MGR Hash Function

Khushboo Bussi[1], Dhananjoy Dey[2], P.R. Mishra[2], B.K. Dass[1]

[1]Department of Mathematics, University of Delhi,
Delhi-110 007, INDIA.
khushboobussi7@gmail.com, dassbk@rediffmail.com
[2]Scientific Analysis Group, DRDO, Metcalfe House Complex,
Delhi-110 054, INDIA.
{dhananjoydey, pr_mishra}@sag.drdo.in

August 4, 2015

Abstract

GOST-R is a Russian Standard Cryptographic Hash function which was first introduced in 1994 by Russian Federal for information processing, information security and digital signature. In 2012, it was updated to GOST-R 34.11-2012 and replaced older one for all its applications from January 2013. GOST-R is based on modified Merkle-Damgård construction. Here, we present a modified version of GOST-R (MGR-hash). The modified design is based on wide pipe construction which is also a modified Merkle-Damgård construction. MGR is much more secure as well as three times faster than GOST-R. AES like block cipher has been used in designing the compression function of MGR because AES is one of the most efficient and secure block cipher and it has been evaluated for more than 12 years. We will also analyze the MGR hash function with respect to its security and efficiency.

Keywords: AES, collision resistance, dedicated hash functions, differential attack, Merkle-Damgård construction, wide-pipe hash .

1 Introduction

Hash is the term basically originated from computer science where it means chopping up the arbitrary length message into fixed length output. In the beginning hash functions were used in computer science for table look-ups, searching and sorting only [1]. With the invention of public key and digital signatures [2], cryptographic hash functions have a significant role in cryptography. A hash function checks the integrity as well provides the authentication (in the case of keyed hash functions i.e. MAC) of the message that has been transmitted. Cryptographic hash functions become indispensable tool for public key cryptography (PKC) [3]. The hash functions essentially many-to-one functions, which map arbitrary length

messages to fixed length digest that are mostly shorter than the length of the message. We can define it in the following manner.

Definition 1. *A hash function h, is a function $h : \mathcal{D} \to \mathcal{R}$, where $\mathcal{D} = \{0,1\}^*$ and $\mathcal{R} = \{0,1\}^n$ for some $n \geq 1$.*

Hash functions are further divided into *cryptographic* and *non-cryptographic*. In this paper we will discuss only cryptographic hash functions.

A cryptographic hash function needs to satisfy certain properties which are mentioned as follows:

(i) **Preimage resistance:** it is infeasible to find the message whose hash value or digest is given, i.e., for any given $h(m)$ it is computationally infeasible to find m.

(ii) **Second preimage resistance:** it is infeasible to modify a message without changing its hash value, i.e., for any given m_1, it is computationally infeasible to find m_2 ($\neq m_1$) such that $h(m_1) = h(m_2)$.

(iii) **Collision resistance:** it is infeasible to find two different messages having same hash value, i.e., it is computationally infeasible to find m_1 and m_2 such that for $m_1 \neq m_2$, $h(m_1) = h(m_2)$.

There are cryptographic hash functions which satisfy many more properties than the mentioned above but those three are mandatory conditions. According to Preneel, hash functions satisfying properties (i) and (ii)above are called *one way hash functions* and those which satisfy all the three properties mentioned above are called *collision resistant hash functions* [4].

We have so many dedicated cryptographic hash functions like MD5, SHA-1, SHA-2 family [5] and recently *Keccak* [6] has been added to this class as 'SHA-3' [7] [8]. Their ubiquity can be seen in today's world as these are significantly used in a day to day life from an email account to online shopping, from digital signature to internet banking. So, it is necessary that the hash functions which are needed to provide a great deal of e-security, should be efficient enough to meet up today's needs. Their very much usage in real world, made cryptographers to think about the upcoming designs which can give a good trade-off between security and efficiency.

In this paper we describe a modified version of GOST-R hash function and we will call this as MGR hash function from now onwards where we replace the underlying block cipher of compression function of GOST-R by AES like block cipher and also change mode operation. This paper is organized in the following manner. In Section 2, we give a brief description of GOST-R hash function. Our newly designed MGR hash function is described in details in Section 3. In Section 4, we present the security analysis of MGR hash function using various tests and its efficiency along with the comparison with GOST-R algorithm.

2 GOST-R

GOST-R [9] 34.11-94 is a Russian Federal standard hash algorithm originated in 1994. It is a block cipher based hash function. GOST-R 34.11-94 is superseded by GOST-R 34.11-2012 in 2013 and become standard hash algorithm of Russia. It takes the message of arbitrary length and gives the output in 256-bit or 512-bit. It is a standard hash function which is used to maintain the integrity, authentication and non-repudiation of the information during its transmission, processing and storage in computer systems.

In GOST-R 34.11-2012, the hash sizes are of 256-bit and 512-bit with initial values are equal to $(00000001)^{64}$ and 0^{512} respectively[1]. The compression function consists of *non-linear bijections*, *byte permutations* and *linear transformations*. The substitution of bytes are non-linear bijections which brings non-linearity in the output and then the bytes are permuted using byte permutations. 64-bit length strings are right multiplied to a 64×16 matrix A over the field $GF(2)$ to achieve the linear transformation. So, the compression function consists of LPS (Linear transformation, Permutation and Substitution). There are thirteen 512-bit constants which will be used iteratively. In each iteration, the round function gives an output of 512-bit. The underlying block cipher in each round function acts on a constant and 512-bit message block. Each round has a thirteen steps corresponding to thirteen constants and LPS is applied at every step of the block cipher.

All stages of GOST-R can be understood easily by the following diagram given by *Ma et al.* [10]

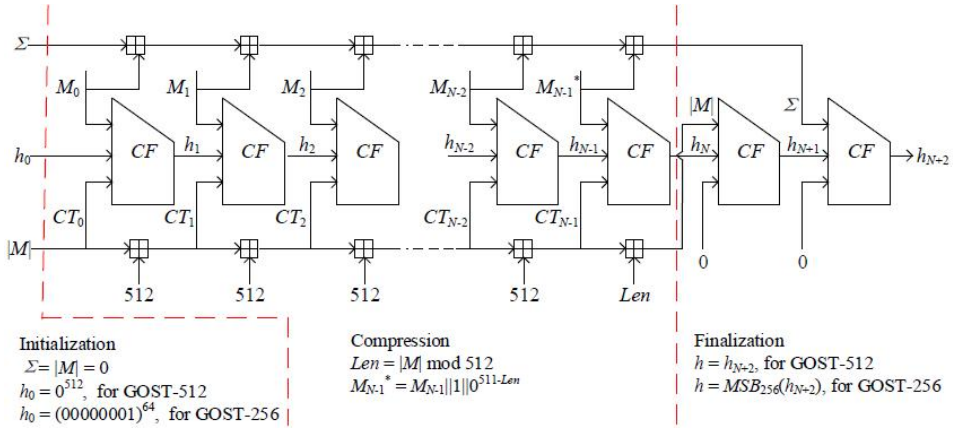

Figure 1: Three Stages of GOST-R hash function

The stage 1 is initialization stage where $\sum$, N (counter) and h are assigned the constant values 0, 0 and $\mathcal{IV}$ respectively. At the next stage, the input message is padded and t blocks of length 512-bit are taken. For each message block M_i, the iteration compression function $CF(h, M_i)$ is applied. Finally at stage 3, CF is applied on the total length of message $|M|$ to get the final hash digest.

This design is based on Merkle-Damgård construction [11]. So it does have same weaknesses like length extension attack, MAC forgery attack and many more.

[1] where 0^{512} represents 512-bit zeros and $(00000001)^{64}$ represents $0x\overbrace{0101\cdots01}^{64-times}$

GOST-R gives a good trade-off between efficiency and memory storage but security part suffers due to Merkle-Damgård based Construction. In MGR we have changed the compression function and it is based on wide-pipe construction or precisely double-pipe construction [12] to get rid of weaknesses and hence trying to make it less vulnerable to attack and getting a better trade-off between efficiency and security. In the following section we explain our new design MGR in detail.

3 MGR Hash Function

We have mentioned MGR hash function in the earlier section. In this design, the underlying block cipher of GOST-R is replaced by AES [13] [14]like block cipher. It can also give an output of variable length say, i.e. any length in between 256-bit to 512-bit. We can attain any length of the digest between 256 to 512-bit, in the multiple of 32-bit. For different hash digest length, we just need to fix different but constant initial values. Here, we will have a hash digest of 256-bit. The message is firstly padded so it becomes a multiple of 1024-bit and an initial register of 1024-bit consisting of an initial value h_0 of 512-bit along with a constant(counter) of 512-bit are taken as input of eight AES-128 in parallel. As, it is a double-pipe hash, hence the chaining variable is of half of the block length i.e. 512-bit. Eight simultaneous AES works on the input of h_0 and message block. After the first round of AES the output 1024-bit is left rotated by 96-bit to ensure the diffusion with every round of AES. Every round of the block cipher is indeed a round of AES; there is a need of 1024-bit key (schedule) for every round and it is done by using round (pre-defined) constants as key and updated message as message block. The design is described in detail as follows:

3.1 Padding

The message can be of arbitrary length and need not be multiple of block length b always. So, there should be a padding rule for the all the input messages. Given an input message M of length ℓ bits, append the bit 1 to end of M, and then append $(-\ell - \mu - 1) \equiv k \; mod \; b$ '0' bits, where μ is a 128-bit representation of ℓ. Append k '0' bits to $M||1$ and then append μ to this. In this way the message block will always be the multiple of the block length b.

Padding ensures that the message would not give collision easily as the length of the message is included twice. [15]

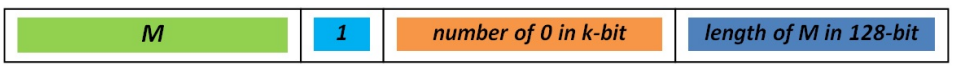

Figure 2: Padding Procedure

3.2 Parsing

MGR is a block cipher based hash design, the padded message needs to be divided into the blocks of 1024 bits, i.e. in this case $b = 1024$. Let L be the length of

padded message. Divide padded message into t ($= L/1024$) 1024-bit blocks. So, we have

$$M||Pad(M) = M_1||M_2||\cdots||M_t,$$

where each M_i is a 1024-bit block.

3.3 Initial Values

An initial register of compression function is of 1024-bit representation and its input coming from two parts:

Initial value $\mathcal{IV}$ of MGR is 512-bit representation of zero.

$$\mathcal{IV} = \overbrace{000\cdots000}^{512-times}$$

The 512-bit counter C_0 is

C_0 = `2beec95eb257ff260e2337f9c80839f128f679ef52105f4484a48cef9c77702d`
`64893fb6a9981b8898b463797aefd08d8a16c2a4e9abb2ae23d5d95c16de0d64`

3.4 Key Schedule

Compression function of MGR consists of 10 rounds. Each round of it consists of 8 parallel single round AES. So, there is a need of key generation for every round. The nine constants of 1024-bit are used as key for each round of our compression function and they are generated with the help of SHA-512. Message 'MGR Hash Function' is hashed by SHA-512 and once we get 512-bit digest, it is again hashed using it as input to SHA-512 and both the 512-bit outputs (earlier and later) are concatenated to get a 1024-bit constant. This process continues till we get all the nine constants and the nonce which will be used as counter C_0 (512-bit). These constants are given in Table 1.

Thus, we can write

$$u_0 = \text{SHA-512(MGR Hash Function)}$$

For $i = 1$ to 17

$u_i = SHA\ 512(u_{i-1})$

The nine round constants are generated by the following way:

For $j = 1$ to 9

$RC_j = u_{2j-2}||u_{2j-1}$

Values of round constants have been given in Appendix-1 and

$C_0 = u_{18} = SHA\ 512(u_{17})$

Message is encrypted by each round of AES, using these constants as keys and also use key schedule for other rounds of AES.

For $j = 1$ to 9

$$k_j = AES_1(RC_j, k_{j-1})$$

where AES_1 represents first round of AES and k_0 ($=$ M_1) is the first message block taken.

3.5 Hash Construction

An initial value of 512-bit and counter of 512-bit, which we already discussed, along with message block of 1024-bit is taken as a input for the compression function f and is processed as

$$h_i \leftarrow f(M_i, h_{i-1}||C_{i-1}) \tag{1}$$

where $C_{i-1} = C_0 \boxplus_{512} (i-1)$

Hence f has three inputs namely the chaining variable (h_{i-1}), counter (C_{i-1}) each of 512-bit and the message block (M_i) of 1024-bit and gives an output (h_i) of 512-bit.

3.6 The Compression function

The compression function of MGR has basically AES as a underlying block cipher. It consists of 10 rounds and each round consists of 8 parallel single round of AES. In this design, there is an $\mathcal{IV}$ of 512-bit concatenate with a 512-bit constant (counter) and a message block of 1024-bit and 8 parallel AES-128 acting on message as a key and initial register H_0 as a message. AES-128 has been used as a underlying block cipher in the compression function, taking first 128-bit of message as a key and first 128-bit of H_0 as message and simultaneously next 128-bit of message has been taken by the next parallel AES and so on. Thus, there are 8 parallel AES acting on 1024-bit message and 1024-bit initial register to have an output of 1024-bit. In the end of one round of AES, a left rotation of 96-bit has been given to the overall 1024-bit output block. The left rotation ensures the avalanche effect as a change in one bit of the message block will bring change not just in the respective 128-bit block but also make sure that the change will diffuse to other blocks of output. So, change in one bit will bring change $\approx 50\%$ of the output. For the next round onwards of AES, there is a need of key scheduling which we have already discussed in the earlier section. After 10^{th} round we have an updated register of 1024-bit and now the most significant (MSB) 512-bit of the updated register can be taken as h_1 (chaining variable) for the next message block.

So, for one message block, we would have twenty rounds of AES which will be a good trade-off between the security and the efficiency of the design.

3.7 Mode of Operation

The compression function will work iteratively on all the message blocks till they get exhausted. The 512-bit h_1 is the output of one round of compression function applying on M_1 (first message block), it has been taken as a chaining variable alongwith the counter C_1 to the next message block M_2 and hence it goes on like this till all the message blocks have been taken as an input. When the last message

block M_t is taken as input, the 256-bit MSB of the final output h_t, is the hash digest h'_t.

Thus, we can write

$$\begin{aligned} M||Pad(M) &= M_1||M_2||\cdots||M_t \\ h_1 &= f(M_1, \mathcal{IV}||C_0) \\ h_2 &= f(M_2, h_1||C_1) \\ \vdots &\ \vdots\ \ \vdots \\ h_t &= f(M_t, h_{t-1}||C_{t-1}) \\ h'_t &= MSB_{256}h_t \end{aligned}$$

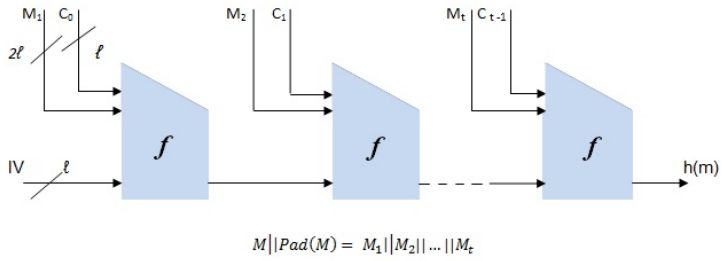

Figure 3: Compression function

3.8 MGR Computation

MGR of message M, first message block $M_1 = k_0$ is calculated using following algorithm.

$$M||Pad(M) = M_1||M_2||\cdots||M_t$$

(i) **Notation**: 512-bit initial value

$$h_0 = \overbrace{000\cdots000}^{512-times}$$

C_0 = `2beec95eb257ff260e2337f9c80839f128f679ef52105f4484a48cef9c77702d`
`64893fb6a9981b8898b463797aefd08d8a16c2a4e9abb2ae23d5d95c16de0d64`

(ii) **Initial Register**: 1024-bit $H_0 = h_0||C_0$

(iii) **Constants**: Nine 1024-bit round constants

For i=1 to 9, RC_i

(iv) **Processing**:

$M_1^{(1)} = AES_1(M_1, H_0)$ [First round of eight simultaneous AES on 1024 bit Message (as key) and 1024 bit initial register (as message)]

$M_1^{(1)'} \leftarrow rotl_{96}M_1^{(1)}$

(v) **Key Schedule**: For $M_1^{(1)}$ (as a output of 1st round of AES, we need to have a key expansion to proceed for other rounds of AES)

for $i = 1$ *to* 9 **do**

$M_1^{(i+1)} = AES_1(k_i, M_1^i \oplus RC_i)$

$M_1^{(i+1)'} \leftarrow rotl_{96} M_1^{(i+1)}$

$k_i = AES_1(RC_i, k_{i-1})$

end

$h_1 \leftarrow MSB_{512} M_1^{10'}$

$h_1' = MSB_{256} h_1$

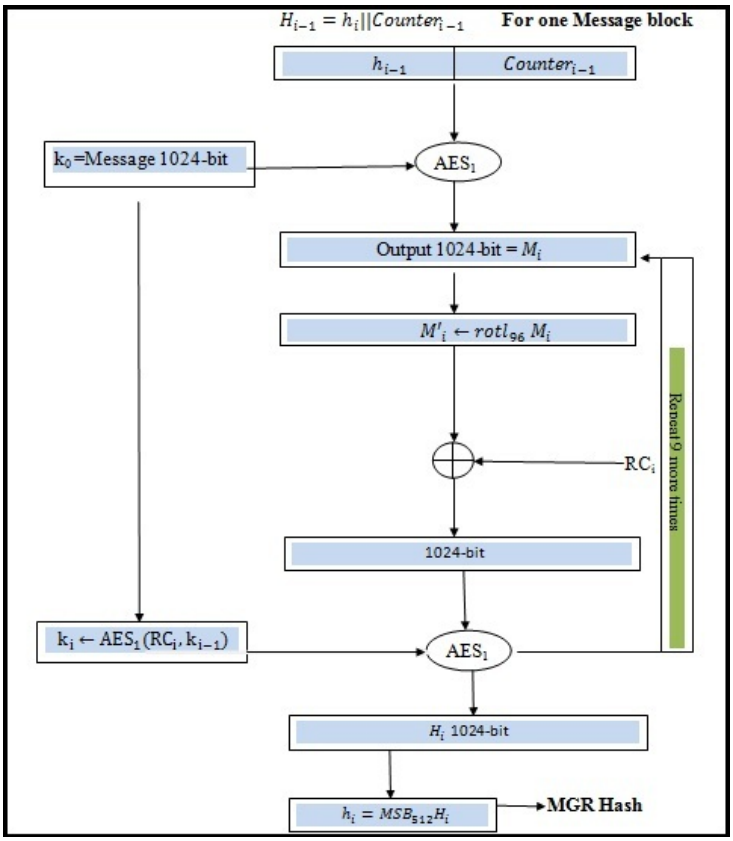

Figure 4: MGR Hash algorithm

3.9 Test Value of MGR Hash

Test values of the three inputs are given below:

input : $M_1, M_2, \ldots, M_t$
for $j = 1$ *to* t **do**
  $H_0 = \mathcal{IV}||C_0$
  $M_j^{(1)} = AES_1(M_j, H_0)$
  $M_j^{(1)'} \leftarrow rotl_{96} M_j^{(1)}$
  for $i = 1$ *to* 9 **do**
    $k_i = AES_1(RC_{i-1}, k_{i-1})$
    $M_j^{(i+1)} = AES_1(k_i, M_j^{(i)} \oplus RC_i)$
    $M_j^{(i+1)'} \leftarrow rotl_{96} M_j^{(i+1)}$
  end
  return $h_j \leftarrow MSB_{512} M_j^{(10)'}$;
  $H_j = h_j||C_0 \boxplus_{512} (j-1)$
end
return $h_t \leftarrow MSB_{512} M_t^{(10)'}$;
$h'_t = MSB_{256} h_t$

Algorithm 1: MGR Compression Function

$$
\begin{array}{lcllll}
MGR(a) & = & c35c673c & fac55bc3 & 29611897 & 792a7b99 \\
 & & 8d272bc4 & 94c21bbc & 4588f58f & 219ee40b \\
MGR(ab) & = & 7035f7d3 & 565751a6 & 6e96d91f & 602871d4 \\
 & & 2d273736 & 67af6c81 & cdeeede6 & 92927338 \\
MGR(abc) & = & 7796a0f3 & 01f8b6dd & 83078b80 & b028b2c3 \\
 & & 7f11e20f & c1d09b57 & 1dc0e979 & b79c8813
\end{array}
$$

4 Analysis of MGR Hash Function

In this section we will be discussing the analysis of MGR hash function.

4.1 Efficiency of MGR Function

The following table provides the comparison in the efficiency of MGR with GOST-R on an Intel(R) i7-3770 CPU @ 3.40 Ghz processor with 2 GB RAM.

File Size (in MB)	**GOST-R** (in Sec)	**MGR** (in Sec)
1.05	3.51	**1.26**
5.43	17.95	**6.35**
11.10	36.65	**12.77**
16.70	55.22	**19.17**

This shows that the efficiency of MGR is better than GOST-R, it is almost three times faster than GOST-R.

4.2 Avalanche Effect

We have taken an input file M of 1024-bit and computed $MGR(M)$. By changing i^{th} bit of M we can generate M_i files where $1 \leq i \leq 1024$. The hamming distance of each M_i from M is exactly 1. We then calculate $MGR(M_i)$, thus computed the hamming distance d_i, between $MGR(M_i)$ and $MGR(M)$ for $1 \leq i \leq 1024$ and finally computed the hamming distances in the corresponding 32-bit words of the hash values [16]. 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	*GOST-R*	***MGR-hash***
Max	27	25	25	25	25	25	25	28	155	**153**
Min	8	8	9	7	6	7	6	8	97	**99**
Mode	15	17	15	16	17	16	15	15	124	**127**
Mean	15.87	15.96	15.99	16.06	15.91	15.86	15.85	15.96	128.17	**127.47**

Table 1: **Hamming Distances**

To satisfy strict avalanche effect, change in one bit should alter the final hash value by $\approx 50\%$, i.e., each d_i almost should be 128 for $1 \leq i \leq 1024$. We have seen from the results above that d_i's were lying between 99 and 153 and in most of the cases $d_i = 127$. The mean value is also coming to be approx. 128 (i.e., 127.47). The following table and figure show the distribution of the 1024 files with respect to their differences (distance) in bits.

4.3 Preimage resistant

Given the MGR hash value of 256-bit, it is 'hard' to find a message block of 1024-bit. According to the design of MGR if we want to backtrack from hash digest to the last updated register of 1024-bit of the compression function, we need to assume remaining 768-bit of the register and this would require 2^{768} possible choices. After finding the last updated register, we need to go backward from eight AES in a parallel way to find the next register. The key schedule also need to be backtracked for the key input of AES. This process will continue for nine rounds so the number operations would be $\gg 2^{256}$. Thus, MGR is a preimage resistant.

In GOST-R, the preimage attack has been improved upto 6 rounds by *Ma et al.* in [10] where to find a preimage, firstly the multi-collisions has been built and then

Range of Distance	No. of Files	% *GOST-R hash*	% ***MGR-hash***
128 ± 5	517	50.39	**50.49**
128 ± 10	830	80.86	**81.05**
128 ± 15	972	93.85	**94.92**
128 ± 20	1012	99.02	**98.83**

Table 2: **Hamming Distances range of distances**

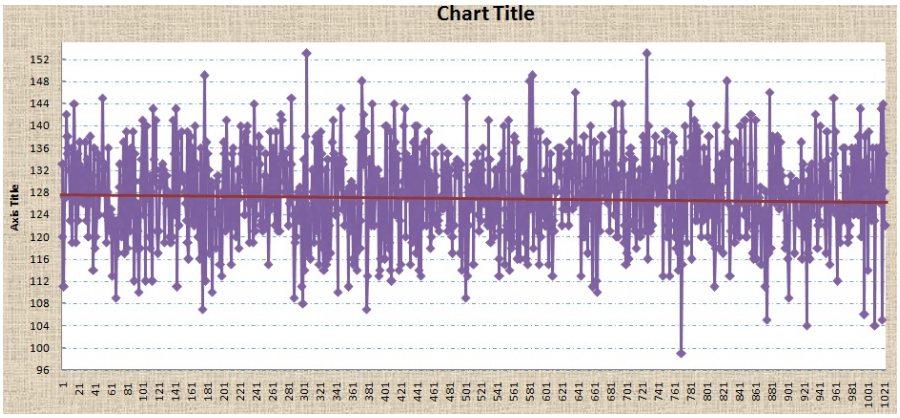

Figure 5: Hamming Distances range of the 1024 files

Meet-in-the-Middle (MitM) preimage attack is been applied but in MGR multi-collsions can not be found (as discussed in section 4.7), thus this attack will not work on MGR.

4.4 Collision resistant

The compression function of MGR is based on wide-pipe (specially double pipe) construction. Thus, it eliminates the multi-collision attack [17], length extension attack, fixed point attack [18] and herding attack [19]. In the paper by *Ma et al.* on cryptanalysis of GOST-R [10], collision attack has been found in GOST-R upto 6.5 rounds by rebound attack [20] and SuperS-box technique [21]. In the compression function of GOST-R, the S-box is not balanced as the S-box of AES, hence the differential probability of GOST-R is higher than that of AES and finding the differential path for inbound phase is easier than AES, thus rebound attack can be worked efficiently on GOST-R than MGR (whose compression function is based on AES like block cipher).

Also we have already seen in avalanche section (section 4.1) that changing one bit in the input generated minimum 6 bits change in the output of the first round and with each round it is getting enhanced and diffusing it to other blocks of the output. Hence it would be difficult to control these bits as the number of rounds increases. Thus, using differential attack to find the collision would be difficult.

4.5 Partial collision resistant

The partial collision resistant means it is 'hard' to find any collisions with in the words of the outputs (i.e., no collisions with respect to words of the outputs). It can be seen from Table 1 that by changing one bit of input, the difference in the outputs of the corresponding words is minimum 6 (not 0 for any word). So MGR is *partial collision resistant* hash function.

4.6 Near-collision resistant

A hash function is called near-collision resistant if it is infeasible to find two inputs with hash output differ in small number of bits i.e.,for two different messages M and M', their MGR hash value is almost same (differ in small number of bits), which means hamming weight of $(H(M) \oplus H(M'))$ is small. To test whether MGR hash is a near collision resistant, we have already seen the result of 1024 input files in Sec.4.2 (avalanche effect). Here a random message M of 1024-bit has been taken and M_i where $1 \leq i \leq 1024$ such that each M_i differs from M in one bit, the minimum weight of their sum of hash digest of 256-bit i.e., minimum hamming weight of $(H(M) \oplus H(M_i))$ is coming out to be 99 and maximum is 153 from the table 1. Hence, we can conclude that MGR Hash is near collision resistant.

4.7 Bit-Variance test

The bit-variance test measures the impact of change in input bits on the hash digest bits. More specifically, given an input message, all the changes of input message bits are taken up (whether they are small or big) and the bits in the corresponding digest are evaluated for each such change. Afterwards, 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. If the probability, $P_i(1) = P_i(0) = 1/2$ for all digest bits $i = 1, \ldots, 256$ then, the *MGR* function has attained maximum performance in view of bit variance test [22]. The bit variance test indeed measures the uniformity of each bit of the output. Since it is computationally infeasible to consider all input message bit changes, we have evaluated the results for only up to 1024 files, viz. $M, M_1, M_2, \ldots, M_{1024}$ which we have generated for conducting avalanche effect, and found the following results:

Digest length = 256 Number of digests = 1025
Mean frequency of 1s (expected) = 512.50
Mean frequency of 1s (calculated) = 511.60
Standard Deviation = 14.42

The experiments were performed on 1024 different messages. Thus MGR passes the bit-variance test. Plotting the probability (Figure 5.4) of each of the bits (256-bit), we see that the average probability is approximately 0.50.

4.8 Security Analysis

All the Merkle-Damgård based constructions like MD hash functions, SHA-1, SHA-2 [17] lead to multi-collisions. Although GOST-R is also based on Merkle-Damgård construction but it has MD-strengthening and also compression function takes three inputs, a block counter N with a chaining variable h and message block M. In GOST-R, if we have found a collision in messages of same block length then this will lead to multi-collision. This weakness has been overcome in MGR. MGR is based on wide-pipe design. Here we are not taking the whole chaining variable as a hash value, infact we are truncating it and taking MSB_{256} of the output as a hash digest. This will provide a better scernario with respect to security. If we would

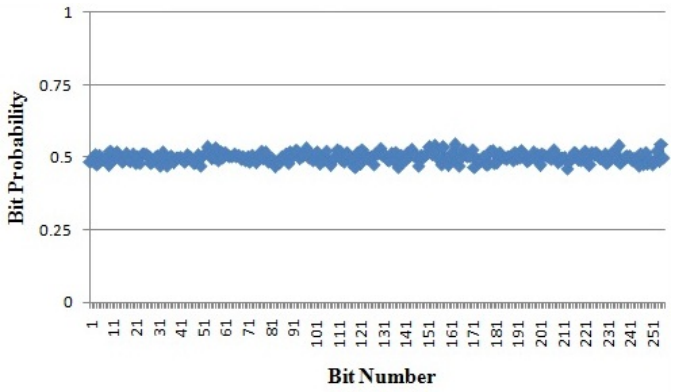

Figure 6: The probability of the bit position

find collision in distinct messages of same block length, we would have a digest of 256-bit but for further collisions we need chaining variable of 512-bit (hash digest $\|$ LSB_{256} of chaining variable) latter we need to assume as for 512-bit chaining variable MSB_{256} is known by hash digest but LSB_{256} is unknown hence need to be assumed. Thus MGR will not give multi-collision. Even if we want to backtrack given hash value, we would be given a hash digest of 256-bit and the chaining variable is 512-bit, so we require to assume 256-bit string to have an updated register i.e., 2^{256} operations. This is an advantage of wide-pipe construction of MGR as it ensures the preimage resitance. In the *chosen plaintext attack* we have few pairs of chosen plaintexts (1024-bit) and corresponding ciphertexts (256-bit) using MGR hash. From these pairs we may try to backtrack a given message but the number of operations would be more than 2^{256}. Hence finding a differential path for differential attack will not be easy.

MGR is more efficient than GOST-R as we have seen in earlier section.

MGR is better than GOST-R from storage point of view too as in MGR message digest is being calculated as soon we hav our first message block, the whole message storage is not needed for calculating the digest of a single block whereas in GOST-R the whole message is needed before dividing into blocks of length 512-bits message as it is padded from the left and blocks are cut from the right.

MGR can have vast applications in digital world. Due to its variable length output (any length between 256-bit to 512-bit), it can be used as pseudorandom generator and for masking. It can be used in digital signature and public-key cryptography. It can play a major role in key derivation for block cipher and stream cipher.

5 Conclusion

We have discussed a dedicated hash function *MGR* whose security is based on its design. It is secured against multicollision attack, fixed point attack, length extension attack and herding attack and gives better efficiency (three times faster) than GOST-R. Here, we have also discussed the analysis of MGR by showing avalanche effect, bit-variance test and other properties posses by cryptographic

hash function. Hence, this hash function, MGR can be a better subsitute for GOST-R and can be used for several many purposes of hash functions.

References

[1] Yaksic .V *A Study of Hash Functions for Cryptography*, SANS Institute, 2002. Available online at

`www.giac.org/paper/gsec/3294/study-hash-functions-cryptography/105433`

[2] Diffie . W & Hellman M.E., *New Directions in Cryptography*, 1976. Available online at

`www.cs.tau.ac.il/~bchor/diffie-hellman.pdf`

[3] Stinson . D . R, *Cryptography - Theory and Practice*, CRC, 2006.

[4] Preneel . B, *Analysis and Design of Cryptographic Hash Functions*, PhD Thesis, Katholieke Universiteit Leuven, 1993. Available online at

`http://www.cosic.esat.kuleuven.be/publications/thesis-2.pdf`

[5] Rompay . B *Analysis and Design of Cryptographic Hash Functions, MAC Algorithms and Block Ciphers*, PhD Thesis, 2004. Available online at

`http://www.cosic.esat.kuleuven.be/publications/thesis-16.pdf`

[6] Daemon .J, Bertoni .G, Peeters .M & G.V. Assche *The Keccak SHA-3 Submission*, 2011. Available online at

`http://keccak.noekeon.org/`

[7] Federal Information Processing Standards Publication 202, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, NIST, 2014. Available online at

`csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf`

[8] Fleischmann . E, Forler . C & M. Gorski, *Classification of the SHA-3 Candidates*, 2008. Available online at

`http://eprint.iacr.org/2008/511.pdf`

[9] Dolmatov . V & Degtyarev . A, *GOST-R Hash Function*, 2013. Available online at

`tools.ietf.org/html/draft-dolmatov-gost34112012-00`

[10] Ma . B, Li .B, Hao . R & Li . X, *Improved Cryptanalysis on Reduced-Round GOST and Whirlpool Hash Function*, 2014. Available online at

`http://eprint.iacr.org/2014/375.pdf`

[11] Damgård .I, *A Design Principle for Hash Functions*, in Advances in Cryptology - CRYPTO'89, LNCS, Volume 0435, Springer, 1990.

[12] Lucks . S, *A Failure-Friendly Design Principle for Hash Functions*, in Advances in Cryptology - ASIACRYPT'05, LNCS, Volume 3788, 2005.

[13] Daemon .J & Rijmen . V, *The Design of Rijndael : AES - The Advanced Encryption Standard*, Springer, 2001. Available online at

`https://autonome-antifa.org/IMG/pdf/Rijndael.pdf`

[14] Federal Information Processing Standards Publication 197, *ADVANCED ENCRYPTION STANDARD (AES)*, NIST, 2001. Available online at

`csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[15] Joux .A, *Algorithmic Cryptanalysis*, CRC Press, 2009.

[16] Dey . D, Shrotriya . N & Sengupta .I, *R-hash : Hash Function Using Random Quadratic Polynomials Over GF(2)*, International Journal of Computer Science & Information Technology (IJCSIT) Vol. 4, No. 6, December 2012. Available online at

`http://airccse.org/journal/jcsit/4612ijcsit12.pdf`

[17] Joux . A, *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions*. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, volume 3152, Heidelberg 2004.

[18] Kelsey . J & Schneier . B, *Second Preimages on n-bit Hash Functions for Much Less Than 2^n Work*, in Proceedings of Eurocrypt 2005, LNCS, Volume 3494, Springer-Verlag, 2005.

[19] Kelsey .J & Kohno . T, *Herding Hash Functions and the Nostradamus Attack*, in Proceedings of Eurocrypt 2006, LNCS, Volume 4004, Springer-Verlag, 2006.

[20] Mendel . F, Rechberger . C, Schläffer . M & Thomsen . S, *The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In: Dunkelman, O. (ed.)*, FSE 2009, LNCS, Volume 5665, Springer, Heidelberg, 2009

[21] Gilbert . H, Peyrin . T, *Super-Sbox Cryptanalysis: Improved Attacks for AES-like Permutations. In: Hong, S., Iwata, T. (eds.)*, FSE 2010, LNCS, Volume 6147, Springer, Heidelberg 2010

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

Round Constants

RC_1 = e5e2c6f57054c7d8370a22521b87647072caedc187be486200d6cb80c10ee7a6
00aa5131862afb2a8d35f62cad8ee9f1b885f53e3c6e19cf128c46366f1deed7
d1000f6a655a3133b3d22c49d00896b28ce5ed0323b571849e65037e489354e4
d554a010376bd00e1f473beef4e318457771a5fdd87145413cdd17212f153125

RC_2 = dc44008813165d3b01349e6a122af360b90d34ad7ea79632513de7123f03d234
27458660047d7b5a4add0d23c9036414d6aab939402dde0c97e0e4e31cc2b898
cc21c63d7ce83e80f945dd2083b3bbc720b4beef76f814b617ec064d8481eb15
3b70583dbc230a99bbb0d20bb760640924b12d6126a5220440ac3eceda8ab0ca

RC_3 = fb38f95363f147e99c0175d7a2c05d71796b20b607b99aa6c56e7127f6ad612f
46a6d699e06ee4d3a46324d8eba938c1dab6b7b1f2841cfeff1585da31c07c8f
b01f5fa60fcd1c769fe2452786eeeeb2e40d9bbbf83562e171322c7549a98183
71a88c5f52fc2ea771ea05b1f1a0097e9b53316dc00bdeed034fe2328fa23073

RC_4 = cffb44be1ca3c5a12350c77547270b81f0b37f1445405e3efd00b368fd19c3b7
d28f1bd4f8b2c5112ca43bf852067e4ef497be55bc8c942e0a74bdb9c3891f32
5d4f1bb7ada2693303905670e699521ee655f4e177f82ad7e4b7add493dbc27a
cbd8ef8d0560e68d9902bf9826c76c609df7c03988e2ff2515052372a9ba8030

RC_5 = 0eaee7ed0ff2467c55a56ef75b0f22ca94cb39a82b2297f67e69ae44a5943542
8510e7eb676d9ed1d1676a72052c5c85950e915a3c1d20ec0fc8506e82e45da5
07d5caa6ad6a8cf448965e915d2cd75b9165f8036caffd4cd1ccaf87b346d860
9bece859cf0565d763f8fc9d289290dc8db8e9346e4120e058f13bdf2bf3a50f

RC_6 = 31bec14f024caa62471e70981a7afe5c23c90f692e2a62175f074fc8e8960e78
0be8f793e190118fdb378dea2cfb473f0bf4513c0a7d7c721b5c988b95ab422e
544e3c29a5000d62779471b3dc0e62852e89ab49269b5bb049b21a44d872c902
8cf7a1b73e1afb1e9715f8e40762ef4568e620e55b8d8b3a0a8e03a4ff6cb5a0

RC_7 = 3555d83ea53a8b3ed5dd58be9deda8c9738c23f0667f5445944173a09df355f3
f1b882b4445ad73088b9bdddc79eff9b93f36105fb72c09b23a78e179abc28a9
5f93c6277311a24d9a4d63b21cb9c170d224b52a2323310561ec9f6357258e03
7b3ca54cdabecdf6cf4792275cd79c7aef6dfe9c724fc8819360cd5eeda6a2b7

RC_8 = b85da96f7e1bb720a5af134be0346e738847eec42c2120dadcaba8ac291ffeaf
2a5daa77c5d8653a4634d7dd88df99536271ad710180aa189d6b83d7ec505c14
4cb7cc0690bdee53e0abd201ec9d5586c751a42d731ffbaa75b845ef35d92985
e8ebedd5329054d2c0a2d9c70db7c1aa3b2b9a9028fd595539c6c670f6fe2c5c

RC_9 = efbe7a61e4a6e32cda8002b83aaee23a3391dfae30f0c28b675740e2eea5fea1
efb7382c633097666120cfd3e5b32f2390733e1d8b58d98a79e65fb0cd54f58c
e332a3788123bdd9e575a0a6cc1988bebb9523081b23e80be8fda1ced61e9521
0db8c7e89131031cb96ec7f7d5d583b3ea402181a04c5eabc11200251d1dee5a