarrow E_t^{(31)}$ *and* $h_t \leftarrow A_t \oplus B_t \oplus h_{t-1}$
$h \leftarrow \Omega(h_t)$, $\Omega(h_t) = trunc_n(N(h_t) \oplus (h_t))$ where N is the permutation acting on A_i register.

Algorithm 1: Kupy-Neev hash function

3.5. Test value of KN-hash function

Test values of the three inputs are given below:

KN-hash(a)	=	$f9bd54c7$	$0220e770$	$ebabcc0f$	$87f93de4$
		$2fc762fa$	$10e83eb2$	$bc3513ca$	$88a825fd$
KN-hash(ab)	=	$967e0808$	$383147f4$	$ac97bf2c$	$4f5f2d36$
		$a4fab471$	$94aee45f$	$052e169e$	$f5c8991a$
KN-hash(abc)	=	$17097fab$	$e50bf15f$	$53868e47$	$609da983$
		$466f24c9$	$8eba82dd$	$a3ceaf82$	$0b6023a2$

4. Analysis of KN-hash

In this section, we show the analysis of the proposed scheme.

4.1. Efficiency

In this subsection, we discuss the efficiency of the proposed scheme. The following table provides the efficiency of KN-hash on an Intel core 2 duo 32-bit OS E8400 @ 3 Ghz processor with 1 GB RAM.

File Size (in MB)	**KN-hash** (in Sec)
1	2.758
5	13.636
10	27.677

4.2. Avanlanche effect

A 1024-bit input file M is taken and KN-hash of M, i.e., $h(M)$ is calculated. Now changing the i^{th} bit of the message M, 1024 new files M_i is generated where $1 \leq i \leq 1024$.

Also, the corresponding KN-hash of each of the M_i, $h(M_i)$ is calculated for $1 \leq i \leq 1024$. The hamming distance of each of the M_i from the M is exactly 1 and now we compute the hamming distance of $h(M_i)$ from $h(M)$, i.e., the hamming weight d_i such that $d_i = wt(h(M) \oplus h(M_i))$. The hamming distances in the corresponding 32-bit words of the hash value $h(M_i)$ and $h(M)$ is calculated for $1 \leq i \leq 1024$.

The results have been shown in the following table with the maximum, the minimum, the mode and the average value of distances mentioned above.

Changes	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	*KN-Hash*
Max	26	26	25	24	26	24	25	24	155
Min	8	7	8	8	6	8	9	7	107
Mode	16	16	15	16	16	16	16	16	129
Mean	16.00	16.08	16.14	16.01	15.85	15.92	16.01	15.96	128.06

Table 1: Hamming Distances

To satisfy the strict avalanche effect, change in one input bit should change the final hash value by = 50%, i.e., each d_i should be 128 for $1 \leq i \leq 1024$. It has been noticed from the above results that d_i's are lying between 107 and 155 and the most frequent value is $d_i = 129$ for $1 \leq i \leq 1024$. The mean value is very near to 128 (i.e., 128.06).

The following graph and table show the distribution of the 1024 files with respect to their differences (distance) in bits.

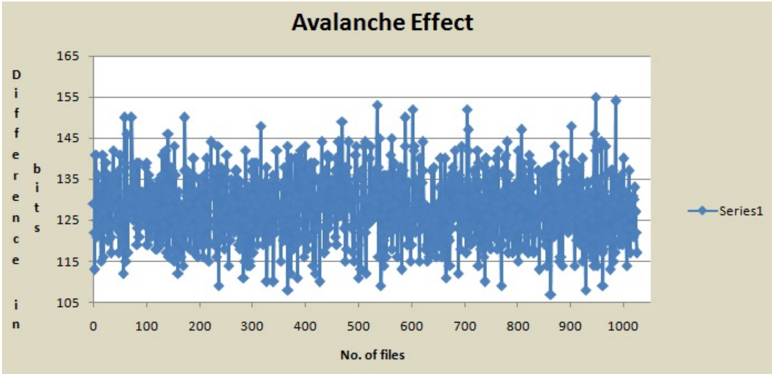

Figure 5: Hamming Distances range of the 1024 files

Range of Distance	No. of Files	% ***KN-hash***
128 ± 5	527	**51.46**
128 ± 10	831	**81.15**
128 ± 15	976	**95.31**
128 ± 20	1013	**98.92**

Table 2: Hamming Distances range of distances

4.3. Bit-variance test

The bit-variance test signifies the impact of change in input bits on the hash digest bits. Given a message, all the changes of the input message is taken up and then we evaluate the corresponding KN-hash digest of all the changes. Then, for each digest bit the probabilities of taking on the values of 1 and 0 are measured considering all the digests produced by applying input message bit changes. Bit-variance test measures the uniformity of each bit of the output. If the probability $P_i(1) = P_i(0) = 1/2$ for all digest bits $i = 1, \ldots, 256$ the KN-hash function has achieved maximum performance with respect to bit-variance test [8].

If we consider the 1025 files which we have used in testing the avalanche effect, viz. $M, M_1, M_2, \ldots, M_{1024}$ which we have generated for conducting avalanche effect, the following results can be found:

Digest length = 256
Number of digests = 1025
Mean frequency of 1s (expected) = 512.50
Mean frequency of 1s (calculated) = 512.08

The observed mean of 1025 files is found to be almost equal to expected mean. Hence, this hash function passes the bit-variance test. The following graph shows the probability of each of the bit (256-bit) to be '1' is approximately 0.50.

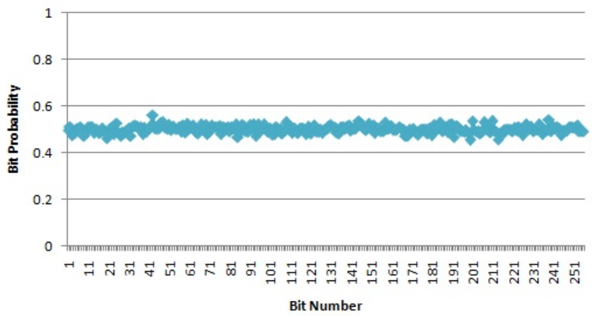

Figure 6: The probability of the bit position

4.4. Near-collision resistant

A hash function is called near-collision resistant if it is computationally hard to find two different inputs with hash outputs differ in small number of bits i.e., for this case if we have two different messages M and M', their KN-hash values shouldn't be almost same. Near-collision has been found in a hash function H means hamming weight of $(H(M) \oplus H(M'))$ (for $M \neq M'$) is relatively small (upto 16-bit).

To check with this, we have taken 100,000 files and evaluated xor of hamming distance of *KN-hash* of two random files from the lot. Two files are randomly chosen from 100,000 by $\binom{100000}{2}$ ways, i.e. 4,999,950,000 files. Analysis shows the minimum and maximum hamming weight of $H(M) \oplus H(M')$ is 79 and 179 respectively. The hamming weight 129 comes maximum number of times i.e. 249,065,436.

No. of files having the difference between and

$$(108 \leq \#files \leq 148)\ = 4,945,683,150\ (i.e.,\ 98.91\%)$$

It means almost 99% of files are between hamming distance 128±20 which is a good estimation with respect to the fact that they won't give a near-collision attack as for near-collision the hamming distance of two files needs to be really small viz. upto 16-bit. Hence analysis shows KN-hash function resists near-collision attack.

4.5. Differential characteristics

The differential attack [2] exploits input differences so the output can be controlled which can eventually help in finding a reduced round collision. It is a chosen plaintext attack where differences in input pairs are chosen to propagate a high probable differential trail. We try to find the expected number of input pairs that satisfy the differential trail which can be propagate to full fledged attack. Here, we try to find an upper-bound for the probability of such a differential trail.

In this proposed design Present S-box is used. Maximum differential probability for arbitrary input difference producing a output difference in a single S-box application is $\frac{4}{16} = 2^{-2}$ [13]. It means for only one active S-box in each round, differential attack still requires 2^{64} chosen plaintexts to distinguish the first 32-bit of KN-hash.

Now, we try to apply differential attack on KN-hash function. The compression function of KN-hash has two permutations. For computing differential characteristics, we replace modular additions in both the permutations by xor which is a weaker version of proposed scheme. Hence calculating the bound for this simplified hash would imply the minimum number of active S-boxes for KN-hash function. There are two independent permutations of the compression function of KN-hash. First we will calculate the number of active S-boxes in permutation N and then in N^* when one bit input difference causes the active S-box at the most significant byte of the word of updated register. In N, one bit input difference causes one active S-box at MSB in the first round and one more active S-box after second round. The total number of active S-box is 12 after 9 rounds. If we keep continuing in this manner till 32 rounds, the number of active S-box would reach about 50. In N^*, one bit input difference causes one active S-box at the end of first round and one more active S-box after second round. The total number of active S-box is 30 after 9 rounds. It has been noticed by computation, at the end of 32 rounds, the number of active S-boxes have come close to 280. After the 32 rounds, both the permutations are getting xored with chaining variable. The first 31 rounds of both the permutations are independent so their active S-boxes won't dependent on each other but the output of last round would get xored. Hence, the common active S-boxes will get cancelled in the final round. So, the minimum number of active S-boxes is $50 + 280 = 330$. Hence the maximal probability of finding a differential characteristic is $(2^{-2})^{330}$, i.e., 2^{-660}. That means, we require, 2^{660} chosen plaintexts to distinguish the most significant 32-bit of the output which is impossible to achieve. With this high complexity, it can be said that differential cryptanalysis is not applicable to simplified version of Kupy-Neev hash hence it will not applicable on KN-hash function.

Hence all the attacks which were applied earlier on Kupyna [4, 14] can not applied on KN-hash due to high complexity in finding a differential trail for rebound attack. We may say those attacks are not applicable on KN-hash function.

5. Conclusion

In this paper, we propose a modified version of Kupyna hash function, i.e., Kupy-Neev which provides a better security with respect to rebound attacks described in [4, 14](the complexity of getting a differential trail for rebound attack is very high making it impossible to achieve). It passes all the statistical tests viz. avalanche effect, bit-variance test and also ensures that differential attack can not be applied on it. It shows resistance to near-collision. It seems to be good substitute for Kupyna-256 and can be used for many of its applications.

Round Constants for permutation N

RC_1	=	2caff5a02f51df53603c5024c7aa47e8743f5e5c007f58784e838c3f6443ded4 86af048638676d354c1fe6775b977a92fb55b40321ec466457493abaf5aa2c92
RC_2	=	1abd61f1e854cd0081f848475c483623534dd75f52f3d3c026ec386eba9e2c0c 00654ecd261459d63fbb55298f21cfe49a27871e67606f7ec6d2152d38b1a2c1
RC_3	=	53034f24df8df1aac35d55ff8efae984207dc07ede317bf1a665331895b7066c 03e9b11a541d70b79702652a6ccd97ecfcb161022e5884ad763e86fe0abaff8f
RC_4	=	d8b60e2e1e754b46b5af065da4d92e4a5ab3c7e86aa4a7517e5a0e197651994e 6d63ff57caeede4290a17a42c80964e79c71033eb5dfb7938b727e1dfc25dafa
RC_5	=	9b85887a7b663471090a3d0f4624e7bc7134163499968645a6af1d219122cdd7 e5b8b484f42fb301328ac3d258209238a40c3a284e10d7f3cb6cb7b6f654dffc
RC_6	=	06ad769f77cd63f5c84e327a21ce6c514748ae4b426d7195d6976396eff5b44b 4ef29290fe7ca48b4d0995e13a6ab6423127434a8976267179debbef850c2f36
RC_7	=	063ee0437d6c66c276e172a1cbd67a61a90b596c5d83952ec22e0092f3e4911c 725a1b9712a76fcf85fd54b4b718521e4b17203998ac4bfd91f0d7f4fb25c8ca
RC_8	=	c39a7ad78e81e9242ccb41d1b1574574da8dbfaaa330359177677fbe0468b636 5308191d09aeffbb2e1f891cb8461290e5501950e3b576c5fd8573de3a5a7874
RC_9	=	ca2892ac52c4584c8e3630280db645997954e76247f5680d1859535c3a84e8f2 8a24d90247189c067007b6795de89f1d1c37fc0fcd711da9e40b9a60335c9f90
RC_{10}	=	49c6a9016e7853a0f40a494a01bc372cb49209d8dad27fde52349733c0e7b2c8 a82638cda99c3129f0115849feb8eaf7edbc2bef5035eef4d9f56f93b685474c
RC_{11}	=	34c22f4dcdebbcd2cd9beb565c5a69b698ff63fd9c070077c30b5dc0951bd993 d35c537a4c09943704407476af25befb8555191a2a51d7ed10c9c3a380f5a325
RC_{12}	=	c5fdc78c1ce337a04bccf1fb7df02469dbd2efd30b342ee205baa132e14ded71 9a12c4af77346d795af044b006ccf1d4f8ac3433d6334eb4c4dde7387974fc33
RC_{13}	=	37e3e6892fa89038918f05740a94e8a1cc41335ea7d326b304e13310d5c08714 07e85f2eccb417119998c0efa433689b34f06d1f87ddb37f7ae4c3b789e6bee4
RC_{14}	=	8da343159c772050af3fb3d5a0bcba355da09a454e3335bf16c985ab0f137e30 0a9dfd52b490473b0d58a1674f2b8255dbfdad86731891362726f02ebdf18a7e
RC_{15}	=	d3cfbb1f842b45e797b5742154159f1306389ad4eac215cc617d8385e685d4de 0b69427d64880996d9d06e87cf6a24d8529b373be56b066b04362c4c7b90f8ba
RC_{16}	=	ded56147b6eefe88b732a54a78964f81a4bc0febc447b7473df9b570aad7acd1 ae86a32afb47b5c79ff468dd9e1e6c5bd58aece5c87ac2ff11a39c583e2a06bd
RC_{17}	=	4b5ec16d05b88072669af65097954a415a905096b3a114f77ac737987e9352c4 660f3a7f2e204ce20df75d30b35d21e312381a59c9d65debf7cc82300d8563f1
RC_{18}	=	343718f2bf5baaaa76cdd610d946bebacbd95986c2985410413bc37ffbdd409d aeb4c1406f620f25c48ca325db43205d4b38a75ab5c598ccd1027ab2fc5e27eb
RC_{19}	=	114b44054fb96f4a8e664b24e92f2a2f1a1a6a5e81b3e63459d6527e763fb90e 27d85c423761b40778b9f19bb351b8a756b1d3a2eb304a08b4747b68d8b22416
RC_{20}	=	5c831813899762d920a441f1a62525a9a545ba96e0d71666e518248349b63c85 008aeba8c112abfc93cd31c00f5835e9105f698b6033100a0591179568d8e3ce
RC_{21}	=	f0bc90231e3443db2a9f54edf4365d8a0b2a25ddd5ccff0e0b8791f53656df51 a983eaa3b2e22fdb57060dc59eb02bdcc39946f5fb2f283043b80c9fc726d61c
RC_{22}	=	ccdda6b43c4d3a163df49e8a451c6221146582965a146cd63e4940ffa98b3860 45a06da94ddf2b56c65684a5571078bf62050e229650c88c981c6f09b5a7556f
RC_{23}	=	d79c0cb92356316b733ee306fb43afddbd5a8d2acd73e5ec611f2e2699611ca4 42b9d41bbf7f1fe683d7c9687e2105d3fec8a8c86c6b4fbe8a3b959bef19752d
RC_{24}	=	22a0dd925bc3c47112cadf83b5517d41e4680f82e12cab8a0c0113e2d0bcc2c4 c533f5878b0fe6c4e94c2cc07a4edc96a0049dba50a93ee9b31f7b8555d54d56
RC_{25}	=	f7cd8d5dc9bb363f32ba80f767981a49e03f15ce47ea7b1f81bd7c12f68ccf7b 1984586276fdfd57e48c56c1835e8c7e86df86fff2819b0724c3f3ffe0d7987c
RC_{26}	=	8bd0f2eaab2cafab5764d382877195ac463a7b0a74876e886497168f74669164 8d24ab227d92b420458e47fd7c32e95b2a5298120e34b87d5be2dc977ee5b03e
RC_{27}	=	8b28210e7343e8c090600f6477e1e78e0de05da55bf7475cff48f737b6a0bdaa 11db94422310a0357777311e10ba51110641da043640b0ee9b057a796dd81e92
RC_{28}	=	550e1c62c3515afa955c7d8e5331b71980e2f651476436bf06c581c37384cfed 35a1c76e41628dbc02badeffadb5268c975df8dfcb169953931aca87828bedce
RC_{29}	=	34979f57c626c8876a89f053067ff126ff0212db7b40adc8378b327f02dd079f 51e61a064a419166684975f76a3797c8d30b53afe1dca48be231c72ff4b4f02b
RC_{30}	=	5ff130189f819a6ed9ec077a66564f97bcfef8f77d7ae0166f4f4c5c4fd64aa4 fb22707a028838e8984b04dd18547be14a8e65f9ef6a4ec020fdd3ecf704d419
RC_{31}	=	295f8412ef720ab64196b275a7322cdd488de46b9e9b9f16f3bbdbaa30aecc4a 7081d36afde5c9844bc55105108600297a1f4cce42bb2f8a328841bc84f78df7
RC_{32}	=	7fa1c85e639e2abda286fc8fd3e9301ee440c058c4bc20de4f67c21bfb4c8c4f 3206bf0a4637a18cd3e47cb8c7a57c008d849a966c31ce92d37ba84477325279

Round Constants for permutation N^*

RC'_{1} = 5d046a2baf8903c2c0796fa29af2b7cafab0747519f905d8c68cd1bf489b12ac
d33d5537869d31f76cac00cc6cbba30aeb67f04e63d241a09b13a10ed1fd7e6a

RC'_{2} = d25c6530098a276988f9819ab2cd1ac144db4a288945aff231b24b849f5baf5f
a17f3262275680a2f31b6be63663f1e6e33b8aff712bfdacffbb7e2901be9114

RC'_{3} = a25749ffe4bf76edb812eac99d8d8bfaae147202115cebdb56e9d2299804ceaa
5e5d19b652c6eb6468ebf0df5ebe4e47d21143fd68a1f002d2f1fdd70f1a61fa

RC'_{4} = 8fc6f48a1cf4c21c3f3f98143c8f59718d674a0ad7a93a62cf3ef3dcc7129579
9687589ea64c7bf47e91651a27f62593503dd5242ac33da26c3c410245d8e1c4

RC'_{5} = 6d86d2f53b56674719f84bd16d3815be975ccea1020f62e11f2565866ad7af93
c8f2b6a49aca4c7c550827fb6b96521bc628c9ad18930d8f79f2e3a21abf5328

RC'_{6} = d2c1875ad58639a87923a6213c753a18fc4d0d46a5f741015c9ed00539a9b145
62a59d725a307154fbce7af0dd0baf86da89846be1d445cc21fce5a08517a660

RC'_{7} = 0221d0cc755151351edb0b5d51d3c1a07f8fc42d47ca4f95b57cae52a215acad
1cdc0258e4df1aede455da4212901ff9f18df3c613cf8dea37a48f2b4f666912

RC'_{8} = 239d297416bb2ce3d812fb4aef59800b15df5ed99e05d3b6d3f07c74a07fba13
730364437c412b9dc52a0f4c1aef045dbe459fdb75857cda57cbdbd3113ddf0a

RC'_{9} = a0b2865c9fd45d09b046c1b67b381066f46677c6a6f70822e889587f14b26731
99fb9c5c6ec4956324696ac9d3caad443ab50ac29a01618c5675ce5a6f51931d

RC'_{10} = db42e8fbf9565f052b02e02dd6ab60729fdfcc216324f538822d33b991be439e
fca0943bd1d9b69e6405c3fe9a7a979aa6a477c8eec930b7ca26d71bce8ceb12

RC'_{11} = fa0df8d26ba2e9d6e047f123640dac1ecc85ba10b6efa98edeb5f41982ea98f8
35a26d838997ce646bead97ddb40cc46261e513a9cbf7238dfb672c73e7d5872

RC'_{12} = 041f785a090566b19d90a4c574f6543ef28b088f996b5819929b25cb9c630b04
416ce2218fdbac3a2b9639313e7178bcb2a6957724d4eacae80ee6563081e018

RC'_{13} = efd9d5f241dca04aa090aca4392d35d4337d1c8b36eae3b3afb240832025e007
6811b601c5593d15cc6aa5a75d254256b4403e08c67e2c1346cde0544ceda29c

RC'_{14} = e043b75fa236ab026afdce5b864422d3c40d8f951d0e4936a39bbef9137fe0f7
8623e4f2b4f4c93080d30ddb26589d1a664603ede1fe262607c2168c0725cd83

RC'_{15} = cd749fe0f6b17d01ec8c68a870948b997f313f1e3b1f30131f04c665cd64abca
e6cf4f2f00a632747a7fa1fa0540c9b75db887173e3a2a38adde17cec958ab18

RC'_{16} = 97d1270874df951188ac3cc51ec172b891774f165150c6e86c147aa7e70195d7
318a467e09b8d359faa1d85d7bbe1d85a60b7f59d90d9c42654e37a7b63e5197

RC'_{17} = 5282219265d0fd08efddfdd6a1df2fdbddb3b10df401e99083aecd996c45c85c
8a85cf10c512dc0107d77bf9cba1492ed191d34b09221de22a86c3f234801f18

RC'_{18} = 67f9280d67a31a356001771d48278681c33238d980574689ab3874283da8b41c
74c17614365f9b9b0be5e52cdb3e5a9cf8f5e948074cd7059c6d197570c77966

RC'_{19} = 5dc4434aba58dbca52f0dea0bf1ee7df4b9c52400b955298c8892030c3d6bcb3
3eaa484994b3075dd4cc358bc68391208943ea7466959241a80759c772c2da5d

RC'_{20} = 7e64e8bfa0221b2bbb8cc97ab13933d116b11fc5ff5e37551e5292efe9b3541e
98b57d5d92a84ad6b8a541f51ce8674dd37f88a2a605ff1b7167e6b925e5beb4

RC'_{21} = a42fdb4232327996be8449fdebbc4a67d59ae460a5c46fc02edc6a75c5efc454
fd33720e6fc50e66c847ac1b7accdeb7d44e9e913732c7c1c1f9c4361455c566

RC'_{22} = c16c0f5e2162475cef9a47a14519a45bef52579e437f3a0eefdbd09eedc51f9b
3a9bcf5cbc592d387e8c9705320aa38611ab6475dbbc3b75d63274a4a3117246

RC'_{23} = eb3e1c4786e7aab3010d31c8293eeff0ca0fac0c804360d4bb2c8133a72f2706
2e7f6ba885cd03067bf9e2eb792ecf24cb641ffecf0c7231c128b026a67b1572

RC'_{24} = 58eef3b65d891e515986ae9eab7da8bd21d5d3274776d2fcb23e6e79a06f7e2e
0ed7bd93afe9e2777208820b8f2b5645fe4083524e48abad20a0d76bfdeb4b20

RC'_{25} = b29f4c02aa1f90d9c8c29f4521358641526b2ac83da18d8d0d628fc18ff49f7e
bbdac636c6f945cd2ad68c30580d77f526fb92300f07090578dec15aefc46d52

RC'_{26} = 81c87e3026252418a1b0cb1f1c0c17012fd21e85428855c1179f567b96fb4ca5
8c25dcab75ba157b2d26cc47cd8d5986e874f26503d632fdc29a74662993b266

RC'_{27} = 650422cab1172d382927dac42dc8f5adae3d51ee4b772519896c48bc5b3a1947
0074d912ff4ea04d570e1ab927b8b8ca0c07aebc48ee5fd68c091b34d439d711

RC'_{28} = 5771e887a1b39882d45a47c75c70087976064ca213de7291db73cc4b6ad3d205
0adab0a1369e7c230c3a95624794596656c3905aa5dc86b12b5e8b8953b951ab

RC'_{29} = 53d8a133925ce4e33a7f1d570e2e494482ae750c59617de406905a5d2ca8b99e
39b3af8eb08695f8f67e3bb39e1bd08aeae3dcfcf5b9d16982024bb50e917b52

RC'_{30} = 429e558e968d4b38fdf18ec099f1ed41c089c1c450798ff8d1125c48880648a3
86cd56c906620f6534e05b0528a97c1f005b8badf6406336f2734d1e52d2f2e7

RC'_{31} = 7eb11ffbf4d71d488b4e0d81dc7b03411c28ad0dfcb62db5ae4be8769c4a0542
fb175654eb6082c3b4836f41f192334429be0cf2ae81655ffb2ea6ee2f0d0a9e

RC'_{32} = 5ae98f187afdc72060f3e9e4997385ad1e3aa44a02420454d2430966dc3648bd
7237fc80b1ce14902b9aca152fdd8545fbaf40cde4aa7daf9e0f7034e4a9cb1d

References

[1] Bussi, K., Dey, D., Kumar, M., Dass, B.K., *Neeva: A Lightweight Hash Function*, IACR Cryptology ePrint Archive, 2016 (042). Available online at https://eprint.iacr.org/2016/042

[2] Biham, E., Shamir, A., *Differential Cryptanalysis of the Full 16-round DES*, Advances in Cryptology- CRYPTO'92, LNCS, Vol. 740, Springer, 1993.

[3] Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P., *Merkle-Damgård Revisited: how to Construct a Hash Function*, Advances in Cryptology, LNCS, Vol. 3621, 2005.

[4] Dobraunig, C., Eichlseder, M., Mendel, F., *Analysis of the Kupyna-256 Hash Function*, IACR Cryptology ePrint Archive, 2015 (956). Available online at https://eprint.iacr.org/2015/956

[5] Gauravaram, P., Knudsen, L., Matusiewicz, K., Mendel, F., Rechberger, C., Schlaffer, M., Thomsen, S., *Grøstl- a SHA-3 candidate*, submission to NIST (Round 3), 2011. Available online at http://groestl.info

[6] Government Committee of Russia for Standards. GOST 34.11-94. Information technology. Cryptographic Data Security. Hash function. Government Committee of Russia for Standards. Moscow, 1994. (In Russian).

[7] Jean, J., Plasencia, M.N., Peyrin, T., *Improved Rebound Attack on the Finalist Grøstl*, FSE, 2012.

[8] Karras, D., Zorkadis, V., *A Novel Suite of Tests for Evaluating One-Way Hash Functions for Electronic Commerce Applications*, IEEE, 2000.

[9] Mendel, F., Rijmen, V., Schläffer, M., *Collision Attacks on 5 Rounds of Grøstl*, FSE, 2014.

[10] Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S., *Rebound Attacks on the Reduced Grøstl Hash Function*, CT-RSA, 2010.

[11] Oliynykov, R., Gorbenko, I., Kazymyrov, O. et. al., *A new encryption standard of Ukraine: The Kalyna block cipher*, Cryptology ePrint Archive, 2015 (650). Available online at https://eprint.iacr.org/2015/650.pdf

[12] Oliynykov, R., Gorbenko, I., Kazymyrov, O. et. al., *A new standard of Ukraine: The Kupyna hash function*, Cryptology ePrint Archive, 2015 (885). Available online at https://eprint.iacr.org/2015/885.pdf

[13] Wang, M., *Differential Cryptanalysis of PRESENT*, 2007. Available online at http://eprint.iacr.org/2007/408.pdf

[14] Zou, J., Dong, L., *Cryptanalysis of the Round-Reduced Kupyna Hash Function*, IACR Cryptology ePrint Archive, 2015 (959). Available online at https://eprint.iacr.org/2015/959

Accepted: 21.06.2016