International Journal of Computer Science and Applications, ©Technomathematics Research Foundation
Vol. 5, No. 4, pp 11 - 25, 2008

# JERIM-320: A NEW 320-BIT HASH FUNCTION COMPARED TO HASH FUNCTIONS WITH PARALLEL BRANCHES

SHEENA MATHEW

*Department of Computer Science,*
*Cochin University of Science and Technology,*
*Kochi, Kerala, India.*
*Email: sheenamathew@cusat.ac.in*

K. POULOSE JACOB

*Department of Computer Science,*
*Cochin University of Science and Technology,*
*Kochi, Kerala, India.*
*Email: kpj@cusat.ac.in*

This paper describes JERIM-320, a new 320-bit hash function used for ensuring message integrity and details a comparison with popular hash functions of similar design. JERIM-320 and FORK-256 operate on four parallel lines of message processing while RIPEMD-320 operates on two parallel lines. Popular hash functions like MD5 and SHA-1 use serial successive iteration for designing compression functions and hence are less secure. The parallel branches help JERIM-320 to achieve higher level of security using multiple iterations and processing on the message blocks. The focus of this work is to prove the ability of JERIM 320 in ensuring the integrity of messages to a higher degree to suit the fast growing internet applications.

*Keywords:* Hash code; Message integrity; SHA-family; RIPEMD-family; FORK-256.

## 1. Introduction

In recent years, due to the prospering use of internet applications, ensuring confidentiality, integrity and authenticity of information is of increased importance for secure data transmission. When two parties are communicating over an insecure channel, they need a method by which the original information sent by the sender can be accepted by the receiver without an uncertainty on possible alteration or leakage. The integrity of the message can be verified by the hash functions which involves all the bits of the whole message. It accepts the variable size message as input and produces a fixed size output as the hash code. A change in any bit or bits in the message results in change in the hash code thus providing an indication of message tampering. When a person “A” sends a message to another person “B”, the hash code is computed using the hash function and appended to the message. After receiving the message, B re-computes the hash code using the same hash function and compares with the original hash code. If both are the same, then B can confirm that the message has started off from the intended sender and it has not been tampered with, during the transmission.

The most important uses of hash functions are in the authentication of information and as a tool for digital signature schemes. Security of the digital signature depends on the cryptographic strength of the underlying hash functions. Hash functions also have other applications such as in e-cash and in many other cryptographic protocols. The succeeding paragraphs present the observations of an overall review of cryptographic hash functions.

(1) Properties [Stallings, (2003)] necessary for hash functions:

(i) Hash functions can be applied to messages of any length.

(ii) It produces an output of fixed length.

(iii) For any given x, it is easy to compute H(x) making both the hardware and software implementation easy.

(iv) For any given value h, it is computationally infeasible to find x such that H(x) = h. (Preimage resistance)

(v) For any given block x, it is computationally infeasible to find y ? x with H(y) = H(x). (Second preimage resistance)

(vi) It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). (Collision resistance).

(2) For an ideal hash function with an m-bit output, finding a preimage or a second preimage requires about $2^m$ operations and the fastest way to find a birthday or square root attack is approximately $2^{m/2}$ operations [ Stallings, (2003)].

(3) Most popular hash functions are designed using Merkle-Damgaard model [Damgard, (1989)], [Merkle, (1989)]. This model simplifies the management of large inputs and produces a fixed length output using a function HF. The message is viewed as a collection of m-bit blocks:

$M = M_{[1]}..M_{[n]}$ with $M_{[i]}$ = m bits for i=1, 2, …., n.

The hash function H can be described as follows:

$HF_0 = IV$; $HF_i = f(HF_{i-1}, M_{[i]})$, where $1 = i = n$; $H(M) = HF_n$.

Here f is the compression function of H, $HF_i$ is the chaining variable between stage i-1 and stage i, and IV denotes the initial chaining value. This iterative construction in the model provides a moderate goal of extending the domain of collision resistant functions. Many hash functions such as MD4 [Rivest, (1990)], MD5 [Rivest, (1992)] and SHA-family [NIST-FIPS-180-2, (2002)] are based on this idea.

## 2. Motivation and Design Factors

It has been observed that the successful use of cryptographic algorithms for detection of file tampering lies in the fact that any small change in the source file will result in a significant change in the signature. MD5, SHA1 and RIPEMD algorithms are popularly used for generating hash codes. But these algorithms have been "broken" at various levels [Biham *et al*., (2005)], [Chabaud and Joux, (1998)], [Dobbertin, (1996)], [Biham and Chen, (2004)], [Wang and Yu, (2005)]. Collisions in the hash code have proved that a file may be modified without a corresponding change in the hash code. Generally a function which has a good diffusion property can not be so light, but most step functions have been developed to be light for efficiency. This is why MD4 type hash functions including SHA-1 are vulnerable to Wang et. al.'s collision finding attacks [Wang and Yu, (2005)]. If a longer hash function such as RIPEMD-320 or SHA-512 is used, the collisions are less likely and the benefits of greater security supersedes the computational compromise of the longer hash function.

The SHA-2 hash functions are quite resistant against those attack techniques which have been used to attack MD4, MD5 and SHA-1. The design of SHA-2 is fragile; even marginal modifications of the hash functions turned out to generate major weakness. The SHA-2 functions are a possible short term alternative to SHA-1. No attacks against SHA-2 functions have been noticed.

An alternative to this is RIPEMD-family [Dobbertin *et al.*, (1996)], which has a somewhat different approach for designing a secure hash function. The attacker who tries to break members of RIPEMD-family should try simultaneously at two ways where the message difference passes. This design strategy is still considered successful in so far as no effective attack on RIPEMD-family has been reported except the first proposal of RIPEMD. The RIPEMD-family has heavier hash functions compared to MD4-family. For example the first proposal of RIPEMD consists of two lines of MD4. The number of steps of the later version RIPEMD-160 is also almost same as that of SHA-0. No attack against RIPEMD-160 or RIPEMD-320 has been reported. Another new hash function FORK-256 [Hong *et al.*, (2006)] is also based on the RIPEMD design.

As a result of a large number of attacks on hash functions such as MD5 and SHA-1 of the so called MD4 family, and also general attacks on the typical construction method [Damgard, (1989)], [Merkle, (1989)], there is an increasing need for developing alternate designs based on new principles for future hash functions. Several attacks on hash functions are focused on alleviating the difference of intermediate values which are caused by the difference in the message. In this context, a hash function can be considered secure, if it is computationally hard to alleviate such difference in its compression function.

Based on these factors we have selected hash algorithms with parallel branches, a RIPEMD based design for comparing with the novel hash function JERIM-320. In the design criteria, more emphasis is given to security over speed because of the practically negligible effect of increase in the time requirement even though it is considered as one of the measures of performance. The efficiency of the new hash function is its design based on potential parallelism.

On the basis of these observations, a new hash function JERIM-320 has been designed, with focus on the following attributes:

- It should be highly secure
- It should have a higher hash length to resist against the birthday attack.
- It should have a structure resistant to all known attacks including Wang et. al's attack.
- It should have a reasonable performance with respect to speed of operation.

The size of the hash value, and that of the intermediate state, is selected as 320-bits. This value has been chosen for the following reasons:

- Since we use 32 bit words, the size should be a multiple of 32.

• Most of the successful shortcut attacks on existing hash functions are found to be at the intermediate state rather than at the final value. The attacker typically chooses two colliding values for an intermediate block, and this propagates to a collision of the full function. But, these attacks would not have been successful, if the intermediate values were larger.

## 3. Description of JERIM-320

The basic notations used in JERIM-320 are shown in Table 1.

Table 1. Basic notations in JERIM-320

| Notation | Description |
|---|---|
| X ^Y | Addition of X and Y modulo 2 (XOR) |
| X + Y | Addition of X and Y modulo $2^{32}$ |
| X V Y | Bitwise OR operation of X and Y |
| X ? Y | Bitwise AND operation of X and Y |
| ¬X | Bitwise NOT operation of X |
| X<<<n | Bit-rotation of X by n bits to the left |

### *3.1. Input Block Length and Padding*

An input message is processed as 512-bit blocks. Padding is used to make the length of the original message equal to a value which is 64 bits less than the exact multiple of 512 bit. The padding consists of a single 1-bit, followed by as many 0-bits, as required. After padding is added, the original length of the message is calculated and added at the end of the message as a 64-bit value. In the case of a really long message, the length of the message is calculated as the original message length modulo $2^{64}$.

### *3.2. Structure of JERIM-320*

Fig. 1. shows the outline of the compression function of JERIM-320. It consists of four parallel branches B1, B2, B3 and B4. The initial chaining variable $CV_i$ is given as input to the compression functions. $CV_i$ consists of 10 registers A,B,C,D,E,F,G,H,I and J. These chaining variables in each branch are initialized as given below.
A1=A2=A3=A4=0xb54ff53a
B1=B2=B3=B4=0x67452801
C1=C2=C3=C4=0xabcdab84
D1=D2=D3=D4=0xc2d3e0f1
E1=E2=E3=E4=0x2e72d96c
F1=F2=F3=F4=0x4ab0cd91
G1=G2=G3=G4=0x9a056873
H1=H2=H3=H4=0x5ca28c67
I1=I2=I3=I4=0xa14fe235
J1=J2=J3=J4=0x863d421c

Each successive 512-bit message block M is divided into sixteen 32-bit sub blocks

$M_0$, $M_1$, …, $M_{15}$ given as $S_i(M)$ as input to all four branches and the following computation is done to update $CV_i$ to $CV_{i+1}$ as
$CV_{i+1}=CV_i$^ ((B1output ^ B2output) + (B3output ^ B4output)). Finally the message is transformed into the 320-bit hash value.

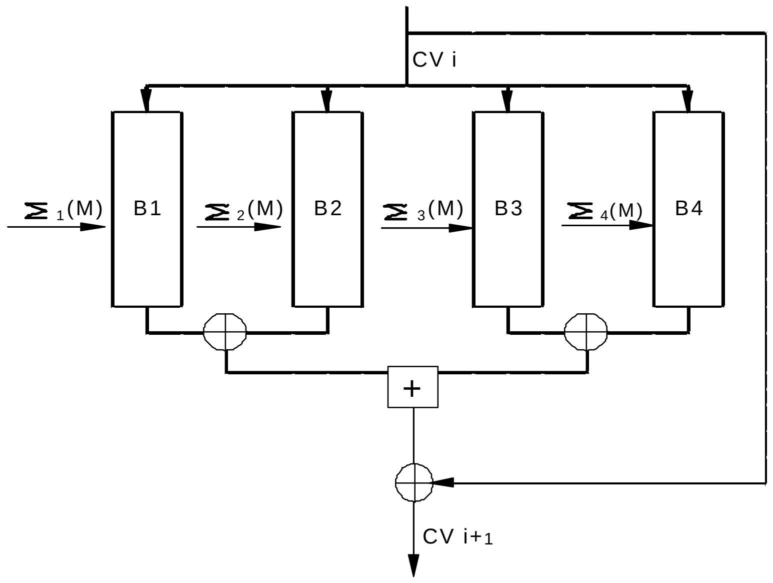

Fig. 1. Outline of the Compression Functions of JERIM-320

### 3.3. Single Step Operations

Five rounds are used in JERIM-320 for each 512-bit message block. The sixteen 32-bit sub blocks of the 512-bit block in each round are processed in four parallel branches. The inputs to each single step operations are the sixteen sub blocks, the chaining variables A1, B1,…J1, A2, B2, …J2, A3, B3,….J3, A4, B4,…..J4 of each branch and the constants $K_{[t]}$. Order of message words in each branch and each round is shown in Table 2 and Table 3. Shift values, Boolean functions and Constants in each branch and each round are shown in Table 4, Table 6 and Table 7 respectively. There are 16 single step iterations as shown in Fig. 2 in each round and in all the four branches. The output of each iteration is copied again into the chaining variables A1, B1,…J1; A2, B2, …J2; A3, B3,….J3; A4, B4,…..J4 and so on.

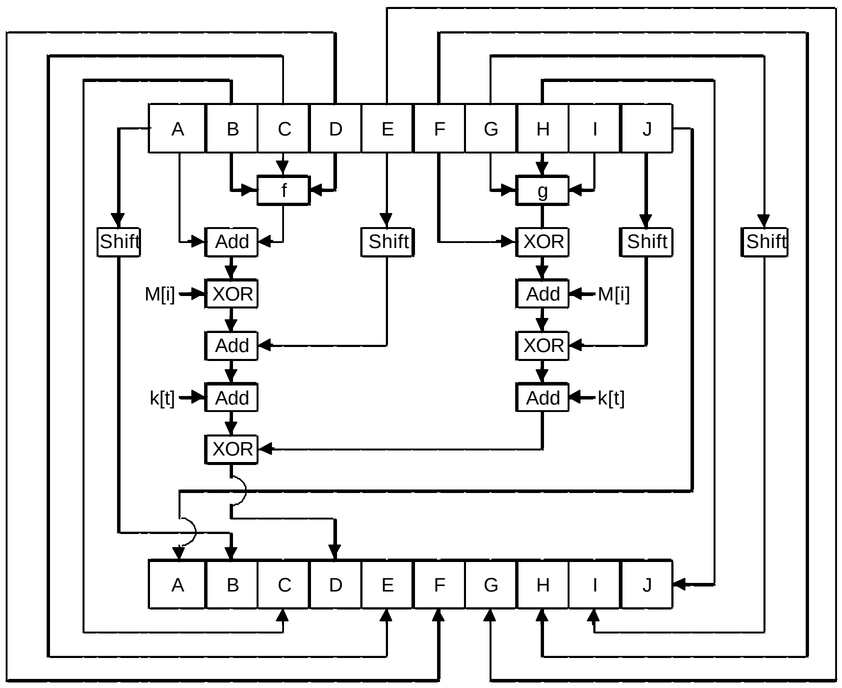

Fig. 2. A Single step Operation of JERIM-320

### *3.4. Order of the message words*

The order in which the blocks are combined is important to prevent the collisions. In order to resist against Wang et. al, different message orderings have been used in different branches as shown in Table 2.

Table 2. Order rule of message words in different branches

| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $B1_{(i)}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $B2_{(i)}$ | 15 | 12 | 8 | 10 | 9 | 14 | 11 | 13 | 4 | 3 | 7 | 5 | 6 | 1 | 0 | 2 |
| $B3_{(i)}$ | 7 | 14 | 15 | 2 | 11 | 1 | 5 | 10 | 9 | 0 | 3 | 4 | 12 | 8 | 13 | 6 |
| $B4_{(i)}$ | 4 | 8 | 3 | 12 | 6 | 9 | 13 | 0 | 1 | 7 | 14 | 15 | 2 | 11 | 5 | 10 |

The following conditions were considered for defining the order of message words [Hong *et al.*, (2006)]:

- Each word is applied twice in the upper and lower parts of the table.
- Each word is applied twice in the left and right parts of the table
- Hence each word is considered 4 times and is indexed by 0 to 15.

To make the attacks more difficult, the order of message in each round for the four branches are considered differently by considering cyclic shift of i as shown in Table 3.

Table 3. Message ordering in different rounds

| Round | Message ordering using cyclic shift of i |
|---|---|
| Round1 | i=0 to 15 |
| Round2 | i=3 to 15, i=0 to 2 |
| Round3 | i=7 to 15, i=0 to 6 |
| Round4 | i=9 to 15, i=0 to 8 |
| Round5 | i=13 to 15, i=0 to 12 |

Each column of its argument is shifted cyclically and independently, so that column i is shifted left by i positions.

## *3.5. Shifts*

The variable shift values as shown in Table 4 provide better immunity against attacks such as differential collision [.Chabaud and Joux, (1998)]. The generalization of inner collisions to a full compression seemed to be harder with variable shift amounts. The design criteria are the following [Dobbertin *et al.*, (1996)]:

- The shifts are chosen between 5 and 15.
- Every message block should be rotated over different amounts.
- Not too many shift constants should be divisible by four.

Since the message order in each round for the four branches are considered differently by considering cyclic shift of i, the order in which shift constant is used in each branch and in each round is also varying.

Table 4. Amount of shifts in each round for different message blocks.

| Round | $M_0$ | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ | $M_6$ | $M_7$ | $M_8$ | $M_9$ | $M_{10}$ | $M_{11}$ | $M_{12}$ | $M_{13}$ | $M_{14}$ | $M_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 15 | 14 | 12 | 13 | 9 | 8 | 7 | 6 | 11 | 12 | 14 | 15 | 6 | 5 | 8 | 9 |
| 2 | 14 | 15 | 13 | 11 | 8 | 7 | 9 | 5 | 12 | 14 | 13 | 11 | 5 | 6 | 7 | 8 |
| 3 | 13 | 12 | 14 | 15 | 7 | 5 | 6 | 9 | 13 | 15 | 12 | 14 | 9 | 8 | 5 | 7 |
| 4 | 12 | 13 | 11 | 14 | 6 | 6 | 5 | 8 | 14 | 13 | 11 | 12 | 8 | 7 | 6 | 5 |
| 5 | 11 | 11 | 15 | 12 | 5 | 9 | 8 | 7 | 15 | 11 | 15 | 13 | 7 | 9 | 9 | 6 |

## *3.6. Boolean functions*

The ten different boolean functions used are given in Table 5. In each single step operation there are two boolean functions f and g. In each round we have different f and g boolean functions as shown in Table 6, which helps to resist attacks. The SAC (Strict Avalanche Criterion) property of boolean functions also help to defy attacks.

Table 5. Boolean Functions

| |
|---|
| f1(x, y, z) = g4(x, y, z) = x ^ y ^ z |
| f2 (x, y, z) = g3(x, y, z) = (x ? y) V (¬x ? z) |
| f3(x, y, z) = g7(x, y, z) = (x ? ¬ y) ^ z |
| f4(x, y, z) = g1(x, y, z) = (x ? z) V (y ? ¬ z) |
| f5(x, y, z) = g9(x, y, z) = x ^ (y V¬ z) |
| f6(x, y, z) = g2(x, y, z) = (x V y) ? (¬x V z) |
| f7(x, y, z) = g6(x, y, z) = (x ?¬ y) ^ z |
| f8(x, y, z) = g5(x, y, z) = (x V z) ? (y V¬ z) |
| f9(x, y, z) = g10(x, y, z) = x ^ (y ? ¬ z) |
| f10(x, y, z) = g8(x, y, z) = x ^ (¬y ? z) |

Table 6. Boolean functons used in each round.

| Branch | Round1 | Round2 | Round3 | Round4 | Round5 |
|---|---|---|---|---|---|
| B1 | f1,g1 | f2,g2 | f3,g8 | f4,g4 | f5,g5 |
| B2 | f10,g10 | f9,g9 | f8,g3 | f7,g7 | f6,g6 |
| B3 | f6,g6 | f7,g7 | f8,g8 | f9,g9 | f10,g10 |
| B4 | f5,g5 | f4,g4 | f3,g3 | f2,g2 | f1,g1 |

### *3.7. Constants*

Here twenty different constants are used as shown in Table 7. These constants represent the first 32-bits of the fractional parts of the cube roots of the first twenty prime numbers. Different constants introduce asymmetry to each round. By using constants we can resist the attacker who tries to find a good differential characteristic with a high probability.

Table 7. Constants used in each round.

| Branch | Round1 | Round2 | Round3 | Round4 | Round5 |
|---|---|---|---|---|---|
| B1 | 428a2f98x | 71374491x | B5c0fbcfx | e9b5dba5$_x$ | 3956c25bx |
| B2 | 59f111f1x | 923f82a4x | Ab1c5ed5x | D807aa98x | 12835b01x |
| B3 | 243185bex | 550c7dc3x | 72be5d74x | 80deb1fex | 9bdc06a7x |
| B4 | c19bf174x | e49b69c1x | efbe4786x | 0fc19dc6x | 240ca1ccx |

## 4. Comparison of JERIM-320 with FORK-256 and RIPEMD-320

Table 8 and Table 9 list the differences and similarities of JERIM-320 compared to FORK-256 and RIPEMD-320 and clearly reveal the supremacy of JERIM-320 over these hash functions. Compared to FORK-256, JERIM-320 is having 320-bit hash length, eighty number of iterations on the message, twenty times processing of each message block, different message ordering in each round for all the four branches, introduction of cyclic shift, different order rule of message words for different branches, variable shift values and more number of Boolean functions and constants.

Table 8. Comparison of JERIM-320 with FORK-256

| **1. DIFFERENCES** | | | | |
|---|---|---|---|---|
| **Sl. No.** | **Properties** | **JERIM-320** | **FORK-256** | **Advantages of JERIM-320** |
| 1. | Message digest length | 320 bits | 256 bits | Makes brute force attack more difficult. |
| 2. | Given a message digest, number of operations required to find the original message. | $2^{320}$ | $2^{256}$ | Finding preimage or second preimage requires more operations. |
| 3. | Operations required to find two messages producing the same message digest | $2^{160}$ | $2^{128}$ | More resistance to birthday or square root attacks. |

| | | | | |
|---|---|---|---|---|
| 4. | Speed | 23.37 Mbps | 48.05 Mbps | The speed of JERIM-320 is also acceptable considering the higher degree of security. |
| 5. | Message block processing | 20 times | 4 times | Helps to resist attacks |
| 6. | Ordering of message words | Message ordering in each round for the four branches are considered differently by considering cyclic shift of i | Ordering rule of message words in different branches is same always. | More resistance to Wang et. al attacks |
| 7. | Shift values | Variable shift values in different rounds | Constant shift values | Provide better immunity against differential collision attack. |
| 8. | Boolean functions | Ten different Boolean functions are used. | Two Boolean functions | The SAC property of Boolean functions helps to resist against attacks. |
| 9. | Constants | Twenty different constants | Sixteen different constants | Increases asymmetry and hence more secure. |
| 10. | Message Block | JERIM-320 can use either the same or two different message sub blocks in a single step operation | FORK-256 uses two different message sub blocks in single step operation | Provides flexibility on security |
| 11. | Number of rounds and single step iterations in each branch. | Five rounds of 16 single step iterations making a total of 80 iterations | Single round of 8 single step iterations only. | Increases the complexity and makes attack more difficult |
| **2. SIMILARITIES** | | | | |
| Sl. No. | **Properties** | **JERIM-320** | **FORK-256** | **Advantages** |
| 1. | Parallel branch | It consists of four parallel branches | Same as JERIM-320 | More resistance to attacks |
| 2. | Input block and padding | Padding is used to make the length of the original message equal to a value which is 64 bits less than the exact multiple of 512 bit. | Same as JERIM-320 | Used in several hash functions |

Table 9. Comparison of JERIM-320 with RIPEMD-320

| **1. DIFFERENCES** | | | | |
|---|---|---|---|---|
| **Sl. No.** | **Properties** | **JERIM-320** | **RIPEMD-320** | **Advantages of JERIM-320** |
| 1. | Parallel branch | It consists of four parallel branches | It consists of two parallel branches | More resistance to attacks |
| 2. | Speed | 23.37 Mbps | 35.63 Mbps | The speed of JERIM-320 is also acceptable considering the higher degree of security. |
| 3. | Message block processing | 20 times | 10 times | Helps to resist attacks |
| 4. | Boolean functions | Ten different Boolean functions are used. | Five Boolean functions | The SAC property of Boolean functions helps to resist against attacks. |

| | | | | |
|---|---|---|---|---|
| 5. | Constants | Twenty different constants | Ten different constants | Increases asymmetry and hence more secure. |
| 6. | Message Block | JERIM-320 can use either the same or two different message sub blocks in a single step operation | RIPEMD-320 uses single message sub block in a single step operation | Provides flexibility on security |
| **2. SIMILARITIES** | | | | |
| Sl. No. | **Properties** | **JERIM-320** | **RIPEMD-320** | **Remarks** |
| 1. | Input block and padding | Padding is used to make the length of the original message equal to a value which is 64 bits less than the exact multiple of 512 bit. | Same as JERIM-320 | Used in several hash functions |
| 2. | Message digest length | 320 bits | Same as JERIM-320 | Makes brute force attack more difficult. |
| 3. | Given a message digest, number of operations required to find the original message. | $2^{320}$ | Same as JERIM-320 | Finding preimage or second preimage requires more operations. |
| 4. | Operations required to find two messages producing the same message digest | $2^{160}$ | Same as JERIM-320 | More resistance to birthday or square root attacks. |
| 5. | Shift values | Variable shift values in different rounds | Same as JERIM-320 | Provide better immunity against differential collision attack. |
| 6. | Number of rounds and single step iterations in each branch. | Five rounds of 16 single step iterations making a total of 80 iterations | Same as JERIM-320 | Increases the complexity and makes attack more difficult |

Compared to RIPEMD-320, JERIM-320 is having increased number of Boolean functions, constants and the more number of lines of message processing. The number of Boolean operations in RIPEMD-320 is five while that in JERIM-320 is ten. Similarly the number of constants in RIPEMD-320 is ten while for JERIM-320 it is twenty. Also each block in RIPEMD-320 is processed ten times while in JERIM-320, it is twenty times. These enhancements make JERIM-320 capable of resisting attacks to a much higher degree. It can be concluded that JERIM-320 is more secure than FORK-256 and RIPEMD-320.

## 5. Security Analysis

### *5.1. JERIM-320*

(1) The main difficulty in cryptanalyzing JERIM-320 comes from the fact that the same message blocks are given as input to each of the four streams in a permuted fashion. The attacker who tries to break JERIM-320 should aim simultaneously at four ways where the message difference passes, which would make the attacks more difficult.

(2) By using one word twice at each single step, it has been made difficult to construct a

differential characteristic with high probability.

(3) To avoid an attack that depends on brute-force methods, the output from the hash function has been made sufficiently long.

(4) While combining the outputs from the four branches, orthogonal operations (+ and ^) are used to create confusion and diffusion which adds to the security.

(5) There is a strong avalanche effect; hence a change in a single message bit affects all the registers after five rounds.

(6) All shortcut attacks on MD / Snefru target one of the intermediate blocks. Increasing the intermediate value to 320 bits helps to prevent these attacks.

(7) The single step operation ensures that changing a small number of bits in the message affects many bits during the various passes. Together with the strong avalanche, it helps JERIM-320 to resist attacks similar to Dobbertin's differential attack [Dobbertin, (1996)] on MD4.

(8) The feed forward of the initial state prevents meet-in-the-middle birthday attacks that finds preimages of the hash function.

### *5.2. Comparison with FORK-256 and RIPEMD-320*

(1) An independent analysis resulting in a 1-bit near collision attack against a reduced version of FORK-256 has been published [Matusiewicz *et al.*, (2007a)]. Then they have shown how to use this result to attack the complete FORK-256 hash function. There are eighty rounds of single step operations in JERIM-320 to make all such chances difficult.

(2) In FORK-256, the use of four streams with message reordering as a means to protect against differential analysis [Matusiewicz *et al.*, (2007a)] is ineffective since the same difference is applied to every message block and the same differential pattern is occurring simultaneously in the four streams. This is taken care in JERIM-320 by using a cyclic shift of i in the message ordering in each round so that different message orderings are used in different rounds. This causes the JERIM-320 algorithm to be much more resistant than FORK-256 against attacks on single branch.

(3) In FORK-256 the same compression functions f and g are used in all the four branches. Also some weakness in FORK-256 compression function has been published on two branches of the algorithm [Matusiewicz *et al.*, (2007b)]. This is overcome in JERIM-320 by using different compression functions in different branches and also in different rounds. Here if an attacker constructs an intended differential characteristic for one branch function, the different compression functions will cause unintended differential pattern in the other branch function, thus finding specific differences for patterns would not be straight forward.

(4) In FORK-256, the differences in the words of the internal state register do not diffuse identically. Thus, only the differences in the words A and E will spread to the other registers in the next round. As a result, a near collision occurs in FORK-256 [Matusiewicz *et al.*, (2007a)]. This factor is taken care in JERIM-320 by using the non linear functions f and g. Moreover these non linear functions are different in each branch and in each round. Also the shift values in each branch and for each iteration are different which

helps to change the internal values.

RIPEMD-320 provides longer hash strings and is a double width string variant of the popular RIPEMD-160. But both of these have two lines of message processing and each message block is processed ten times only. Hence RIPEMD-320 may not provide significant increase in security than RIPEMD-160 and is also susceptible to attacks in the long run.

## 6. Performance Evaluation

The total number of operations, memory requirements and the speed performance of JERIM-320 using one message block in single step operation, JERIM-320 using two message blocks in single step operation, FORK-256 and RIPEMD-320 were compared. The evaluation was done using Pentium IV processor, Linux operating system and C compiler.

As shown in Table 10 the total number of operations used in a single step operation of FORK-256 is 1.3 times more and RIPEMD-320 is 4.03 times less than that in JERIM-320.

Table 10. Comparison between JERIM-320, FORK-256 and RIPEMD-320.

| **Operation in single step.** | **JERIM-320 (using one message block in single step operation)** | **JERIM-320 (using two message blocks in single step operation)** | **FORK-256** | **RIPEMD-320** |
|---|---|---|---|---|
| Addition | 42 | 42 | 97 | 20 |
| Bitwise operation(^,V, ? ,¬) | 187 | 187 | 112 | 36 |
| Shift operation | 33 | 33 | 137 | 9 |
| Total number of operations | 262 | 262 | 346 | 65 |

As shown in Table 11, the memory requirement of JERIM-320 is less than that of FORK-256 and greater than that of RIPEMD-320. JERIM-320 uses 80 iterations for each message block, where as FORK-256 uses only 8 iterations. In each branch there are ten chaining variables in JERIM-320, but FORK-256 has only 8 variables. Moreover each message block is processed 20 times in JERIM-320 where as in FORK-256 it is only 4 times. Due to these, obviously the speed of operation will be slightly less as shown in Table 11 for JERIM-320 than FORK-256. The speed of JERIM-320-using one message block in single step operation is nearly 3.4 times less than that of FORK-256 and 2.5 times less than that of RIPEMD-320. The speed of JERIM-320-using two message blocks in single step operation is 2.05 times less than that of FORK-256 and 1.5 times less than that of RIPEMD-320 as shown in Table 11. Also compared to RIPEMD-320, JERIM-320 makes use of four parallel lines of message processing and hence the variables and computations required in JERIM-320 are more. But the multiple iterations and processing on the message blocks in JERIM-320 will result in much higher security. The speed of JERIM-320 is still very much acceptable.

Table 11. Performance comparison between JERIM-320, FORK-256 and RIPEMD-320

| **Algorithm** | **Speed (Mbps)** | **Memory requirement (Bytes)** |
|---|---|---|
| JERIM-320-using one message block in single step operation | 14.01 | 12003 |
| JERIM-320-using two different message blocks in single step operation | 23.37 | 12039 |
| FORK-256 | 48.05 | 12149 |
| RIPEMD-320 | 35.63 | 8927 |

## 7. Conclusion

A new hash function called JERIM-320 has been designed with improved security and reasonable speed. Its core strength is due to the factors like four parallel lines with 320-bit hash length, eighty number of iterations on the message block, twenty times processing of each message block, different message ordering in each round for all the four branches, introduction of cyclic shift in message ordering, different order rule of message words for different branches, variable shift values and more number of Boolean functions and constants. These enhancements make JERIM-320 capable of resisting attacks to a much higher degree compared hash functions with similar design.

## References

William Stallings (2003), *Cryptography and Network Security: Principles and Practices*, Prentice Hall, Dorling Kindersley, India.

Ivan Damgard, (1989) "A design principle for hash functions", Advances in Cryptology -CRYPTO, LNCS 435, Springer – Verlag, 416-427.

Ralph C.Merkle, (1989), "One way hash functions and DES", Advances in Cryptology –CRYPTO, LNCS 435, Springer – Verlag, 428-446.

R.L.Rivest, (1990), "The MD4 message digest algorithm", Advances in Cryptology-CRYPTO, LNCS 537, Springer-Verlag, 303-311.

R.L.Rivest, (1992), "The MD5 message digest algorithm", (RFC 1320), Internet Activities Board, Internet Privacy Task Force.

National Institute of Standards and Technology (NIST), (2002), FIPS-180-2: Secure Hash Standard, at http://csrc.nist.gov/publications/fips/fips 180-2/fips 180-2.pdf

E.Biham, R.Chen, A.Joux, P.Carribault, C.Lemuet, and W.Jalby, (2005) "Collissions of SHA-0 and Reduced SHA-1", *Advances in Cryptology-EUROCRYPT*, LNCS 3494, Springer- Verlag, 36-57.

F.Chabaud and A.Joux, (1998), "Differential Collissions in SHA-0", *Advances in Cryptology-CRYPTO*, LNCS 1462, Springer- Verlag, 56-71.

H. Dobbertin, (1996), "Cryptanalysis of MD4", Fast Software Encryption, LNCS 1039, Springer-Verlag, 53-69.

E.Biham and R.Chen, (2004), "Near Collissions of SHA-0", Advances in Cryptology- CRYPTO, LNCS 3152, Springer- Verlag, 290-305.

Xiaoyun Wang and Hongbo Yu, (2005), "How to break MD5 and other hash functions", Advances in Cryptology – EUROCRYPT, LNCS 3494, Springer-Verlag 19-35.

H. Dobbertin, A.Bosselaers,B.Preneel, (1996), "RIPEMD-160, a strengthened version of RIPEMD". Fast Software Encryption , LNCS 1039, Springer- Verlag, 71-82.

D.Hong, D.chang, J.Sung, S. Lee, S.Hong, J. Lee, D.Moon, S. Chee, (2006), "A New Dedicated 256-Bit Hash Function : FORK-256", Fast Software Encryption , LNCS 4047, Springer-Verlag, 195-209.

K. Matusiewicz, Thomas Peyrin, Olivier Billet, Scott Contini and Josef Pieprzyk, (2007a), "Cryptanalysis of FORK-256", Fast Software Encryption, LNCS 4593, Springer- Verlag, 19-38.

K. Matusiewicz, S.Contini, J.Pipeprzyk, (2007b), "Weaknesses of the FORK-256 compression function", http://eprint.iacr.org/2006/317.pdf

## 8. Test Vectors

### *8.1 JERIM-320 using one message block in Single step operation*

```
Message:
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
789abcdefghij
Intermediate Values:
Branch1
a2e47dc1 f7de8b66 92610e5a 642d9193 d1b17ef4 8025af21
6be9dfb 49de3ccc 1502d6ae cf596c85
Branch2
2e47d19f 11f7b254 8db858c6 b906f89a 7d24c5df 49d2dbab
9ea4903e 8e6805c0 206273be 9dc7eb34
Branch3
7f4d8321 894e27b7 fddb05d aa293970 74c83bda d387cb05
ce397853 1f7e3eff 45323469 ab4c0c62
Branch4
e4d22ef1 ba06df58 e0c12344 4852c253 e82bc3ba 3e1f9d87
d8cd742a f9d89672 dbbc3c25 6a654e78

Hash Code:
1518cd49 1d4d177f f33c1d89 5786e925 fa38cf04 f6c6413e
e5a1a61 78f83037 7072e78e be33818b

Message:
jbcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
789abcdefghij
Intermediate Values:
Branch1
2bb17fd8 88b2907b fb5597b9 e08e4cf7 d869113e ce86fffb
10f657de 39764f9c 1b563eb9 fa612803
Branch2
3ddd6242 27636afd 6b9313ed 1873d830 610d9b05 aaae8d4
93603614 64060d12 3e3b507c ea1999ee
Branch3
41c3a8e5 e32e6792 f43821e2 bff7dd18 6285f80 7f31331a
5a190dc 769f1f85 919a3d1f 8c9cc946
Branch4
9da92331 d9f50dad 4fd24ffc b7610f52 f4e9f3a0 f07089d5
9a4bf89a 84827901 a2ad3d87 2a8b786e
```

Hash Code:
44a71b45 d3ebb45e 62a2aa2f f9c2c6df ee1d95d6 5905ef0c de51f3ff 3a72b49c 4cb3a37 e4fec1f8

### *8.2 JERIM-320 using two different message blocks in Single step operation*

Message:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
789abcdefghij
Intermediate Values:
Branch1
4383170a ab2f4c9f 720700b2 58b7a70c f37cea35 fe9b3326
8b7871a7 99873e8 b6bf475a d50e448d
Branch2
76eef25 f6bf0918 fb240e3e 570221fd 2e19c9ef baaace8e
850ace28 25f46e36 2b54edcd 6bd8da7a
Branch3
2d9678c2 88c76853 f468278e af7fe30b 6a59130f d722a202
a6bbe9aa 9c08835e 779d9324 a0b53e0
Branch4
4d96f05a e837e968 93693d79 3a6773e 82b71d7b 7aab3f62
69c0c1f2 7906d4f5 997afe39 49834e4b

Hash Code:
94311971 3df02765 304f54e4 952ca42 f0f2a85 3a3ad78c
86ee3f9a 9c2e75f3 8c1d542b 1af336d6

Message:
jbcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
789abcdefghij
Intermediate Values:
Branch1
66baf333 e3f3b3cd 33525789 7d588262 e733f1c4 2cb06831
e2b7c51f e65c07e9 f252424e 2b5e0ece
Branch2
7fe61e0e 9b3911db 5982910 d64958eb e395685f 8e4cd3c9
1e434afe fb49c850 343f7682 e8ebd290
Branch3
18bf4291 69a1ca34 b27811aa 72b5def8 f40b6d7f 37c2fea8
962bba12 237d9f10 50c66664 a7219835
Branch4
7f02f6c4 58652c44 a43a159b 77a2e22b a7488c47 6266f5a7
2deeea13 206c2be5 39784dea 9a9d1860

Hash Code:
7f43b0b0 7ba8b8ce b32d1099 4d6b329 e3dd6362 2434e2d0
e9d7a660 afbf927a cab690b9 b2d5619f
```