An Efficient Cryptographic Hash Algorithm (BSA)

Subhabrata Mukherjee[1] , Bimal Roy[2], Anirban Laha[1]
[1]Dept of CSE, Jadavpur University, Calcutta 700 032, India
[2]Indian Statistical Institute , Calcutta 700 108, India

subhabrata.mukherjee.ju@gmail.com, bimal@isical.ac.in, anirbanlaha@gmail.com

Abstract – *Recent cryptanalytic attacks have exposed the vulnerabilities of some widely used cryptographic hash functions like MD5 and SHA-1. Attacks in the line of differential attacks have been used to expose the weaknesses of several other hash functions like RIPEMD, HAVAL. In this paper we propose a new efficient hash algorithm that provides a near random hash output and overcomes some of the earlier weaknesses. Extensive simulations and comparisons with some existing hash functions have been done to prove the effectiveness of the BSA, which is an acronym for the name of the 3 authors.*

Keywords : *Cryptography, Hash Function, Random, BSA*

1. INTRODUCTION

A cryptographic hash function takes an input string of arbitrary length and produces a message digest that is of a fixed, short length (e.g. 128 or 256 or 512 bits). The digest is sometimes also called the "hash" or "fingerprint" of the input. Hash functions are used in many situations where a potentially long message needs to be processed and/or compared quickly and also for security purposes. The most common application is the creation and verification of digital signatures.

The two most widely used cryptographic hash functions are the MD5 [1] and SHA-1 [2]. MD5 was designed by Rivest as a strengthened version of MD4 . There had been a lot of tweaks and variants in the MD and the SHA series mostly by increasing the length of the message digest. [3] describes in details of the approach to find collisions in MD5 and break other hash functions like RIPEMD, HAVAL, MD4 and SHA-0 by using differential attacks. This has led to the recent development of many other cryptography hash functions, each having its own strengths and weaknesses, aiming to be the “one” which is secure against birthday attacks, cube testers, differential cryptanalysis and several other attacks. The NIST hash function competition is an open competition held by the US National Institute of

Standards and Technology for a new SHA-3 function to replace the older SHA-1and SHA-2, which was formally announced on November 2, 2007 [6]. NIST selected 51 entries in Round 1 out of which the following 14 algorithms proceeded to Round 2 : Blake[7], Blue Midnight Wish[8], Cubehash[9], Echo[10], Fugue[11], Groestl[12], Hamsi[13], JH[14], Keccak[15], Luffa[16], Shabal[17], SHAvite-3[18], SIMD[19], Skein[20].

In this paper, we have proposed an efficient hash function, which is also a block cipher, where the set of initial values of the initialization vector and all the operations depend solely on the plaintext which leads to an excellent avalanche effect, where flipping a single bit in the plaintext changes more than about 50% bits in the ciphertext. Thus making the hash output nearly random. Extensive simulations have been carried out and comparisons have been made, on various factors like the Runs test, Chi-square test, Entropy, Mean Value and Serial Correlation for testing the randomness of the hash output, with the algorithms selected in Round 2 of the NIST SHA-3 competition to evaluate the performance of this algorithm. The rest of the paper is organized like this. Section 2 discusses some preliminary ideas about the Merkle-Damgard construction of hash functions, the securities for a good hash function and a brief note on randomness tests. Section 3 discusses in details about the proposed BSA hash function. Section 4 gives the simulation results of the various tests conducted on the BSA and its performance comparison with some existing algorithms followed by concluding remarks.

2. PRELIMINARIES

2.1 Merkle -Damgard Construction

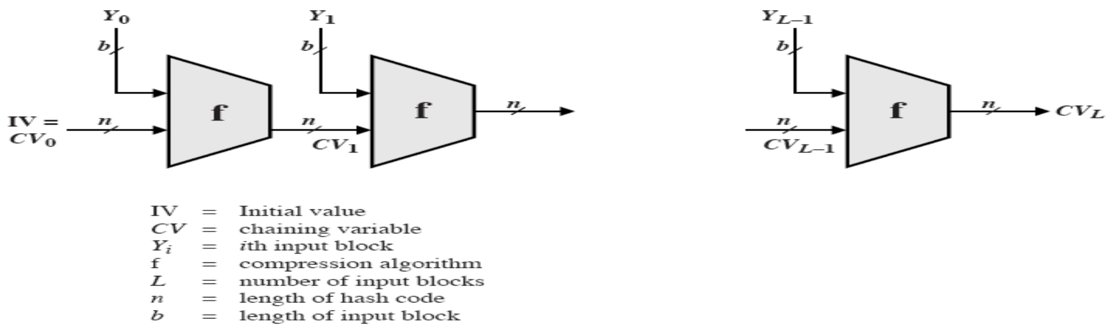

Y_0
b
Y_1
b
Y_{L-1}
b
IV = CV_0
n
f
n
CV_1
f
n
n
CV_{L-1}
f
n
CV_L
IV = Initial value
CV = chaining variable
Y_i = ith input block
f = compression algorithm
L = number of input blocks
n = length of hash code
b = length of input block

Figure 1. Iterative Chaining of Merkle-Damgard Construction

Ralph Merkle [5] and Ivan Damgard [4] proposed an iterative chaining function for block ciphers. In this method the input to each compression function will be an initialization vector / initialization value and a chaining variable and the output will go to the next stage. They independently proved if the compression function is collision resistant, then the hash function will be also. In order to strengthen the above construction they further proposed that the padding should contain the length of the original message. This is called length padding or Merkle–Damgard strengthening. A finalization function is often used in the last stage to further compress the hash output or to increase the avalanche effect. Most widely used cryptographic hash functions like Sha-1 and MD5 use this method. At the heart of the BSA lies this form.

The 2 inputs to the compression function, as describe above, are an intermediate hash value, which is the output of the previous compression function, and a message block. If the compression function is an ideal one, the intermediate hash values should be random. Thus one of the measures to compare or evaluate the performance of a hash function is to perform randomness tests on the hash output to find the level of randomness of the hash function.

2.2 Tests for Randomness [21]

a) Chi-square Test

This is used to test the validity of a distribution assumed for a random phenomenon. The test evaluates the null hypotheses H_0 (that the data are governed by the assumed distribution) against the alternative (that the data are not drawn from the assumed distribution).

b) Run Test

This test is based on based on the frequency of run-lengths (a run is a sequence of consecutive digits)

c) Frequency Test (Mean Value Test)

This test checks that each symbol occurs with equal frequency (for a binary string, proportion of 0's and 1's should be 0.5 each)

d) Serial-Correlation test

Correlation coefficients appear frequently in statistics; if we have n quantities U0, U1, U2….Un-1, the correlation coefficient between them is a measure of the amount lJj+l depends on Uj.

e) Entropy Test

The information density of the contents of the file, expressed as a number of bits per character. An entropy value of ‘1’ (for a binary bit stream) indicates that the file is extremely dense in information—essentially random. Hence, compression of the file is unlikely to reduce its size.

3. DESCRIPTION OF THE BSA HASH FUNCTION IN DETAILS

3.1 Description of the Components and Essential Operations in the BSA function

3.1.1 Padding

a) The first block to the hash function is length padded. The leftmost 4 words contain the number of 1’s in the message. The rightmost 4 words contain the number of 0’s in the message. The remaining middle 8 words contain the length of the message. The size of each word is 32 bits. The size of each block that is input to the hash function is 512 bits.

b) The input plaintext message is broken into 448 bit blocks. The 448-bit block message M_i is extended to a 512-bit block B_i by padding. Here we define two 16-bit strings L_i and R_i where,

L_i = bit position of 1’s in M_i xor-ed to each other

R_i = bit position of 0’s in M_i xor-ed to each other

(bit positions vary from 1-448 in each block).Thus each block input to the hash function is defined as, $B_i = L_i R_i M_i R_i L_i$ making the total block length 512 bits.

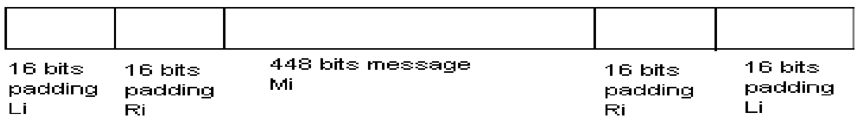


Figure 2: Construct of the 512-bit block B_i after padding

c) In case the last block is less than 448 bits, then it is padded with 1's or 0's if the total number of 1's in the entire plaintext is odd or even accordingly.

3.1.2 Crossover

The crossover point or pivot in each block is chosen as the index corresponding to the number of 1's in that block. Thus the index can vary from 1 to 511 (no crossover for all or no 1's). The strings on both the sides of that index are swapped to perform the crossover. The figure below shows a string x1x2, which is, crossed over a chosen crossover point i giving the resultant string x2x1.

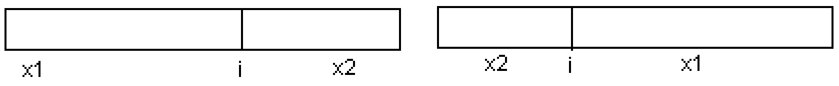


Figure 3. Crossover

3.1.3 Two Parallel Iterative Chains

The BSA consists of 2 iterative chains, which gives it added strength and security and also speed due to parallelism. Each chain, similar to the Merkel-Damgard construction in Figure 1, has an iterative structure consisting of several compression functions. The final result is obtained by concatenating the output of the 2 chains. The output of each chain is a 256-bit value and hence the final hash output is 512-bits in size.

3.1.4 Initialization Vector

There are 2 initialization vectors corresponding to each of the 2 Iterative Chains. The initialization vectors IV1 is taken as the 256 bits (in order) at odd places in Block 1 and IV2 is taken as the 256 bits (in order) at even places in Block1.

3.1.5 Compression Functions

Each compression function consists of 16 Rounds (the number of Rounds is adjustable). In each Round a different bitwise operation is used. A function lookup table is used to determine the operation to be used in each Round. The Block number, which goes as one of the inputs in each compression function, and the Round number give the index to the table. Each compression function operates on a 256-bit value and outputs a 256-bit value.

3.2 Working of the Algorithm in Details

Step 1: Preprocessing

The input message is divided into 448 bit blocks. The last block may need to be padded (Sec 3.1.1.c). The first block is a dummy block that is length padded (Sec 3.1.1.a).

Step 2: Block Processing

Each 448-bit block needs to be preprocessed to make it a 512-bit block (Sec 3.1.1.b) B_i. Crossover (Sec 3.1.2) is performed in the resultant block B_i. The resulting block C_i after crossover is also 512-bits in size. The words in C_i in even position, taken in order, form X_{i1} and the words in odd position, taken in order, form X_{i2}.

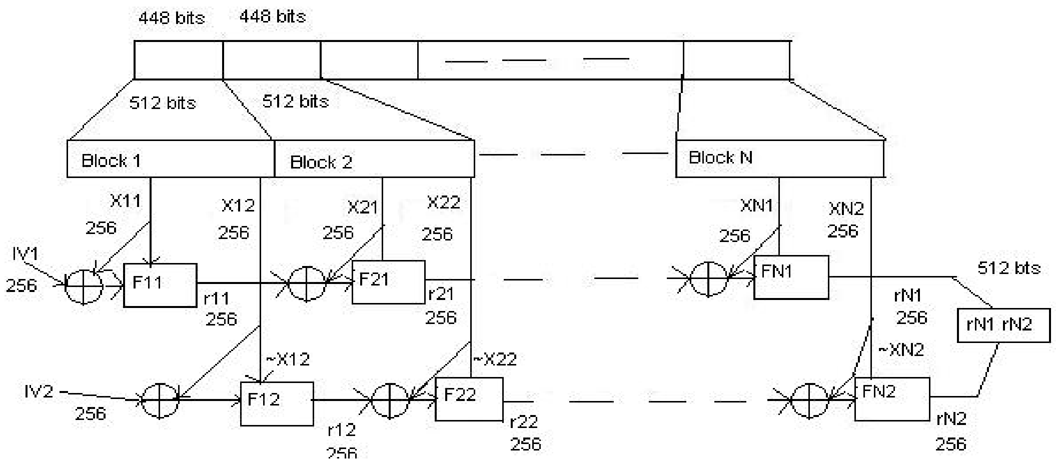


Figure 4 : Block and Chain Processing

Step 3: Chain Processing

As described in Section 3.1.3, there are 2 iterative chains. In chain 1, the compression functions, used iteratively, are denoted by F_{i1} and that used in chain 2 are denoted by F_{i2}.

For the First Block C_1, the input to F_{11} is X_{11} and (IV1 $\oplus$ X_{11}) and the input to F_{12} is ($\sim X_{12}$) and (IV2 $\oplus$ X_{12}). The output hash value of F_{i1} is denoted by r_{i1} and that of F_{i2} is denoted by r_{i2}.

For any other Block C_i (obtained in Step 2):

a. The input to F_{i1} is $X_{i\,1}$ and ($X_{i1} \oplus r_{(i-1)\,1}$).

b. The input to F_{i2} is ($\sim X_{i2}$) and ($X_{i2} \oplus r_{(i-1)\,2}$).

Step 4: Compression Function Processing

Each compression function F_{i1} or F_{i2} consists of 16 rounds each. The function to be used in each round is determined by using the Block number and Round number as an index to a function lookup table.

The function lookup table (Fig. 6) gives the possible input and output bits for a bitwise operation for a specific Round. For example, in the 5th Round of any compression function, if the inputs bits for a bitwise operation are 00, 01, 10, 11 then the outputs bits are 0, 1, 0, 1 respectively i.e. it represents a bitwise XOR operation.

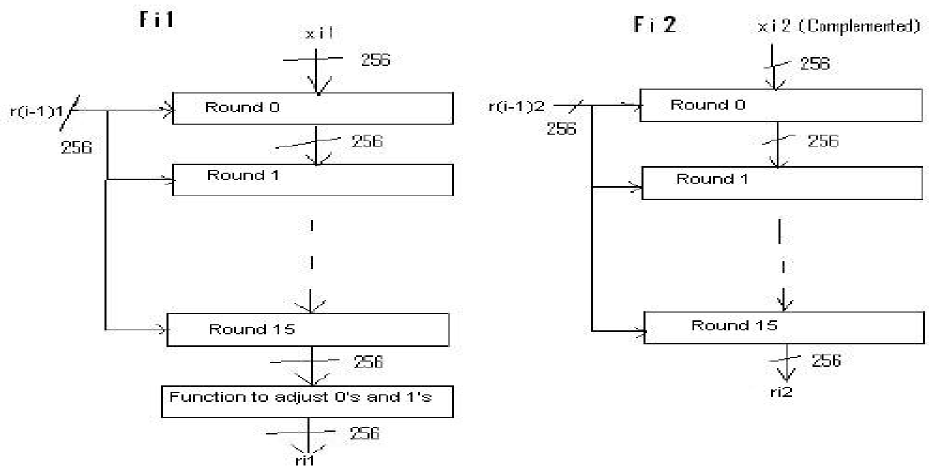
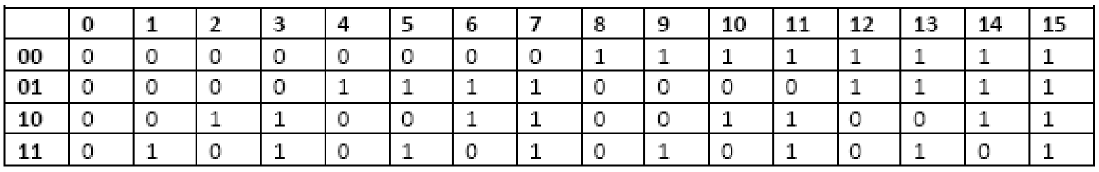


Figure 5 : Compression Function Processing



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 6 : Function Lookup Table

Step 4.1: Chain 1

Step 4.1.1: Basic Operation

The basic function number for Block i and Round j in compression function F_{i1} is determined by (i + j) mod 16. This is used as an index to the function lookup table. Let the bitwise Round function be f1 for the jth Round for a Block i. We have taken the number of Rounds and the number of functions in the lookup table both as 16.

We define 3 variables a, b and c such that a=(i+j) mod 256, b=(i*j) mod 256 and c= (i^j) mod 256.

The result from higher round is X_{i1} for Round 0 and $r_{(i-1)\,1}$ for any other Round..

We now define the following variables:

Reflection11 = f1 ($r_{(i-1)\,1}$ in Gray code, X_{i1} in reverse order of bits).

Reflection21 = f1 ($r_{(i-1)\,1}$ in reverse order of bits, X_{i1} in Gray code).

Reflection31 = f1 ($r_{(i-1)\,1}$, X_{i1})

Rotation11 = Rotate Right (X_{i1} by a bits)

Rotation21 = Rotate Right (X_{i1} by b bits)

Rotation31 = Rotate Left ($r_{(i-1)\,1}$ by c bits)

Result11= Reflection11 | Rotation11

Result21= Reflection21 & Rotation21

Result31= ~ (Reflection31 & Rotation31)

Final result of this round, $r_{i1} =$ (Result11 $\oplus$ Result21 $\oplus$ Result31).

Step 4.1.2: Adjusting the bits

Let Count1 and Count0 be the number of 1's and 0's in r_{i1}.

Case 1: If Count1 > Count0,

Then first check the 7th bit of the 1st word of $r_{i1.}$ If it is 1 then flip it and check the 17th bit, else do the same in the next word of r_{i1}. If the 17th bit is a 1, then flip it and check the 27th bit of that word or else check the next word. Continue in this fashion till either Count1= Count0 or the end of r_{i1} is reached.

Case 2: If Count0 > Count1,

Then first check the 1st bit of the 1st word of $r_{i1.}$ If it is 0 then flip it and check the 11th bit, else do the same in the next word of r_{i1}. If the 11th bit is a 0, then flip it and check the 21st bit of that word or else check the next word. Continue in this fashion till either Count0= Count1 or the end of r_{i1} is reached.

Step 4.2: Chain 2

The basic function number for Block i and Round j in compression function F_{i2} is determined by (i + (j+49) mod 50) mod 16. This is used as an index to the function lookup table. Let the bitwise Round function be f2 for the jth Round for a Block i.. We have taken the number of rounds and the number of functions in the lookup table both as 16.

We define 3 variables a, b and c such that a=(i+j) mod 256, b=(i*j) mod 256 and c= (i^j) mod 256.

The result from higher round is X_{i2} for Round 0 and $r_{(i-1)\,2}$ for any other Round.

We now define the following variables:

Reflection12 = f2 ($r_{(i-1)\,2}$ in Gray code, X_{i2} in reverse order of bits).

Reflection22 = f2 ($r_{(i-1)\,2}$ in reverse order of bits, X_{i2} in Gray code).

Reflection32 = f2 ($r_{(i-1)\,2}$, X_{i2}).

Rotation12 = Rotate Right (X_{i2} by a bits)

Rotation22 = Rotate Right (X_{i2} by b bits)

Rotation32 = Rotate Left ($r_{(i-1)\,2}$ by c bits)

Result12= Reflection12 | Rotation12

Result22= Reflection22 & Rotation22

Result32= ~ (Reflection32 & Rotation32)

Final result of this round, r_{i2} = ~ (Result12 $\oplus$ Result22 $\oplus$ Result32).

Step 5: Iterate and Final Result

Repeat steps 2-5 for every 448-bit block in the input plaintext message.

The final hash output is given by concatenating the output hash value of each chain i.e. the final hash value is (r_{N1} r_{N2}), if N is the total number of blocks in the input plaintext message.

Step 6: END

4. SIMULATION RESULTS

Extensive simulations have been done to compare and evaluate the performance of the BSA. The tests performed have been categorized as :

i) Collision Tests ii) Avalanche Effect Test iii) Randomness Tests

i) Collision Tests

20 lakh random strings were generated each having 0-35,000 bits. The randomness of each string was verified using the randomness tests (Section 2.2). The level of significance in Runs and Chi-square tests were taken at .05% level of significance.

Further simulations have been done with 20 lakh strings, each differing from the other by a single bit hamming distance, each having 10000 bits.

No collisions have been found in both cases.

ii) Avalanche Effect Test

In this test 50,000 to 1 lakh strings each differing from the other by a single bit (Hamming Distance =1) were tested using BSA. The hash outputs of the input strings were compared on the basis of their relative hamming distance. The minimum hamming distance between the hash outputs were found to be 206 bits, the maximum was 288 bits and the average hamming distance was 250 bits. For random strings of arbitrary length, the average hamming distance was 253 bits.

The BSA was found to perform better than the 12 algorithms (listed in the Section 1 that proceeded to the Second Round in the NIST SHA-3 competition) when we compared the average change in the hamming distance of their hash outputs on input strings with hamming distance 1.

Table 1 below, gives the hamming distance between the hash codes of each algorithm for input strings C0 (Hex) and 80 (Hex) with hamming distance 1. Table 2 shows their BSA hash output in hex.

iii) Randomness Tests

For an ideal hash function the output should be as random as possible. In Section 2.2 we have specified a few tests for evaluating the randomness of a hash function. For a fully random hash function, the entropy of the output hash bitstream should be 1, the serial correlation - coefficient should be 0.0 and the mean value should be 0.5. We have taken the help of a widely used software for conducting the randomness tests developed by John Walker called the ENT [22] for performing the Entropy, Mean value and Serial Correlation tests.

Algorithm Name	Hamming Distance between output Hash
BSA	231
FUGUE	187
GROESTL	164
SHABAL	162
SHAVITE-3	154
HAMSI	152
JH	151
BLUE	142
ECHO	136
SKEIN	136
BLAKE	136
LUFFA	130
KECCAK	124

Table 1: Hamming Distance between Hash Codes of Algorithms for Input Strings C0 and 80

Message (hex)	Output Hash Code (hex)
C0	`1dd86640 5931a2e3 932c2494 ce21cf63 62d4823b b31a9858 12502e02 e420873e c9316b5b 6b6f8f66 3e8b6d2d 9e6c488e a549f56e cdab4f49 3379cb87 4c756713`
80	`834c4014 4d773438 89943f2c 2d28570d 999bc461 91812aa9 83ee1484 e34040a5 610bc1fb 2fa1c41f 3d8b2f3c a8a9258c 8e8179bd bf95592e 737ee5aa c4b5c91c`

Table 2: BSA Hex Code of Strings C0, 80

Algo Name	Entropy Value	Mean Value	Serial Corr. Value	Runs Test	Chi square Test
Blake	0.98847	0.4369	0.0040	94.9	98
Blue Midnight Wish	0.98856	0.4371	0.0039	94.4	98
Echo	0.98854	0.4371	0.0038	96.5	99
Fugue	0.98853	0.4370	0.0031	95.2	99
Grostl	0.98850	0.4370	0.0035	95.4	98
JH	0.98856	0.4371	0.0034	94.9	99
Shabal	0.98857	0.4371	0.0032	95.1	98
Shavite-3	0.98844	0.4368	0.0036	94.7	98
Hamsi	0.98850	0.4370	0.0035	95.4	99
Keccak	0.98852	0.4370	0.0041	95.1	99
Luffa	0.98863	0.4373	0.0036	94.6	99
Skein	0.98858	0.4372	0.0036	95.2	99
BSA(Input String <= 2Kb)	0.98892	0.4381	0.0006	94.3	98.5
BSA (any Input String size)	0.98854	0.4371	0.0027	93.5	97.5

Table 3: Randomness Test Results between Hash Codes of Different Hash Function

Further, we have performed the Runs test and Chi-square tests ourselves at 0.05% and 0.01% level of significance. The tests were conducted on 2600 strings each having 0-35,000 bits. Table 3 gives the comparison between different algorithms and the BSA on the results of the randomness tests conducted on the output hash bitstream of each algorithm. The entropy value is measured per bit, the Runs test and Chi-square test results of each algorithm indicate the percentage of the total number of hash output values that is accepted by those tests at 0.05% and 0.01% level of significance.

5. Conclusion and Future Work

It is clearly evident from the simulation results that BSA performs the best when all the factors of randomness tests and avalanche effects are combined. Further tests are underway in evaluating the strength of the BSA against differential cryptanalysis and other cryptographic attacks. The BSA showed promising results for all the tests conducted so far.

References:

[1] R.L. Rivest. The MD5 message-digest algorithm, Request for Comments (RFC 1320), Internet Activities Board, Internet Privacy Task Force, 1992.

[2] FIPS 180-1. Secure hash standard, NIST,US Department of Commerce,Washington D.C,SpringerVerlag, 1996

[3] How to Break MD5 and Other Hash Functions Xiaoyun Wang and Hongbo Yu , Eurocrypt,2005

[4] Damgard I, “A design principle for Hash functions Advances in Cryptology-Crypto’89”, Lecture Notes in Computer Science. Springer-Verlag, no.435, pp. 416- 427, 1990.

[5] Merkle R, “One Way Hash Functions and DES Advances in Cryptology, proceedings of CRYPTO 1989”, Lecture Notes in Computer Science. Springer- Verlag, no.435, pp. 428 - 446, 1990.

[6] The First SHA-3 Candidate Conference, K.U. Leuven, Belgium, Feb. 25-28, 2009

[7] Jean-Philippe Aumasson; JPA, Luca Henzen, Willi Meier, Raphael C.-W. Phan, “SHA-3 Proposal Blake”, Candidate to the NIST Hash Competition, 2008

[8] Svein Johan Knapskog; Danilo Gligoroski, Vlastimil Klima, Mohamed El-Hadedy, Jørn Amundsen, Stig Frode, “Blue Midnight Wish”, Candidate to the NIST Hash Competition, 2008

[9] D.J Bernstein, “Cubehash”, Candidate to the NIST Hash Competition, 2008

[10] Henri Gilbert; Ryad Benadjila, Olivier Billet, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, Yannick Seurin "SHA-3 Proposal: ECHO", Candidate to the NIST Hash Competition, 2008

[11] Shai Halevi, William E. Hall, and Charanjit S. Jutla, “Fugue”,Candidate to the NIST Hash Competition, 2008

[12] Søren Steffen Thomsen, Martin Schläffer, Christian Rechberger, Florian Mendel, Krystian Matusiewicz, Lars R. Knudsen, Praveen Gauravaram, “Groestl”, Candidate to the NIST Hash Competition, 2008

[13] Özgül Kücük, "The Hash Function Hamsi" , Candidate to the NIST Hash Competition, 2008

[14] Hongjun Wu, “JH”, Candidate to the NIST Hash Competition, 2008

[15] Guido Bertoni, Joan Daemen, Michaël Peeters , Gilles Van Assche, “Keccak”, Candidate to the NIST Hash Competition, 2008

[16] Dai Watanabe; Christophe De Canniere, Hisayoshi Sato,"Hash Function Luffa: Specification", Candidate to the NIST Hash Competition, 2008

[17] Jean-François Misarsky; Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Marìa Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, Marion Videau "Shabal, a Submission to NIST’s Cryptographic Hash Algorithm Competition",, Candidate to the NIST Hash Competition, 2008

[18] Eli Biham; Orr Dunkelman. "The SHAvite-3 Hash Function", Candidate to the NIST Hash Competition, 2008

[19] Gaetan Leurent, “SIMD”, Candidate to the NIST Hash Competition, 2008

[20] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker, “Skein”, Candidate to the NIST Hash Competition, 2008

[20] Donald E. Knuth, “The Art of Computer Programming”, Second Edition, pp 39-45,59-60,pp 65-68,pp 70-71

[21] John Walker, ENT Program, http://ftp.fourmilab.ch/random/