Defence Science Journal, Vol. 67, No. 6, November 2017, pp. 636-644, DOI : 10.14429/dsj.67.10791

HeW: A Hash Function based on Lightweight Block Cipher *FeW*

Manoj Kumar[#,@,*], Dhananjoy Dey[#], S.K. Pal[#], and Anupama Panigrahi[@]

[#]Scientific Analysis Group, Delhi - 110 054, India
[@]Department of Mathematics, University of Delhi, Delhi - 110 007, India
[*]E-mail: manojkumar@sag.drdo.in

ABSTRACT

A new hash function *HeW:* A hash function based on light weight block cipher *FeW* is proposed in this paper. The compression function of *HeW* is based on block cipher *FeW*. It is believed that key expansion algorithm of block cipher slows down the performance of the overlying hash function. Thereby, block ciphers become a less favourable choice to design a compression function. As a countermeasure, we cut down the key size of *FeW* from 80-bit to 64-bit and provide a secure and efficient key expansion algorithm for the modified key size. *FeW* based compression function plays a vital role to enhance the efficiency of *HeW*. We test the hash output for randomness using the NIST statistical test suite and test the avalanche effect, bit variance and near collision resistance. We also give the security estimates of *HeW* against differential cryptanalysis, length extension attack, slide attack and rotational distinguisher.

Keywords: Block cipher; *FeW*; Lightweight block cipher; Wide-pipe construction

NOMENCLATURE

Br_i	16-bit branch
MK	64-bit master key
MK_i	16-bit word
rk_i	16-bit subkey
rF	Round function
$rk_i^j(k)$	32-bit round key
F	Compression function
$\oplus$	Bitwise exclusive-OR operation
$\lll n$	Left cyclic shift by n bits
$\ggg n$	Right cyclic shift by n bits
$[i]_2$	Binary representation of integer i
RC	Round constant $[i]_2$ for round i
$\|$	Concatenation of two n-bit strings
&	Bitwise AND of two n-bit strings
$B \leftarrow A$	A is transformed to B

1. INTRODUCTION

Last two decades will be commemorated as a revolutionary period in the field of information technology. There is a sharp increase in the usage of internet in mobile applications and shopping through e-commerce portals. We need to secure the internet data traffic to boost the confidence of common people and thereby achieving the dream goals like digital India movement[1] by Government of India. Hash function plays an important role in authentication of data traffic over the internet. Hash functions are mainly intended to ensure the integrity of data in cryptographic applications[2]. But there is other usage of hash functions in speeding up the search of data in look-up tables[3]. Hash function takes an arbitrary length input message and converts it into a fixed size output[4]. The outcome is known as the message digest and works like a thumb print for the intended message. Any single bit difference in the input should result in approximately 50 per cent change in output bits.

Hash functions were introduced by Diffie and Hellmen in 1970 and most of the hash designs were based on block ciphers. The first hash function was based on block cipher DES[5]. There are hundreds of new hash functions published since their evolution[6,7]. The widely used hash functions are MD5[8,9] and SHA-1 family[10]. NIST announced SHA-3 competition for selecting a secure and efficient hash function. In 2012, sponge based construction Keccak was selected as SHA-3 standard[11]. The design of hash functions can be divided into three categories: hash function based on block ciphers, hash function based on arithmetic functions and dedicated hash functions[12]. The majority of cryptographic hash functions lies in dedicated hash function category.

In the process of designing a secure and efficient hash function, we should make use of the cryptographic components that are well reviewed over the years as well as efficient to implement in software and hardware[3,13]. Block ciphers have a long fascinating history and data encryption standard (DES) is the first established block cipher. There are much clear security definitions to prove the security claims for a block cipher and we can utilise the design and evaluation effort of a block cipher[5]. Therefore, we have used the lightweight block cipher *FeW*[14] in the compression function to increase the efficiency without compromising the security. Since, the key expansion algorithm in block ciphers is not designed very carefully, it

Received : 17 October 2016, Revised : 29 March 2017
Accepted : 08 August 2017, Online published : 06 November 2017

may lead to an attack on block cipher based hash function. We need a strong key schedule for the block cipher which can be used to design a compression function. Therefore, we modified the key size of block cipher to 64-bit and provide a stronger key expansion algorithm for *FeW* used in *HeW*.

2. LIGHTWEIGHT BLOCK CIPHER: FEW

FeW is a lightweight block cipher with 64-bit block size and 80/128 bits key size proposed by Kumar[14], *et al*. It is based on Feistel-M structure which is an admixture of Feistel and generalised Feistel structures. *FeW* is designed to achieve high efficiency in software based applications. Nemati[15], *e. al.* have illustrated that *FeW* can be implemented in hardware with very small area requirement. It suggests that *FeW* can also be applied in hardware based platforms.

We now briefly discuss the round function and key expansion algorithm for 64-bit key. Swap function is used after 32 rounds of each iteration.

2.1 One Round FeW

We divide the 64-bit input block into four branches Br_1, Br_2, Br_3 and Br_4 of size 16-bit each. Round function rF takes Br_3, Br_4 and 32-bit round key as input and produces the 32-bit output. Most significant 16 bits of the output are XORed with Br_1 and least significant 16 bits are XORed with Br_2, which gives the new values of Br_3 and Br_4 for next round. Old values of Br_3 and Br_4 remains unchanged and these are the new values of Br_1 and Br_2 respectively for next round. One round of *FeW* is shown in Fig. 1.

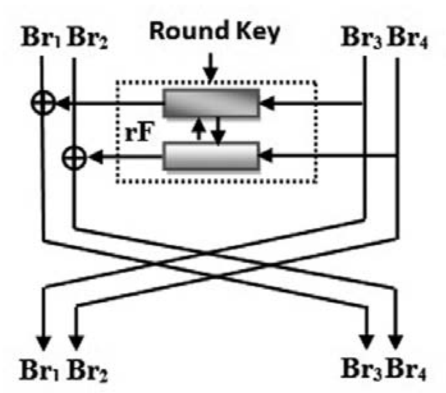

Figure 1. $\boldsymbol{FeW_{1R}}$.

2.2 Round Function (rF)

Round Function takes 32-bit input X_i in the form of two 16-bit Feistel branches. First, these 2 branches are XORed with two 16-bit round subkeys. Thereafter, it mixes the data between Feistel branches by swapping the least significant bytes of the two branches. Then, S-box *S*(Table 1) is applied 4 times in parallel on each branch. Finally, there is an application of two different permutation layers on each branch. We get the output Y_i from rF. Round function of *FeW* is shown in Fig. 2.

Table 1. S-box (S)

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

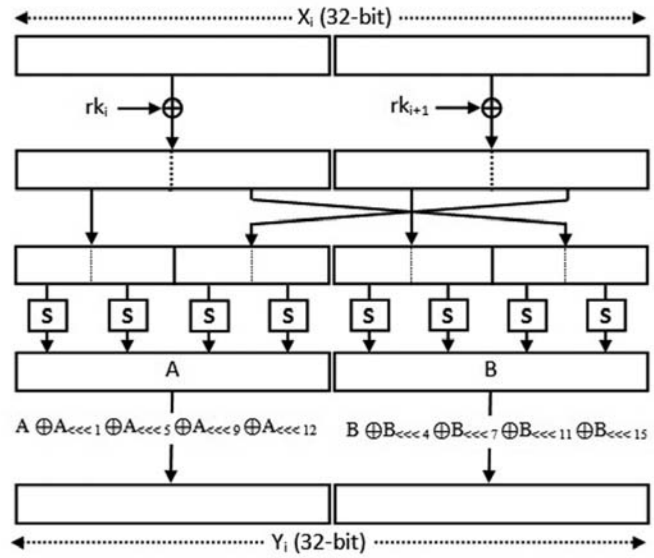

Figure 2. Round function *rF*.

2.3 Key Expansion Algorithm (FeW_{KE})

Block cipher based hash function treats the input message as a key for the underlying block cipher used in the compression function. Any tiny weakness in the key expansion algorithm can lead to a serious attack on the hash function, so we need a stronger key expansion algorithm. We reduce the key size to 64-bit and present the key expansion algorithm of *FeW* for the 64-bit key which is much stronger than the key expansion algorithm for 80-bit key. We use the modified version of *FeW* to design the compression function of *HeW*. We write the 64-bit master key MK as a concatenation of four 16-bit words MK_1, MK_2, MK_3, and MK_4. Current contents of MK_1 is stored as the first 16-bit round key. Key register is updated using S-box and cyclic shift. S-box is applied on most significant 4 bits of MK_1 & MK_4 and least significant 4 bits of MK_4 while the middle 8 bits of MK_4 is XORed with a round constant RC. Finally, the 64-bit register is left rotated by 13 bits. After updating the key register, current contents of MK_1 is stored as the subsequent 16-bit round keys. Key expansion algorithm for 64-bit key is given in Fig. 3.

2.4 Swap Function

We have 64-bit output after processing the 64-bit input message and the 64-bit key in each round. After 32 rounds, swap function is used to exchange the current contents in the least significant 32 bits and most significant 32 bits.

3. MERKLE-DAMGÅRD AND WIDE-PIPE CONSTRUCTIONS

There are many approved hash construction methods which can be used to design a hash function based on a block cipher[15-17]. Merkle-Damgård is the basic construction method which is used by the majority of hash function designs[18]. This method uses only one compression function f to compute the hash digest. After padding the arbitrary length input message, it processes the b-bit message block and n-bit $\mathcal{IV}$ as input and generates the n-bit hash digest after processing all message blocks iteratively.

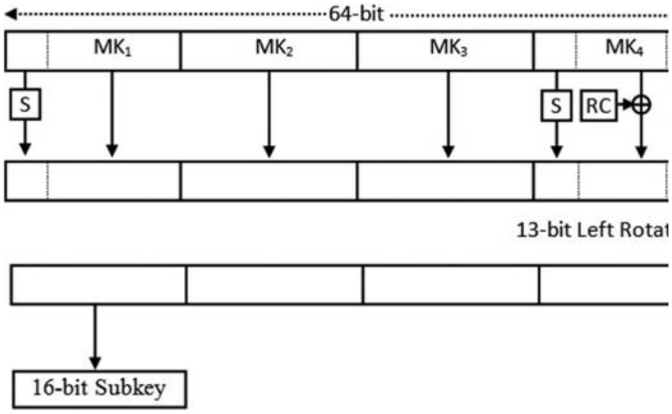

Figure 3. Key expansion algorithm.

$$f: \{0,1\}^n \times \{0,1\}^b \rightarrow \{0,1\}^n$$

Wide-pipe construction was proposed by Stefan Lucks[18,19]. This method was proposed to counter the weaknesses in Merkle-Damgård construction which was prone to the length extension attack. This method uses two compression functions f and g to compute the hash digest. After padding the arbitrary length message, first function f is used to iteratively process the b-bit message block and w-bit $\mathcal{IV}$ to generate w-bit output. After processing the complete message, second function g takes w-bit input to generate the n-bit message digest.

$$f: \{0,1\}^w \times \{0,1\}^b \rightarrow \{0,1\}^w$$
$$g: \{0,1\}^w \rightarrow \{0,1\}^n$$
$$where\ w \geq n$$

4. PROPOSED HASH FUNCTION: HEW

We use Wide-pipe construction method to design our proposed hash function *HeW*. Message block size and chaining variable size are to be of same length ($2n$-bit) to generate the n-bit hash digest. Compression function takes two inputs (512-bit message block m_i and 512-bit chaining variable h_{i-1}) and outputs a 256-bit hash digest, where initial value of chaining variable is fixed as $h_0 = \mathcal{IV} = 0^{512}$.

4.1 Padding Rule

HeW iteratively processes the 512-bit input message blocks. The length of input message may not be a multiple of 512, so we need to pad[20] the arbitrary length input message to make it a multiple of 512. If the message length is a multiple of 512 then we add one dummy padding block to the message. Suppose length of an input message M is ℓ bits. We append the bit '1' at the end of message M, after that we append $(-\ell - 2) \equiv k\ mod\ 512$ '0' bits and finally the bit '1' is appended at the end of padding. We now have a padded message m whose length is a multiple of 512.

4.2 Parsing

We divide the input padded message m in t blocks of size 512-bit each as follows:

$$m = M \,||\, Pad\ (M) = m_1 \,||\, m_2 \,||\, ... \,||\, m_t$$

We process one 512-bit message block m_i at a time iteratively.

4.3 Compression Function

In each iteration of compression function F, we process the 512-bit message block m_i by dividing it into the eight 64-bit words $m_i^j : 0 \leq j \leq 7$. There are eight parallel applications of *FeW* inside F and these 64-bit words are used as key. For each 64-bit word, we apply key expansion algorithm FeW_{KE}. We get 32 round keys of size 32-bit each corresponding to the one 64-bit word. In total, we generate 256 32-bit round keys for eight 64-bit words. We divide 512-bit chaining variable h_{i-1} into eight 64-bit words $h_{i-1}^j : 0 \leq j \leq 7$. We take these 64-bit words as input messages to the eight applications of *FeW*. FeW_{1R} is applied using round keys $rk_i^j(k) : 0 \leq j \leq 7$, $1 \leq k \leq 32$ and message $h_{i-1}^j : 0 \leq j \leq 7$. After each round, 512-bit register is rotated left by 16 bits. After 32 rounds, FeW_{SWAP} is applied on each 64-bit word. After processing the last 512-bit message block, the most significant 256-bit is stored as hash digest of the message. Figure 4 gives the processing of one message block using *HeW*.

4.4 Hash Construction

Compression function of *HeW* takes chaining variable h_{i-1} and message block m_i as inputs in each iteration. Compression function updates the chaining variable to h_i after each iteration. After processing all of the t message blocks, the most significant 256 bits are received as the hash digest for the input message M as follows (Algorithm 1):

$$h_0 = \mathcal{IV}$$
$$h_i = F(h_{i-1}, m_i) \qquad for\ 1 \leq i \leq t$$
$$Hash(M) = trunc_{256}(h_t)$$

5. ANALYSIS

Software and hardware performance of *HeW* is presented here. We also discuss the statistical analysis of *HeW* and differential cryptanalysis, length extension attack, slide attack and rotational attack on the compression function of *HeW*.

5.1 Software Performance

We have used an Intel(R) Core(TM) *i*7-3770 CPU @3.40 GHz processor with 8 GB RAM and 64-bit operating system for benchmarking. We run the code of *HeW* and SHA-256 several times for three different size data files and calculated the throughput as average running time in MB/Sec. We show the performance comparison of *HeW* and *SHA-256* in Table 2. The results indicate that *HeW* performs better than *SHA*-256 in software.

5.2 Hardware Performance

Nemati[15], *et. al.* have illustrated that lightweight block cipher *FeW* is quite efficient for hardware oriented applications. It is shown that *FeW* can be implemented

Table 2. Software performance

File size (MB)	*HeW* (s)	SHA256 (s)
1	0.227	0.352
5	1.127	1.738
10	2.238	3.471

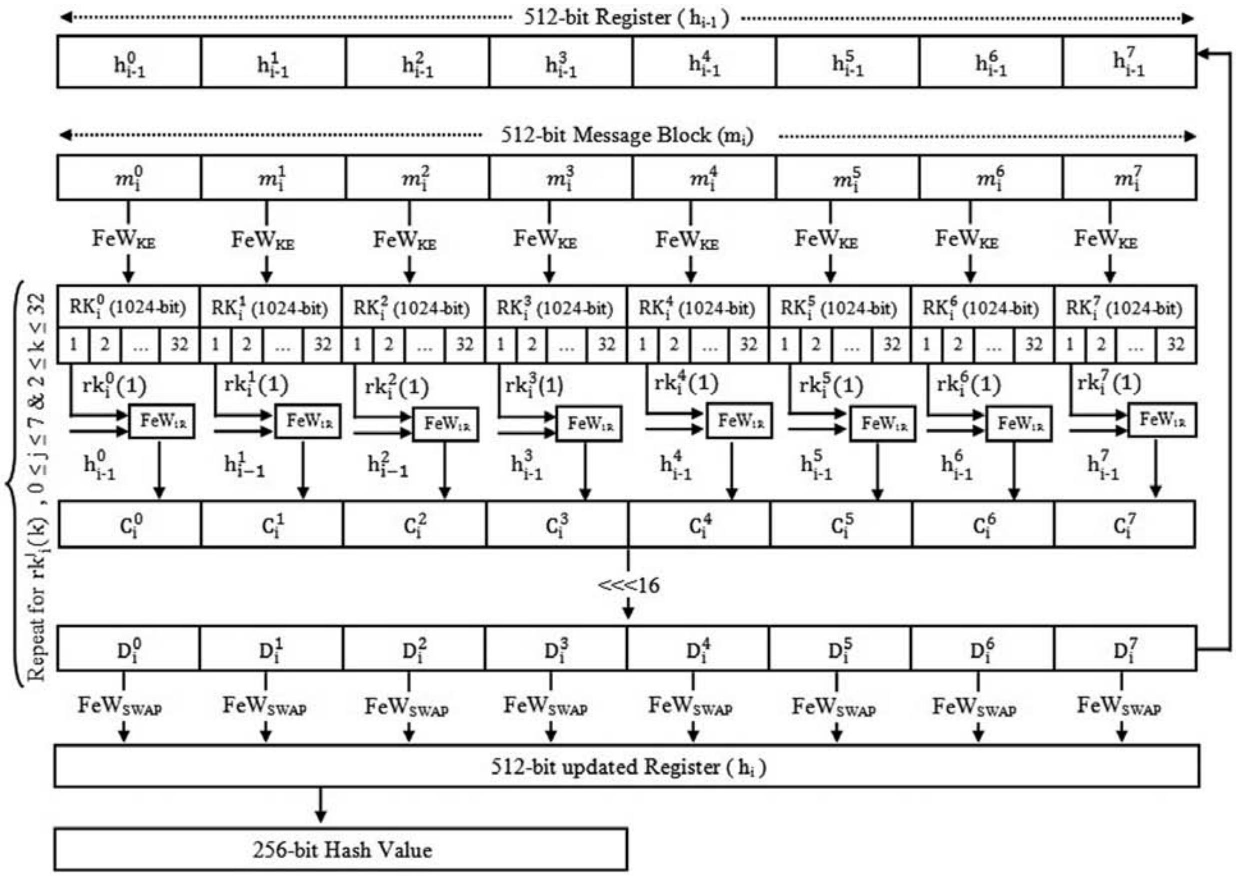

Figure 4. Compression function *F*.

input : $m_1, m_2, \ldots, m_t$
for ($i = 1$ *to* t) do
$\mathcal{IV} = 0^{64} \| 0^{64} \| \ldots \| 0^{64}, \quad h_0 = \mathcal{IV}$
$(h^0_{i-1} \| h^1_{i-1} \| \ldots \| h^7_{i-1}) \leftarrow h_{i-1},$
$(m^0_i \| m^1_i \| \ldots \| m^7_i) \leftarrow m_i,$
for ($j = 0$ *to* 7) do
$RK^j_i \leftarrow FeW_{KE}(m^j_i),$
$RK^j_i = rk^j_i(1) \| rk^j_i(2) \| \ldots \| rk^j_i(32)$
end
for ($k = 1$ *to* 32) do
for ($\ell = 0$ *to* 7) do
$C^\ell_i \leftarrow FeW_{1R}(rk^\ell_i(k), h^\ell_{i-1})$
end
$C_i = C^0_i \| C^1_i \| \ldots \| C^7_i,$
$D_i \leftarrow rotl_{16}(C_i)$
$D_i = D^0_i \| D^1_i \| \ldots \| D^7_i$
$(h^0_{i-1} \| h^1_{i-1} \| \ldots \| h^7_{i-1}) \leftarrow (D^0_i \| D^1_i \| \ldots \| D^7_i)$
end
for ($j = 0$ *to* 7) do
$h^j_i \leftarrow FeW_{SWAP}(D^j_i),$
end
$h_i = h^0_i \| h^1_i \| \ldots \| h^7_i$
end

Algorithm 1. Hash construction

in hardware with very small area requirements. It will be practically implemented using 125 number of slices and 264 look up tables (LUT). We have used *FeW* eight times in parallel in compression function of *HeW* with reduced key size (64-bit). Reduction in the key size will not have much effect on its performance. We estimate that *HeW* can be efficiently implemented in hardware with a maximum of 1000 slices and 2112 look up tables. This seems to be a good number in terms of hardware performance.

5.3 NIST Randomness Tests

Hash digest for any arbitrary length message must satisfy the randomness properties[21]. We test the random nature of hash digest using NIST Statistical Test Suite SP800-22[22]. We need 100 different files and each file should contain approximately 10 lakh bits for testing the randomness. We process each message and get a 256-bit hash output for the intended message. To generate the required 10 lakh bits, we keep on applying the hash function *HeW* until we get the 10 lakh bits in the output file. We have the following results (Table 3) on 100 files using the NIST suite for the 5 basic randomness tests.

5.4 Near-collision Resistance

If two different input messages generate the almost same hash value, then this can lead to a collision attack[23]. If it is computationally hard to find two different messages whose hash output differ in the small number of bits then hash

Table 3. NIST test results

Statistical test	P-Value	Proportion
Frequency	0.026948	100/100
Block frequency	0.2022686	100/100
Runs	0.637119	99/100
Overlapping template	0.085587	100/100
Serial	0.102526	99/100

function is called near-collision resistant. We checked the near-collision resistance of *HeW* by generating the large number of input files. We have generated 100,000 random input message files and calculated their hash value using *HeW*. We selected two random files from the hash digest lot and calculate their hamming distance. We can choose two files out of 100,000 files in $\binom{100000}{2}$ different ways which gives 4,999,950,000 different file combinations. We analysed the results for all combination of files. We can get the hamming distance values in the range of $0,1,2,\ \ldots,256$. We got the minimum and the maximum value of hamming distance as 78 and 181 bit differences, respectively. The maximum value for the hamming distance occurred 249,073,042 times which is recorded for the 128 bit difference.

We get the difference between 108 and 148 for the following number of files

$$(108 \leq \backslash \# \mathit{files} \leq 148) = 4{,}948{,}691{,}207 (\mathit{i.e.}, 98.97\%)$$

We got approximately 99 per cent of the files having the hamming distance range between 128 ± 20 which indicates that these won't lead to any near-collision attack. The hamming distance between two files needs to be really small viz. up to 16-bit to generate a near collision. Hence, we can say that *HeW* is resistant to near-collision attack.

5.5 Avalanche Effect

Avalanche criterion states that if we change 1-bit in the input then there must be an approximate 50 per cent change in the output bits[23]. We tested the Avalanche effect on the output of *HeW*. We started with a 1024-bit message M_0 which is shown in Appendix B.

For $1 \leq i \leq 1024$, we generated 1024 messages (M_i) with 1-bit difference from M_0 as follows:

$$M_i = M_0 \oplus (1 << i)$$

We applied *HeW* on these 1025 messages and calculated 256-bit hash for each message. For $1 \leq i \leq 1024$, we found the hamming distance between Hash (M_0) and Hash (M_i) as shown in Table 4. We also computed the hamming distances word-wise. We divided the 256-bit hash output into the eight 32-bit words ($W_1, W_2, \ldots, W_8$). Results for the minimum (Min), maximum (Max), mode and average value of distances is shown in Table 4. We plotted the hamming distance range of 1024 files for 256-bit hash digest in Fig. 5 which shows that they are almost uniformly distributed i.e., change in one bit of the input carries 50 per cent change in the hash digest.

5.6 Bit Variance Test

Bit variance test is one of the statistical tests for testing the random nature of the binary data. This test measures the

Table 4. Hamming distances

Changes	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	*HeW*
Min	7	7	8	8	7	7	8	8	96
Max	24	24	24	26	24	24	25	25	153
Mode	17	17	16	17	17	15	16	16	126
Mean	16.08	15.94	15.95	16.05	16.14	15.89	16.07	16.01	128.17

Table 5. Range of hamming distances

Range of hamming distances	Number of files within range	Change in output bits of *HeW* digest (per cent)
128 ± 5	538	52.53
128 ± 10	806	78.71
128 ± 15	969	94.62
128 ± 20	1011	98.73

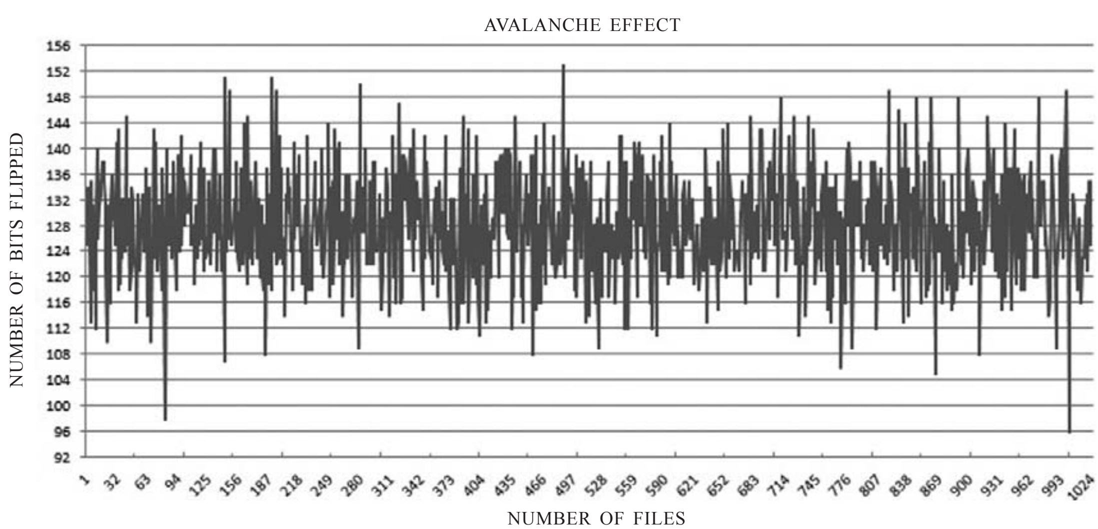

Figure 5. Hamming distances range of the 1024 files.

impact for change in the input message bits on the digest bits. A variable length input is transformed to a 256-bit hash digest using *HeW*. If there is a change in one or some of the input bits, then impact of this change on each of the output bit should be uniform[23]. We took the same set of 1025 messages which we have used to measure the avalanche effect. We got the 256-bit hash output for each of the 1025 messages. For each bit position in the hash digest, we calculate the probability of this bit being 1. If the probability, $P_i(1) = P_i(0) = 1/2$ for all digest bits $i = 1, ..., 256$ then we assured that *HeW* passes the bit variance test. Since it is computationally infeasible to consider all input message bit changes, we have considered the results only for 1025 files, viz. $M_0, M_1, M_2, \ldots, M_{1024}$. We found the following results for mean frequency of 1s:

Digest length = 256
Number of digests = 1025
Mean frequency of 1s (expected) = 512.50
Mean frequency of 1s (calculated) = 512.44

We plotted the probability for each of the bits (256-bit) in Fig. 6 and observed that average probability of 1's is approximately 0.50. This indicates that *HeW* passes the bit variance test.

5.7 Differential Cryptanalysis

Differential attack is the basic cryptanalysis technique used on block ciphers. It was the first successful attack applied on DES by Biham and Shamir[24], which reduced the key search complexity of DES than the exhaustive search. We used the probabilistic relationship between the input and output differences of a cipher to mount this attack. We analysed the components of a cipher to construct a high probability trail by joining several one round relations. We used lightweight block cipher *FeW* to design the hash function *HeW*. Security proof of *FeW* is provided by Kumar[14], *et.al.* which shows that *FeW* is secure against differential cryptanalysis. It is proved that differential attack on *FeW* cannot be applied beyond 14 rounds. We have theorem 1 for the bound on the number of active S-boxes in any three rounds of *FeW*.

Theorem 1. Any three rounds of *FeW* have a minimum of five active S-boxes[14].

We used the technique of counting the minimum number of active S-boxes in a differential trail[25,26]. *HeW* uses single $4 \times 4 S$ -box inside the compression function. The maximum differential probability in one S-box application[14] is 2^{-2}. There are 8 parallel applications of FeW_{1R} on the 512-bit register inside the compression function. After each round, 512-bit register is rotated left by 16 bits. We called the FeW_{1R} block as active 64-bit word, if there is some non-zero nibble as input to FeW_{1R} block. We start with a non-zero difference in a 4-bit nibble within one 64-bit message block. After applying key expansion algorithm, it is guaranteed that the non-zero difference in any 4-bit nibble will be used as a round subkey after 2 rounds. We do not count the S-boxes which are activated during the key expansion. We considered the effect of one 4-bit non-zero nibble only. We counted the number of active FeW_{1R} blocks which are shown in the Fig. 7 and Table 6. We also have the following theorem for FeW_{1R} blocks.

Theorem 2. After every 2 rounds in the compression function of HeW, one new 64-bit block gets activated for input to the FeW_{1R}.

All FeW_{1R} blocks (i.e. 8) gets activated after 17 rounds. Using theorem 1 and 2, we find the minimum number of active S-boxes in the full round differential trail of *HeW* as follows:

(i) There are 60 active S-boxes in the first 16 rounds of compression function.
(ii) Due to one active FeW_{1R} block, there are 25 active S-boxes in the last 16 rounds.
(iii) We get 200 active S-boxes in the last 16 rounds due to the 8 active FeW_{1R} blocks from round 17 to 32.

Thus, any 32-round differential trail will consist of 260 active S-boxes, which guarantees that we can get $\left(2^{-2}\right)^{260}$ i.e, 2^{-520} as the maximum differential probability for any 32-round trail of *HeW*. As a result, we require 2^{520} chosen plain-text pairs to distinguish the most significant 64-bit of the hash digest. This bound ensures that differential attack cannot be applied to the hash function *HeW*.

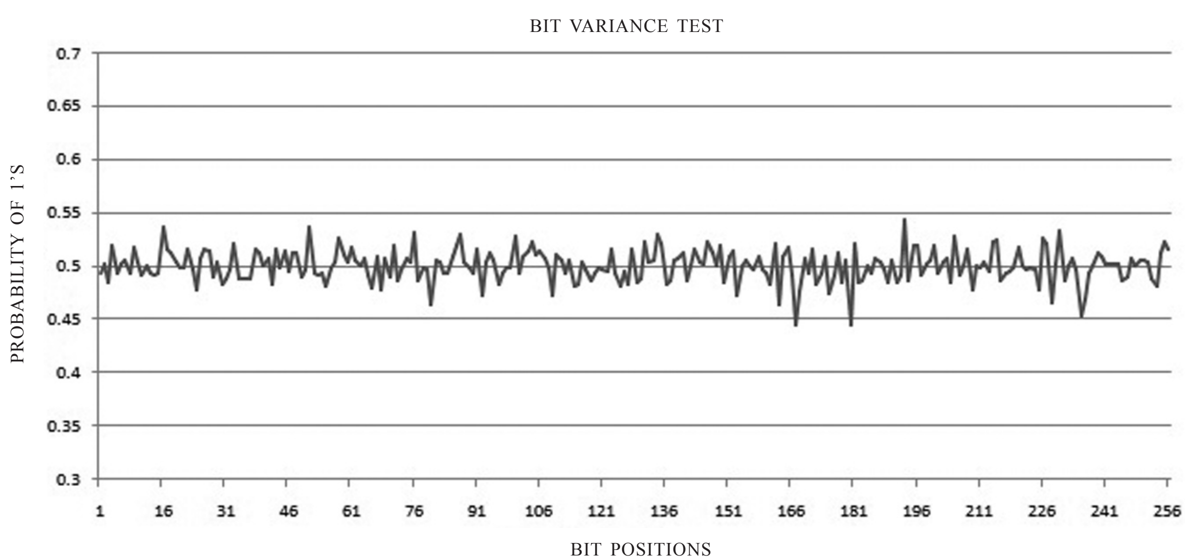

Figure 6. The probability of the bit position.

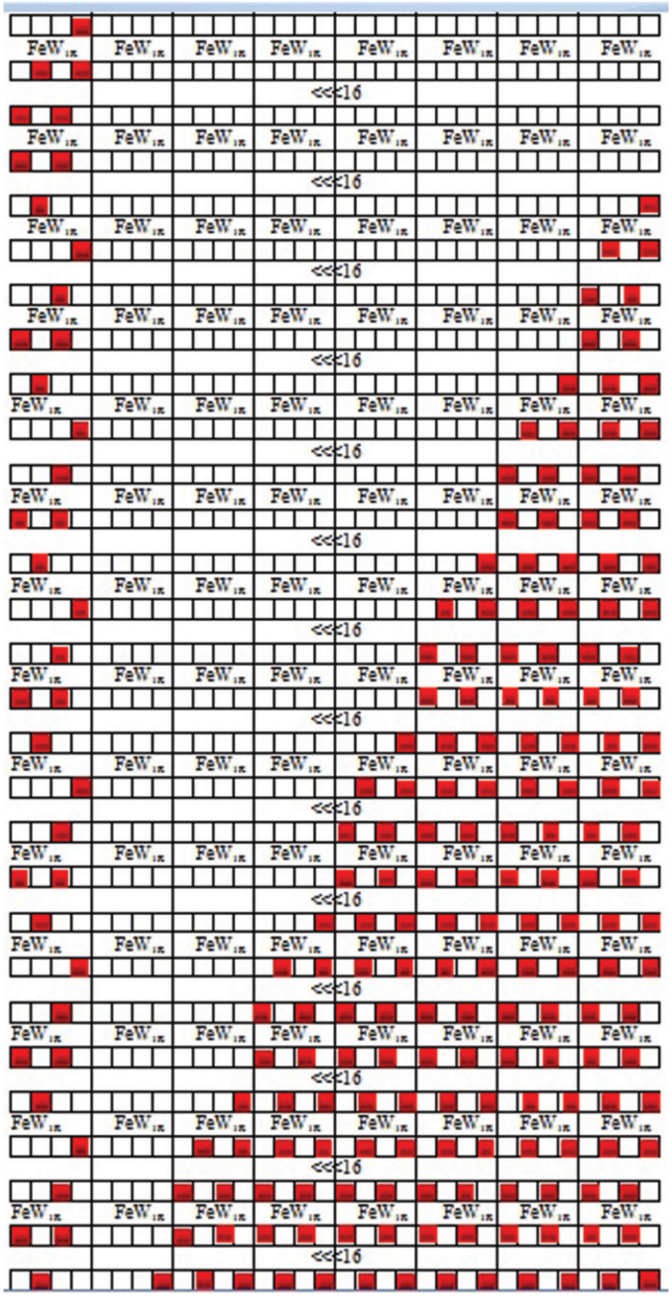

Figure 7. Differential trail for round 3 to 16.

5.8 Length Extension Attack

If we used hash function as a message authentication code (MAC), then length extension attack can lead to forgery attack against MAC's. This attack was devised for MD5 hash algorithm which process the n-bit message and n-bit $\mathcal{IV}$ in one iteration and finally generates n-bit hash digest[18]. For a message M, we get padded message as $m = M \parallel Pad(M)$. If we use MD5 hash function and know the length of the message, then we can use $H(m)$ as $\mathcal{IV}$ and append the message M' as $m' = H(m) \parallel M'$. We now calculated the hash value of the extended message, which will be a valid MAC for the message m'. To prevent this attack, we can use wide-pipe mode of hash construction which takes $2n$-bit $\mathcal{IV}$ and $2n$-bit message as input and n-bit hash digest is generated. *HeW* takes two inputs (512-bit $\mathcal{IV}$ and 512-bit message block) and outputs 256-bit hash digest. In case of hash function *HeW*, length of the hash output is half of the length of $\mathcal{IV}$, therefore we conclude that length extension attack cannot be applied on *HeW*.

Table 6. Minimum number of active FeW_{1R} blocks in 32-round trail

Round	No of active FeW_{1R} blocks	Round	No of active FeW_{1R} blocks
1	0	17	8
2	0	18	8
3	1	19	8
4	1	20	8
5	2	21	8
6	2	22	8
7	3	23	8
8	3	24	8
9	4	25	8
10	4	26	8
11	5	27	8
12	5	28	8
13	6	29	8
14	6	30	8
15	7	31	8
16	7	32	8

5.9 Slide Attack

Slide attack was proposed for block ciphers and it is used to recover the key in a block cipher[27]. It exploits the weakness in the key schedule of a block cipher and construct a slid pair using the similarity in the round keys. We have used the block cipher *FeW* to design the hash function *HeW*, so we need to consider the security from slide attack. There are two types of possible slide attacks. The first kind of slide attack applies sliding on round transformation, while the second kind of attack applies sliding on message block. There are certain preventive measures used in *FeW* to counter this attack. The first layer of security is the use of round constant in the key expansion algorithm. Secondly, we imbibe a 16-bit left rotation in *HeW* which is another measure to prevent the slide attack. We, therefore conclude that slide attack cannot be applied to *HeW*.

5.10 Rotational Distinguisher

Rotational distinguisher was proposed to analyse the ARX based structures[28]. This attack has been less effective on the designs using S-box and MDS type layers in their round function[27]. There is an application of 4x4 S-box in the round function of *HeW*. This attack can work for *HeW*, if the rotation amount is a multiple of the size of the S-box (i.e. 4). The rotational value other than 4 will be destroyed by the application of 4x4 S-box. If we take the rotational value as 4, then rotational pair will be further destroyed by the application of nibble permutation layer on 16-bit branches inside round function and 16 bits rotation after every round. We, therefore conclude that rotational distinguisher cannot be effectively applied to our hash function *HeW*.

6. CONCLUSION

A new hash function *HeW* which is based on a lightweight block cipher is proposed in this paper. The compression function of *HeW* is built using a software oriented lightweight block cipher *FeW* which can also be implemented in hardware efficiently. The collision resistance bound for *HeW* is 2^{128}, which is better than present security recommendations of 2^{112}. We have presented the analysis of *HeW* for differential attack, length extension attack, slide attack and rotational distinguisher. We applied NIST test suite on the data generated using *HeW* and it passes the randomness tests. It also passed other tests including avalanche effect, bit variance test and near-collision resistance. Software efficiency of our design is better than SHA-256. The compression function of MD4 and SHA-1 family are based on Merkle-Damgård construction which is prone to the length extension attack. Therefore, our proposed scheme can work as a better alternative to the MD4 and SHA-1 family in terms of security and efficiency.

REFERENCES

1. Government of India, digital India - Power to empower. http://digitalindia.gov.in (Accessed on Accessed on 13 September 2016)
2. Naor, M. & Yung, M. Universal one-way hash functions and their cryptographic applications. *In* Proceedings 2st ACM Symposium on the Theory of Computing, 1990, pp. 33-43.
   doi: 10.1145/73007.73011
3. Bartkewitz, T. Building hash functions from block ciphers. their security and implementation properties, Ruhr-University Bochum, 2009, pp. 1-22.
4. Preneel, B. Analysis and design of cryptographic hash functions. Doctoral Dissertation, Katholieke Universiteit Leuven, 1993, pp. 1-338.
5. Lai, X. & Massey, J. Hash functions based on block ciphers. *In* Eurocrypt, LNCS, Springer, 1993, **658,** pp. 55-70.
   doi: 10.1007/3-540-47555-9_5.
6. Damgard, I. A design principle for hash functions. *In* Crypto, LNCS, Springer, 1989, **435**, 416-427.
   doi: 10.1007/0-387-34805-0_39
7. Gauravaram, P. & Knudsen, L.R. Cryptographic hash functions. *In* Handbook of Information and Communication Security. *Springer*, 2010, pp. 59-80.
8. Rivest, R.L. The MD4 message digest algorithm. *In* Advances in Cryptology, Proceedings Crypto 1990. LNCS, Springer-Verlag, 1991, **537**, 303-311.
   doi: 10.1007/3-540-38424-3_22
9. Coron, J.S.; Dodis, Y.; Malinaud, C. & Puniya, P. Merkle-damgard revisited: How to construct a hash function, advances in cryptology. *In* Proceedings Crypto, LNCS, Springer-Verlag, 2005, **3621**, pp. 430-448.
   doi: 10.1007/11535218_26
10. Regenscheid, A.; Perlner, R.; Chang, S.; Kelsey, J.; Nandi, M. & Paul, S. Status report on the first round of the SHA- 3 cryptographic hash algorithm competition. NIST Internal Reports 7620, 2009. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents (Accessed on 27 December 2015).
11. Bertoni, G.; Daemen, J.; Peeters M. & Assche, G. V. The Keccak SHA-3 submission. 2011. http://keccak.noekeon.org/ (Accessed on 2 August 2015).
12. ISO/IEC 10118, Information Technology, Security Techniques, Hash-functions, Pt 1: General, 2000; Pt 2: Hash-functions using an n-bit Block Cipher Algorithm, 2000; Pt 3: Dedicated Hash-functions, 2003; Pt 4: Hash-functions using Modular Arithmetic, 1998.
13. Stam, M. Block cipher based hashing revisited, fast software encryption. LNCS, Springer-Verlag, 2009, **5665**, 67-83.
    doi: 10.1007/978-3-642-03317-9_5
14. Kumar, M.; Pal, S.K. & Panigrahi, A. *FeW*: A lightweight block cipher. Cryptology ePrint Archive, Report 2014/326, 2014. http://eprint.iacr.org/2014/326.pdf (Accessed on 25 June 2015).
15. Nemati, A.; Feizi, S.; Ahmadi, A.; Haghiri, S.; Ahmadi, M. & Alirezae, S. An efficient hardware implementation of FeW lightweight block cipher. *In* IEEE Symposium on Artificial Intelligence and Signal Processing, 2015, pp. 273-278.
    doi: 10.1109/AISP.2015.7123493.
16. Preneel, B.; Govaerts, R. &Vandewalle, J. Hash functions based on block ciphers: A synthetic approach. Advances in Cryptology, *In* Proceedings Crypto, LNCS, Springer-Verlag, 1994, **773**, 368-378.
17. Rogaway, P. & Steinberger, J.P. Constructing cryptographic hash functions from fixed-key block ciphers. Advances in Cryptology, *In* Proceedings Crypto, LNCS, Springer-Verlag, 2008, **5157**, pp. 433-450.
    doi: 10.1007/978-3-540-85174-5_24
18. Lucks, S. A failure friendly design principle for hash functions. Asiacrypt, LNCS, Springer, 2005, **3788**, 474-494.
    doi: 10.1007/11593447_26
19. Hirose, S. Some plausible constructions of double-block-length hash functions. Fast Software Encryption, LNCS, Springer-Verlag, 2006, **4047**, 210-225.
    doi: 10.1007/11799313_14.
20. Dunkelman, O. & Biham, E. A framework for iterative hash functions. HAIFA, 2nd NIST Cryptographic Hash Workshop, 2006. https://eprint.iacr.org/2007/278.pdf (Accessed on 18 January 2016).
21. Karras, D. & Zorkadis, V. A novel suite of tests for evaluating one-way hash functions for electronic commerce applications. *IEEE*, 2000, pp. 464-468.
22. Gallagher, P. A statistical test for random and pseudorandom number generators for cryptographic application, April, 2010. https://www.nist.gov/. (Accessed on 28 April 2016).
23. Bussi, K.; Dey, D.; Kumar, M. & Dass, B.K. Kupy-neev hash function. *Italian J. Pure Appl. Math.,* 2016, **36,** 929-944.
24. Biham, E. & Shamir, A. Differential cryptanalysis of

the full 16-round DES. *In* Proceedings Crypto, LNCS, Springer, 1993, **740**, pp. 487-496.
doi: 10.1007/3-540-48071-4_34.
25. Wang, M. Differential cryptanalysis of reduced-round present. Africacrypt, LNCS, springer, 2008, **5023**, 40-49.
doi: 10.1007/978-3-540-68164-9_4.
26. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M. J. B.; Seurin, Y. & Vikkelsoe, C. Present: An ultra-lightweight block cipher. CHES, LNCS, Springer, 2007, **4727**, 450-466.
doi: 10.1007/978-3-540-74735-2_31.
27. Wu, W.;Wu, S.; Zhang, L.; Zou, J. & Dong, L. LHash: A lightweight hash function. Inscrypt, LNCS, 2014, **8567**, 291-308.
doi: 10.1007/978-3-319-12087-4_19.
28. Khovratovich, D. & Nikolic, I. Rotational cryptanalysis of ARX. Fast software encryption. *LNCS,* 2010, **6147**, 333-346.
doi: 10.1007/978-3-642-13858-4_19

ACKNOWLEDGEMENTS

We would like to thank Director, SAG and Ms. Pratibha Yadav for their valuable guidance and continuous support to work in this direction. We would also like to thank anonymous reviewers who have invested their precious time to suggest the improvements in the paper.

CONTRIBUTORS

Mr Manoj Kumar received M.Phil form CCS University, Meerut, in 2004 and pursuing PhD from Department of Mathematics, University of Delhi. He is currently working as a Scientist in Scientific Analysis Group, DRDO. His research area includes design and analysis of block ciphers.
In current study, his contributions are in design and analysis of the hash function and implementation of various tests.

Dr Dhananjoy Dey received his PhD form Jadhavpur University. He is currently working as a Scientist in Scientific Analysis Group, DRDO. His areas of interest are design and analysis of hash functions.
In current study, his contributions are in overall design sketch and various tests like bit variance test, length extension attack and near collision resistance.

Dr Saibal K. Pal received his PhD form University of Delhi. He is currently working as a Scientist 'G' in Scientific Analysis Group, DRDO. His interest areas include multimedia and network security, computational intelligence and data mining.
In current study, he has guided at various design stages and provided his inputs to improve the design.

Dr Anupama Panigrahi received her PhD from Allahabad University (HRI). She has been post-doctoral fellow in ISI, Kolkata. She is currently working as an Assistant Professor in Department of Mathematics, University of Delhi. Her main research areas include number theory and elliptic curve cryptography.
In current study, she contributed to provide the overall guidance and critical suggestions in analysis of the scheme.

Appendix A

Test Vectors

We generate the test values of hash digest for three different inputs: a, ab and abc . The hash output for each input is given below:

Hash(a)	=	3d3292c7dcf9d9f0990bdb41afe37d10	69d5bb87e9474945d0560a0ae539dd10
Hash(ab)	=	90c4984c4ccc7dfa44d21c2537b0ba3f	d6b744bb90c28a8eaa44f5f039cad560
Hash(abc)	=	0e7f4db99d30a4ebac17845ba756c504	c753ae8a23516b24e9fe349b2e238b3d

Appendix B

Message M_0

We take the following 1024-bit Message M_0 (in hex) for Avalanche and Bit variance tests:v

1234567890abcdef 1234567890abcdef1234567890abcdef1234567890abcdef
1234567890abcdef 1234567890abcdef1234567890abcdef1234567890abcdef
1234567890abcdef 1234567890abcdef1234567890abcdef1234567890abcdef
1234567890abcdef 1234567890abcdef1234567890abcdef1234567890abcdef