ISSN: 2277-9655
Impact Factor: 4.116

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

DESIGN AND IMPLEMENTATION OF GEOMETRIC BASED CRYPTOGRAPHIC HASH ALGORITHM: ASH-256

Pallipamu.Venkateswara Rao *, K.Thammi Reddy, P.Suresh Varma
Department of Computer Science and Egineering, Adikavi Nannaya University, India
Department of Computer Science and Egineering, GITAM University, India
Department of Computer Science and Egineering, Adikavi Nannaya University, India

DOI: 10.5281/zenodo.56973

ABSTRACT

Online communication takes a major part in our daily life. Since sending or receiving information over internet is inevitable, usage of hash function is essential to check whether the information is correct or not especially for sensitive or confidential information. In this paper a new cryptographic hash function, Algorithm for Secure Hashing (ASH-256) has been proposed which is based on geometric concepts. In ASH-256, each 64-bit block of a given 512-bit block is increased to 96-bits by using Expansion table (E-Table) of DES(Data Encryption Standard) algorithm and divided into two equal sub-blocks. Each sub-block is used to generate three points of a triangle, which are involved in area calculation. The calculated area values are in turn processed to generate message digest. ASH-256 is more secure and exhibits strong avalanche effect and also simple construction and easy to implemention, when compared to standard hash function SHA2(256).

KEYWORDS: Cryptography, Hash function, Message digest, Authentication, Geometric concepts.

INTRODUCTION

Authentication is the most important concept in on line transactions. Cryptographic hash functions are the tools to generate message digest which can be used in variety of applications such as digital signatures, message integrity, password protection, random number generation and also some of the security protocols. Hash function takes arbitrary length input (or empty) and produces a fixed length output as hash code or message digest. For several years some of the popular hash functions played active role in various security applications but now a days they can be broken by fast computing systems with modern technology and thus loosing importance. Therefore designing of hash functions is a continuous process. In the year 2007, NIST (National Institute of Standards and Technology) announced a public request for candidate algorithm nominations to design new cryptographic hash algorithm (SHA-3) family and in this context *Keccak's algorithm* has been selected as SHA-3 family in 2012. This paper is organized into six sections. The section 2 comprises an overview of cryptographic hash functions. The section 3, propose a new secure hash function, Algorithm for Secure Hashing (ASH-256), which is designed based on geometric concepts. The section 4 deals with results and discussion. The implementation and analysis of ASH-256 is discussed in section 5. Finally the section 6 concludes with a note on future enhancement.

OVERVIEW OF CRYPTOGRAPHIC HASH FUNCTIONS

Hash functions are currently a hot topic of research in cryptography. The area of information security welcomes new approaches to the design of secure hash functions. There are innumerable hash functions [24] that have been developed for the past 20 years [4, 20]. Recently a new hash function has been standardized by NIST which is called as SHA-3(*Keccak* algorithm). Some of them are briefly discussed below [5, 6]:

a. The SHA family (SHA-3/ SHA-2/ SHA-1/ SHA-0)

In 2007, The NIST announced a publicly available contest, just like AES (Advanced Encryption Standard) contest, for selection of a new hash function [1, 2]. The institute received various proposals from individuals and different

research organizations. By conducting conferences (3 rounds) [3, 7] and inviting comments from outside and inside the institution on these submissions, finally in 2012, the NIST selected and announced *Keccak* as a new SHA-3 algorithm. The SHA-3 must provide message digests of 224, 256, 384 and 512 bits which can be allowed substitution for the SHA-2 family. Secure Hash Algorithms are the popular algorithms which were designed to overcome the flaws in earlier algorithms such as MD4 and MD5 etc., . These were published by NIST. Other than SHA-3 the SHA family [23] consists of SHA-0, SHA-1 and SHA-2.

SHA-2: The NSA [8] was designed a family of hash functions which are called as SHA2 (or SHA2 family) [16]. This version mainly consists of two sub algorithms SHA256 and SHA512. The variants of SHA256 are SHA224 and SHA512 is SHA384. The variants are also strong as SHA256 and SHA512. The main difference between the algorithms and variants is that truncation of their outputs and initial vector values. So far there are no successful attacks on this set of algorithms. But for future requirements, NIST conducted a completion and selected a new hash function *Keccak*, it is called as SHA-3.

SHA-1: This is also a 160-bit hash function which was designed by the United States National Security Agency and published by NIST as a FIPS [8]. It is a most popular hash function and used in number of security protocols and applications. In 2005, cryptanalysts found a weakness [9] in the algorithm, so that it is not safe to use in the secure communication [13], most of the cryptographers are moving to SHA2 [18, 19].

SHA-0: In 1993, A 160-bit hash functions under the name Secure Hash Algorithm-"SHA" (This algorithm is often called SHA-0 now) was published by NIST. But shortly it was withdrawn due to significant flaw [11], immediately refined this version as SHA-1.

b. The MDx Family

The series of MD (Message Digest) algorithms [25] were developed and used in various security applications. Here '*x*' stands for version number. Which were developed by R. Rivest of RSA DataSecurity Inc. Initial version of the MD is called as MD1. In 1990, MD2 [28] was proposed and recommended to replace BMAC [12]. Later MD3 was developed but not published. The MD4 [27] gives the revolution in the design of hash functions. This hash function has been influenced in the development of later designs. It was published in 1995 and used in for certain period later discarded by cryptographers due to flaw found in the algorithm. In 1991, MD5 [10, 26] was designed by R. Rivest which is an improved version of MD4. It also produced 128-bit message digests. This is a popular algorithm and widely used in variety of security communications. In 1996, a flaw was found in the algorithm [9, 14] which leads move to other algorithms like SHA1, SHA2 etc. Next version was also designed as MD6 and submitted in the NIST competition but was not selected.

c. The HAVAL Algorithm

In 1992, Zheng et.al. proposed a one way hashing algorithm called HAVAL. HAVAL takes arbitrary length input and produces a message digests of 128, 160, 192, 224 and 256 bits as per specifications. The processes of a message block in HAVAL [15] depends on a parameter. It allows a tradeoff between efficiency and security by means of the parameter and number of rounds which can be chosen equal to 3,4 and 5. HAVAL is faster than MD5, its fastness depends on the number of passes.

d. The RIPEMD Family

In 1992, the RIPEMD (Race Integrity Primitives Evaluation Message Digest) [21, 22] hash function was designed in the framework of the European RIPE project. Its structure is more or less similar to the MD4. The compression function consists essentially of two parallel lines (line1 operation and line2 operation) of 5 rounds of 16 steps. In 1996, a collision attack has been found by Dobbertin on versions of RIPEMD [17] reduced to two rounds out of three. The variants of RIPEMD are RIPEMD128, RIPEMD160, RIPEMD256 and RIPEMD512.

ALGORITHM FOR SECURE HASHING (ASH-256)

The name of this new algorithm is proposed as ASH-256 because 'ash' means the powdery residue of matter that remains after burning which is irreversible, similarly this algorithm ASH-256(Algorithm for Secure Hashing) is also the same property. It takes a message as input with a maximum length of less than 2^{64} bits and produces a 256-bits message digest as output. The given message is divided into 512 bit blocks and process each block with initial vector or intermediate vector (intermediate hash code). The overall process consists of the following steps:

ISSN: 2277-9655
Impact Factor: 4.116

a. Append padding bits

The message is padded so that its length is congruent to 448 modulo 512 (length=448 mod 512). Padding is always added, even if the message is already of the desired length. Thus the number of padding bits is in the range of 1 to 512.The padding consists of a single 1-bit followed by the necessary number of 0-bits.

b. Append length

A block of 64 bits which contains the length of the message (before padding) is appended to the message. This block is treated as an unsigned 64-bit integer (most significant byte first).

c. Initialize MD buffer

A 256-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 32-bit registers (R0, R1, R2, R3, R4, R5, R6, and R7). These registers are initialized to the following 32-bit integers (Hexadecimal values):

R0=6a 09 e6 67
R1=bb 67 ae 85
R2=3c 6e f3 72
R3=a5 4f f5 3a
R4=51 0e 52 7f
R5=9b 05 68 8c
R6=1f 83 d9 ab
R7=5b e0 cd 19

These values are same as the initial vector values of SHA2 (256) which are standardized by Federal Information Processing Standards Publications (FIPS PUBS) .These values are stored in big-endian format, which is the most significant byte of a word in the low-address byte position.

d. Processes message in 512- bit blocks

The heart of the algorithm is a compression function this module is labeled H_{ASH} in Fig. 1 and its logic is illustrated in Fig. 2.

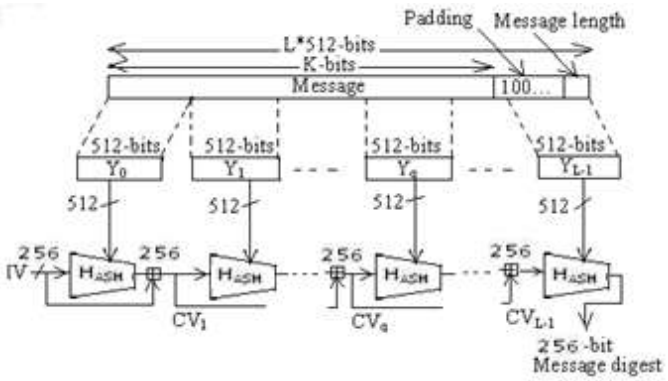


Fig. 1. Message digest generation using ASH-256

depicts the overall processing of a message to produce a message digest or hash code. The outcome of the first two steps (after append padding bits and append length) yields a message that is an integer multiple of 512-bits in length. The expanded message is represented as the sequence of 512-bit blocks Y_0, Y_1, Y_2… Y_{L-1}, so that the total length of the expanded message is L x 512 bits (L=the number of 512 bit blocks), that is the result is a multiple of sixteen 32-bit blocks. Here K represents the actual length of the message in bits; "IV" is the initial vector which is used to initialize the eight 32-bit registers (R0, R1, R2, R3, R4, R5, R6 and R7). CV_1, CV_2, CV_q.. and CV_{L-1} represents carry vector which holds intermediate and final result of the hash function, respectively. Each round it takes two inputs one 512-bit block (V_q) of the message and a 256-bit carry vector (CV_q). At the end of the L^{th} stage produces 256-bit message digest.

Initially given input message is divided into 512-bit blocks and each block can be passed into the hash function along with the 256-bit initial vector. The hash function is also called as Hash algorithm (H_{ASH}).

The Hash function (H_{ASH}) logic:

1. *Modification function-1* converts the given 512-bit block into modified 512-bit block.

2. *Expansion Function-1* converts the 256-bit initial vector into expanded vector (512-bits).
3. *XOR Operation* performs XOR operation on modified 512-bit block and expanded vector (512 bits).
4. *Expansion Function-2* expands 64-bits to 96-bits.
5. *Area calculation function* computes area of a triangle.
6. *Modification function-2* modifies the result of step 5.

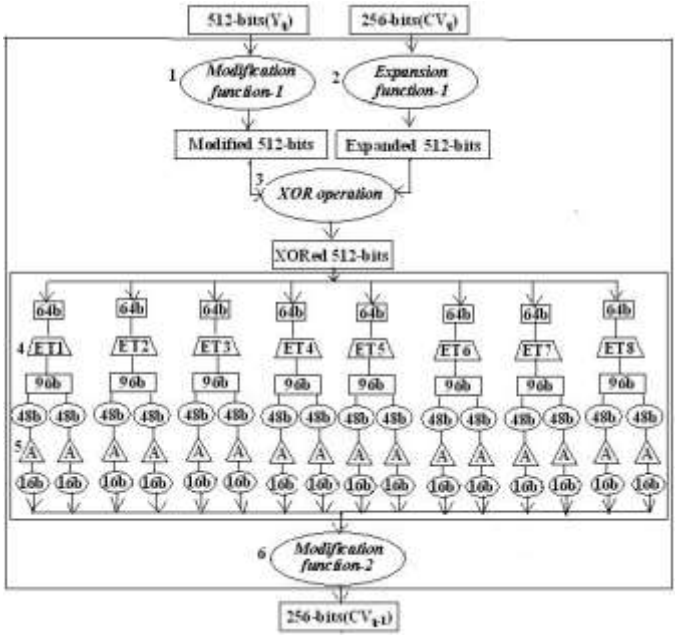


Fig. 2. The logic of hash function

Where,

Y_q =the q^{th} 512-bit block of the message

CV_q =chaining variable processed with the q^{th} block of the message

1=modification function discussed in Section 3

2=expansion function discussed in Section 3

3=16 bit XOR(Exclusive-OR) operation performed on every 16-bit block of the modified 512-bit block (Y_q) and

1. *Modification function-1*

Each 512-bit block of input is divided into 64 sub-blocks comprising 8-bits in each sub block. One temporary array of size 8 (Temp8 []) is taken and initialized with zeroes. The modification function consists of two sub functions as shown in Fig 3 and Fig 4.

Subfunction-1: XOR operation is applied to first 8-bit sub-block with Temp8 [] and then the resultant value is copied in to Temp8 []. Next the second 8-bit sub-block is XORed with Temp8 [] and the resultant value is copied again into Temp8 []. This process continues until the message of the 512-bit block exhausts.

Subfunction-2: The above result of Temp8 [] is incremented by 1 every time when performing XOR operation on each 8-bit units of 512-bit block, which generates a modified 512-bit block.

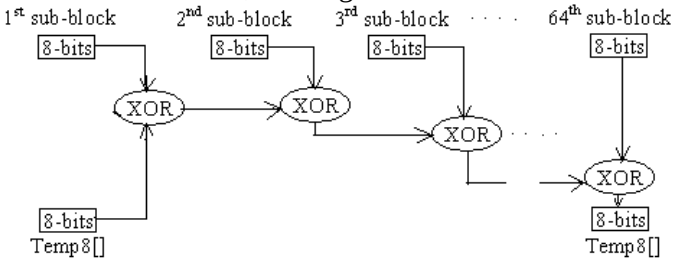


Fig. 3. Operation of Subfunction-1

ISSN: 2277-9655
Impact Factor: 4.116

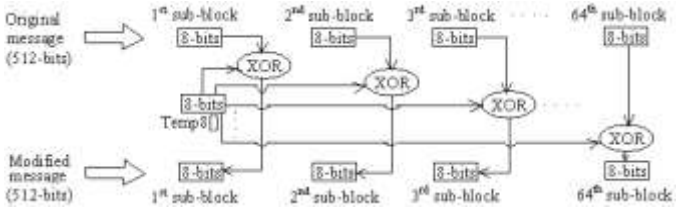


Fig. 4. Operation of Subfunction-2

2. *Expansion function-1*

The 256-bit Initial Vector (IV) is one of the input to the Hash function (H_{ASH}). To copy the contents of the registers from R0 to R7 circularly into an array of size 512 (EV512 []) until total number of bits are equal to 512. The result is called as Expanded Vector. The concatenation of the registers contents are as shown below:

R0||R1||R2||R3||R4||R5||R6||R7||R0||R1||R2||R3||R4|| R5||R6||R7 (|| = Concatenation operation).

3. *XOR operation*

Perform 8-bit XOR operation on first 8-bits of modified 512 bit block and first 8-bits of expanded vector, and the result is copied into first 8-bit positions of an array (M512[]) of size 512. Again perform XOR operation on second 8-bits of modified 512 bit block and second 8-bits of expanded vector, its result is also copied into next 8-bit positions of the array. This process is continued until the exhaustion of both the blocks.

4. *Expansion function-2*

The result of above step (*3. XOR operation*) is divided in to 8 equal sub blocks; each sub block contains 64 bits. Expansion table (E-table) of the DES (Data Encryption Standard) is applied to each sub block twise then produces 80 bits in first time and 96 bits in second time. The expansion process is shown from TABLE I.(A) to TABLE I.(C).

TABLE I. (A)
INDICATING POSITIONS OF 64 BITS

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

TABLE I. (B)
EXPANSION OF TABLE 1.(A)

ISSN: 2277-9655
Impact Factor: 4.116

63	0	1	2	3	4	5	6	7	8
7	8	9	10	11	12	13	14	15	16
15	16	17	18	19	20	21	22	23	24
23	24	25	26	27	28	29	30	31	32
31	32	33	34	35	36	37	38	39	40
39	40	41	42	43	44	45	46	47	48
47	48	49	50	51	52	53	54	55	56
55	56	57	58	59	60	61	62	63	0

TABLE I. (C)
EXPANSION OF TABLE 1.(B)

62	63	0	1	2	3	4	5	6	7	8	9
6	7	8	9	10	11	12	13	14	15	16	17
14	15	16	17	18	19	20	21	22	23	24	25
22	23	24	25	26	27	28	29	30	31	32	33
30	31	32	33	34	35	36	37	38	39	40	41
38	39	40	41	42	43	44	45	46	47	48	49
46	47	48	49	50	51	52	53	54	55	56	57
54	55	56	57	58	59	60	61	62	63	0	1

Rearrange the bits according to TABLE I (A) and TABLE I (B) bit positions. The above process is applied for remaining 7 sub-blocks.

5. *Area calculation function*

From step 4, each 96-bit block is divided into 2 sub-blocks of 48-bits in length. Again each sub-block is further divided into 3 sub-blocks of 16 bits. Now here the important point is that every 16-bit block act as a point in the geometric graph, i.e. first 8-bits act as X-axis value and second 8-bits act as Y-axis value. So, every 48-bit block is

represented as x1, y1, x2, y2, x3 and y3 which are represented three points of a triangle as shown in Fig 4. Convert these points into integers before calculating the area of a triangle.

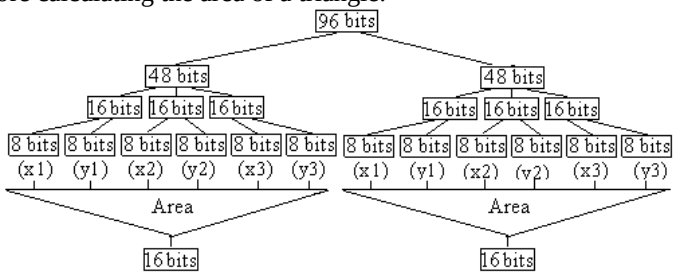


Fig. 4. Reduces from 96 bits to 16 bits and 16 bits (16-bits+16-bits=32-bits)

6. *Modification function-2*

The resultant area is converted into binary numbers, and zeroes are added on left hand side of the result if the number of bits is less than 16. This 16-bit block further divided into 4 sub-blocks of 4-bits in length. XOR operation is performed on each 4-bit sub-block with temporary array of size 4 (Temp4 []), which is initialized with zeroes. The same process is followed for 2^{nd} , 3^{rd} and 4^{th} sub-blocks. Again XOR operations are performed on Temp4[] with the precedent four 4-bit sub-blocks. The resultant 16-bits are converted into haxadecimal numbers which is the part of the message digest. The same procedure is continued for all resultant areas.

e. Output

The 256-bit message digest is obtained by concatenating the results of all triangle areas (in the form of hexadecimal numbers).

ASH-256 pseudo code:

Process the message in successive 512-bit blocks:
break message into 512-bit blocks
//…….
begin
for each block
  break block into sixty-four 8-bit big-endian blocks b[i], $0 \leq i \leq 64$
//…… (1. Modification function-1)……
  1. Perform XOR operation on all 8-bit blocks as follows:
    Subfunction-1:
    Temp8[]={0,0,0,0 ,0,0,0,0};k=0;
    for j from 0 to 63
        for i from 0 to 7
            Temp8[i]=Temp8[i] XOR b[k+i]
        end loop
        k=k+8
    end loop

    Subfunction-2:
    k=0;
    for j from 0 to 63
        for i from 0 to 7
            modifiedmsg[k+i]=Temp8[i] XOR b[k+i]
            //each time Temp8[i] value is incremented by 1
        end loop
        k=k+8
    end loop
//…….. (2. Expansion function-1)……..
    2. Extend the 256-bit MD buffer into 512-bit block:
EV512[]=(R0||R1||R2||R3||R4||R5||R6||R7||R0||R1||R2||R3||R4|| R5||R6||R7) (|| = Concatenation operation).

ISSN: 2277-9655
Impact Factor: 4.116

//…… (3. XOR operation)……

3. Declare an integer array of size 512(M512 []), k=0;
for j from 0 to 511
for i from 0 to 7
M512 [k] =modifiedmsg512 [j+i] XOR EV512 [j+i]
K++;
End loop
j=j+8
end loop

//…….. (4. Expansion function-2) …….

4. Divide the above (step 3) result into 8 equal sub blocks
Apply DES Expansion box technique to each sub block twice
1st Sub-block: (64-bits-> [E-Box] -> 80-bits)
(80-bits-> [E-Box] -> 96-bits)
The above operation is applied for remaining sub blocks (from 2nd sub-block to 8th sub-block).

//…….(5. Area calculation function) …….

5. Each 96-bit block is divided into two sub-blocks
96-bits -> 48-bits + 48-bits
Each 48-bit block is further divided into 3 sub-blocks
48-bits-> 16-bits + 16-bits + 16-bits
These three sub-blocks act as three points of a triangle, using these points calculate the area of a triangle.

Following equations are used in area calculation:

$a = \sqrt{(x2-x1)^2+(y2-y1)^2}$
$b = \sqrt{(x3-x2)^2+(y3-y2)^2}$
$c = \sqrt{(x1-x3)^2+(y1-y3)^2}$
$s = (a+b+c)/2$

$area = \sqrt{s*(s-a)*(s-b)*(s-c)}$

Where
The parameters a, b, and c are sides and s is semi-perimeter of a triangle (sqrt = square root).
The above process (area calculation) is applied for all 48-bit blocks. The resultant area of each 48-bit block is converted into bits, and zeroes are added on left hand side of the result, if the number of bits is less than 16.

//…….(6. Modification function-2) …….

6. Perform following operations for each 16-bit block

Which are obtained from step 5.

The first 16-bits are further divided into 4 equal sub-blocks, which are stored in the following arrays (size of each array is 4)

1. subblock1[]
2. subblock2[]
3. subblock3[]
4. subblock4[]

Temp4[]={0,0,0,0};
Temp4[] is a temporary array of size 4 and initialized with zeroes and do the following operations:

for i from 0 to 3
Temp4[i] =Temp4[i] XOR subblock1[i]
end loop
for i from 0 to 3
Temp4[i] =Temp4[i] XOR subblock2[i]
end loop
for i from 0 to 3
Temp4[i] =Temp4[i] XOR subblock3[i]
end loop

[Rao* *et al.*, 5(7): July, 2016] ISSN: 2277-9655
IC™ Value: 3.00 Impact Factor: 4.116

```
     for i from 0 to 3
Temp4[i] =Temp4[i] XOR subblock4[i]
     end loop
```

Perform XOR operation on Temp4[] with the precedent four 4-bit subblocks of the 16-bit block:
Declare an array of size 16, i.e. array16[];

```
     for i from 0 to 3  and j from 0 to 3
          array16[j] =Temp4[i] XOR subblock1[i]
     end loop
     for i from 0 to 3  and j from 4 to 7
array16[j] =Temp4[i] XOR subblock2[i]
     end loop
     for i from 0 to 3  and j from 8 to 11
          array16[j] =Temp4[i] XOR subblock3[i]
     end loop
     for i from 0 to 3  and j from 12 to 15
array16[j] =Temp4[i] XOR subblock4[i]
     end loop
```

The content of array16 [] is converted into hexadecimal numbers. This process is repeated for remaining 16-bit blocks.

End

All hexadecimal numbers are copied into carry vector (or registers) inorder, which is the message digest of a given message if it is single 512-bit block otherwise perform addition modulo-256 with previous carry vector and the result is used as input to the hash function along with next 512-bit block. The above process is repeated until the last 512-bit block.

Example:

Input string = "The quick brown fox jumps over the lazy dog."
Number of characters = 36(including point (.))
Total bits =288

a) Before padding:

0 0 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 0 1
1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 0 0 0 1
1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 0
1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0
1 0 0 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0

b) After padding:

0 0 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 0 1
1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 0 0 0 1
1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 0
1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0
1 0 0 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

c) After append length:

0 0 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 0 1
1 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 0 0 0 1
1 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 0
1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0
1 0 0 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0

1) Modification function-1:

Temp8[]={0,0,0,0,0,0,0,0}; // let us take temporary array and initialize with zeroes

Perform XOR operation on each 8-bit block of above step with Temp8 [];

After completion of XOR operation Temp8[] contains: Temp8 []={1,1,1,0, 0,1,0,0};

$(11100100)_2=(228)_{10}$ // temp8[] bits converted into decimal

Again Perform XOR operation on each 8-bit block of above result with Temp8[]={1,1,1,0, 0,1,0,0}=$(228)_{10}$ each time the Temp8[] value is incremented by 1 ,that is 228, 229 …etc.

{0 1 0 1 0 1 0 0} ^ {1 1 1 0 0 1 0 0}={1 0 1 1 0 0 0 0}
1st 8-bits $(228)_{10}$

{0 1 1 0 1 0 0 0}^{1 1 1 0 0 1 0 1}={1 0 0 0 1 1 0 1}
2nd 8-bits $(229)_{10}$

{0 1 1 0 0 1 0 1}^{1 1 1 0 0 1 1 0}={1 0 0 0 0 0 1 1}
3rd 8-bits $(230)_{10}$

...

Modified512[]={1 0 1 1 0 0 0 0 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1 1… 0 0 1 0 0 1 1 1};

2) Expansion function-1:

Concatenating initial vector register values in the following fashion (concatenate itself-R0||R1||R2||R3||R4||R5||R6||R7|| R0||R1||R2||R3||R4||R5||R6||R7):

0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 0 1 0 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 0 0 1 0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 0 1 0 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 0 0 1

The resultant vector is called Expanded Vector whch is explained in Section 2.

3*) XOR operation:*

Perform 16-bit XOR operation on every 16-bits of modified message with corresponding 16-bits of expanded vector.

(Modified message values) (Expanded vector values)
1st 16-bits 1st 16-bits
{1011000010001101}^{0110101000001001} = {1101101010000100}
2nd 16-bits 2nd 16-bits
{1000001110010110}^{1110011001100111} = {0110010111110001}
…
32nd 16-bits 32nd 16-bits
{1010001000100111}^{1100110100011001} = {0110111100111110}

4) Expansion function-2:

The result of above step (*3. XOR operation*) is divided in to 8 equal sub blocks; each sub-block contains 64 bits. The Expansion table (E-table) of DES (Data Encryption Standard) is applied to each sub-block twise then produces 80-bits in first time and 96-bits in second time. The same process is applied to remaining sub-blocks, which is shown in TABLE II.

For example:

The 1st 64-bits is expanded to 96-bits as shown below:

TABLE II
Expansion of 64-bits to 96-bits

64-bit block	E-Table is applied on 64-bit block produces 80-bits	E-Table is applied on 80-bit block produces 96-bits

ISSN: 2277-9655
Impact Factor: 4.116

01011011	0010110110	000101101100
00100001	1001000011	110010000110
10100110	1101001101	011010011010
10001111	0100011110	101000111101
01100100	1011001001	110110010011
11100111	0111001111	001110011111
11100100	1111001001	111110010010
10100000	0101000000	001010000001

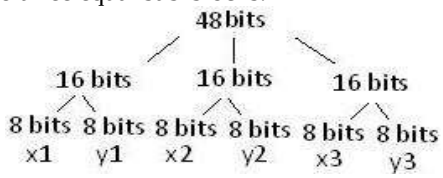
The same process is applied to the remaining 64-bit blocks.

5) Area calculation function:

From step 4, every 96-bit block is divided into two sub-blocks of 48-bits in length. Calculate area of a triangle for each 48-bit block as follows:

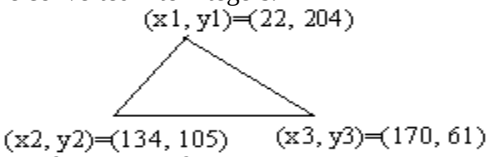
Area calculation:

The 48-bit block is further divided into three equal sub-blocks.



(x1, y1), (x2, y2) and (x3, y3) bits are converted into integers.

(x1, y1), (x2, y2) and (x3, y3) bits are converted into integers.



$a=\sqrt{(x2-x1)^2+(y2-y1)^2}=\sqrt{(134-22)^2+(105-204)^2}=149.48$

$b=\sqrt{(x3-x2)^2+(y3-y2)^2}=\sqrt{(170-134)^2+(61-105)^2}=56.85$

$c=\sqrt{(x1-x3)^2+(y1-y3)^2}=\sqrt{(22-170)^2+(204-61)^2}=205.79$

$s= (a+b+c)/2=206.06$

Area (A1)=sqrt(s(s-a)*(s-b)*(s-c)) =682

$= (0000\ 0010\ 1010\ 1010)_2$

Similarly find out the remaining areas.

6) Modification function-2:

Modify the above result (Area (A1)) as follows:

Temp4[]={0,0,0,0};

Perform XOR operation on each 4 bit block of Area (A1) and Temp4 [];

{0000}^{1010}={1010}
{1010}^{1010}={0000}
{0000}^{0010}={0010}
{0010}^{0000}={0010}

Temp4[]={0,0,1,0}; // modified Temp4[];

Perform XOR operation on each 4 bit block of Area (A1) and modified Temp4[]:

{0010}^{1010}={1000}
{0010}^{1010}={1000}
{0010}^{0010}={0000}
{0010}^{0000}={0010}

Modified Area ($A^1$1)={ 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0}

Similarly find out the remaining modified Areas ($A^1$1,$A^2$2,…).

Modified Area($A^1$1)={1000 1000 0000 0010}={8802}

Modified Area($A^2$2)={0000 0001 0010 1100}={012c}

Modified Area($A^3$3)={0111 0100 0011 1101}={743d}

Modified Area($A^4$4)={1100 0010 1110 0001}={c2e1}

ISSN: 2277-9655
Impact Factor: 4.116

Modified Area($A^5$5)={1001 0100 1101 1010}={94da}
Modified Area($A^6$6)={1111 0100 1011 1000}={f4b8}
Modified Area($A^7$7)={1101 0101 1000 1110}={d58e}
Modified Area($A^8$8)={1100 0100 1000 0111}={c487}
Modified Area($A^9$9)={1110 0111 1000 1001}={e789}
Modified Area(A^{10}10)={0010 0000 0011 0111}={2037}
Modified Area(A^{11}10)={0010 0100 0100 1001}={2449}
Modified Area(A^{12}10)={0110 1011 1101 0000}={6bd0}
Modified Area(A^{13}10)={0111 1100 1011 0111}={7cb7}
Modified Area(A^{14}10)={1100 0100 1000 0010}={c482}
Modified Area(A^{15}10)={0101 1001 1101 0111}={59d7}
Modified Area(A^{16}10)={0000 0010 0011 0110}={0236}
Message digest of a given message is concatenation of all the results of Modified Areas i.e.
"8802 012c 743d c2e1 94da f4b8 d58e c487 e789 2037 2449 6bd0 7cb7 c482 59d7 0236"
The avove hexadecimal values are the message digest of a given message if it is the last or single 512-bit block else copy these values into initial vector(or registers) and repeat the same process until last 512 bit block.

RESULTS AND DISCUSSION

These two algorithms have been implemented in Java and run on Intel® Dual CPU 1.86GHz, 1GB RAM and Windows-XP.
Outputs of ASH-256 for three sample inputs:
Input String : "The quick brown fox jumps over the lazy dog."
Output (Hash Code) : " 8802 012c 743d c2e1 94da f4b8 d58e c487 e789 2037 2449 6bd0 7cb7 c482 59d7 0236 "

Even a small change in the message will, with overwhelming probability, result in a completely different hash due to the *avalanche effect*. For example, changing *d*og to *c*og produces a hash with different values for 126 of the 256 bits and changing *d*og to *e*og produces a hash with different values for 126 of the 256 bits (Fig. 8).
Input String : "The quick brown fox jumps over the lazy cog."
Output (Hash Code): "84cf 499f 4fa1 2035 2b84 e68e 303b a06c 1cc0 312f db51 b29e 4d97 85d1 002d 1aa0"
Input String : "The quick brown fox jumps over the lazy eog."
Output (Hash Code): "2001 bd07 230a d1c9 175d 7ae0 58d4 4710 fa53 e79e 0034 1238 1aad 6bd3 0cc6 6bd8"
Outputs of SHA2 (256) for three sample inputs:
Input String : "The quick brown fox jumps over the lazy dog."
Output (Hash Code): "ef53 7f25 c895 bfa7 8252 6529 a9b6 3d97 aa63 1564 d5d7 89c2 b765 448c 8635 fb6c"
Input String : "The quick brown fox jumps over the lazy cog."
Output (Hash Code): "35e4 3eb1 9676 8d53 338a 7ea9 24c7 1e50 36c3 5763 f3f1 1347 63e3 4dd5 d019 76e9"
Input String : "The quick brown fox jumps over the lazy eog."
Output (Hash Code): "bd0c f2c3 ee82 be88 9a1b 5b75 4d2e 4a2a e07d 22da 40c3 9380 da06 1092 ca16 38d7"
In the above example, changing *d*og to *c*og produces a hash with different values for 117 of the 256 bits and changing *d*og to *e*og produces a hash with different values for 125 of the 256 bits (Fig. 8).
These two algorithms have been implemented in Java and run on Intel® Dual CPU 1.86GHz, 1GB RAM and Windows-XP.
Outputs of ASH-256 for three sample inputs:
Input String : "The quick brown fox jumps over the lazy dog."
Output (Hash Code) : " 8802 012c 743d c2e1 94da f4b8 d58e c487 e789 2037 2449 6bd0 7cb7 c482 59d7 0236 "
Even a small change in the message will, with overwhelming probability, result in a completely different hash due to the *avalanche effect*. For example, changing *d*og to *c*og produces a hash with different values for 126 of the 256 bits and changing *d*og to *e*og produces a hash with different values for 126 of the 256 bits (Fig. 8).
Input String : "The quick brown fox jumps over the lazy cog."
Output (Hash Code): "84cf 499f 4fa1 2035 2b84 e68e 303b a06c 1cc0 312f db51 b29e 4d97 85d1 002d 1aa0"
Input String : "The quick brown fox jumps over the lazy eog."
Output (Hash Code): "2001 bd07 230a d1c9 175d 7ae0 58d4 4710 fa53 e79e 0034 1238 1aad 6bd3 0cc6 6bd8"

ISSN: 2277-9655
Impact Factor: 4.116

Outputs of SHA2 (256) for three sample inputs:
Input String : "The quick brown fox jumps over the lazy dog."
Output (Hash Code): "ef53 7f25 c895 bfa7 8252 6529 a9b6 3d97 aa63 1564 d5d7 89c2 b765 448c 8635 fb6c"
Input String : "The quick brown fox jumps over the lazy cog."
Output (Hash Code): "35e4 3eb1 9676 8d53 338a 7ea9 24c7 1e50 36c3 5763 f3f1 1347 63e3 4dd5 d019 76e9"
Input String : "The quick brown fox jumps over the lazy eog."
Output (Hash Code): "bd0c f2c3 ee82 be88 9a1b 5b75 4d2e 4a2a e07d 22da 40c3 9380 da06 1092 ca16 38d7"
In the above example, changing *d*og to *c*og produces a hash with different values for 117 of the 256 bits and changing *d*og to *e*og produces a hash with different values for 125 of the 256 bits (Fig. 8).

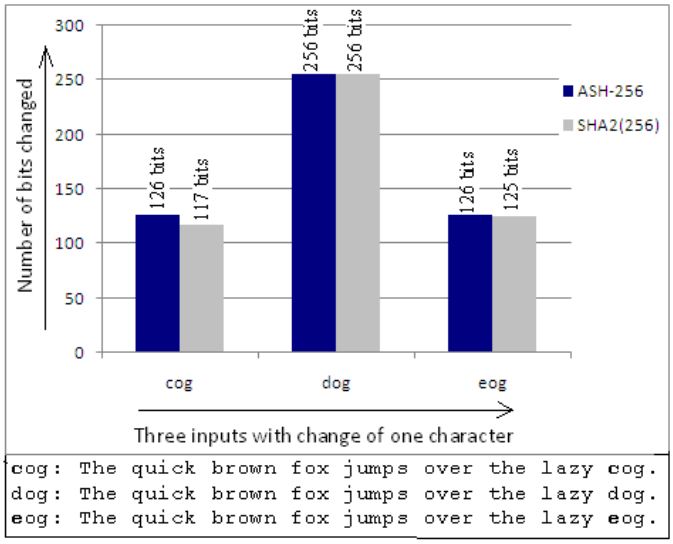


Fig. 8. Comparison of two algorithms with three inputs (dog is replaced by cog and eog)

Sample messages are given in TABLE III. Messages are given in ASCII format, while the corresponding hash results are in hexadecimal format.

TABLE III
COMPARISON OF ASH-256 AND SHA2 (256)

Input	Hash code of ASH-256	Hash code of SHA2(256)
" " (empty string)	289caa11880507deb7c61 ef34bfe5bf3ba226ad5055 1d2ffe1fd1679a0b83ec3	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
"a"	ed3ac8083dfecb651ce36aec84c478f53fd6455c364fc8573dd60df83216981c	ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
"abc"	af6192fede3cd581066b1cf2ce22471d367cd5b80bb36e84c0c879c0099db0b2	ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
"0123456789"	6be2b83ad4bbd85d4731f799b90db6cd3aa979ed94c5d85d5273ad55bc7e0442	84d89877f0d4041efb6bf91a16f0248f2fd573e6af05c19f96bedb9f882f7882
"message digest"	e9526cadc2e76607d4aba1b6527d58da93b0d800573a79f4ed1ec48228a	f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4efa3a1413d39

ISSN: 2277-9655
Impact Factor: 4.116

	30661	3cb650
"a…z"	3a98c7f896dbf0d09f48dc1bd7b9441fa4c90108770d79e4de1c6600f5ae6345	71c480df93d6ae2f1efad1447c66c9525e316218cf51fc8d9ed832f2daf18b73
"A…Z a…z0…9"	c0d8f2d95f85d2f9fe048f65772af3cf7dacfb52221af9758d54076ede230664	db4bfcbd4da0cd85a60c3c37d3fbd8805c77f15fc6b1fdfe614ee0a7c8fdb4c0
8 times "1234567890"	f3c828b5ff17f868e0c0077a9f6ccb70c6a5995c2209f1e648c33efba5cdec2a	f371bc4a311f2b009eef952dd83ca80e2b60026c8e935592d0f9c308453c813e

ANALYSIS OF ASH-256

a. Security Analysis

The hash function ASH-256 is designed to achieve mainly two goals, one is the change of single bit which change lot of bits in hash code that means it can show strong avalanche effect and the secondly it must be irreversible (i.e. one way property of hash function). The change of a bit will reflect the change of the hash code by performing XOR operations as mentioned in *Modification function-1*. However, XOR is linear, and doesn't prevent differential attacks. But change of a bit reflects lot of change of the hash code. Security of hash value can be achieved by *Expansion function and Area Calculation function* using geometric concepts as mentioned in *Section 5*. The One way property is explained below:

The Output of *Section 4*, each block is 96 bits in length wich are further divided into two equal sub-blocks, each sub-block contains 48-bits. Let us consider the first sub block
(000101101100110010000110011010011010101000111101
), this first sub block is further divided into three sub blocks of 16-bits long. The triangular representation of these three subblocks as three points are shown in Fig. 9 which is converted into integers before calculating the area.

Ist Sub block is represented in three points:
1st point(x1,y1) = ((00010110), (11001100))
2nd point(x2,y2) = ((10000110), (01101001))
3rd point(x3,y3) = ((10101010), (00111101))

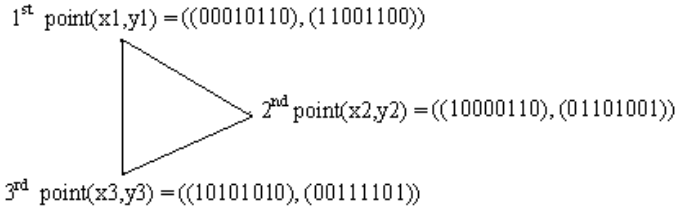


Fig. 9. Representation of triangle with three points

Area Calculation:
Area of above triangle=1000 1000 0000 0010= 8802

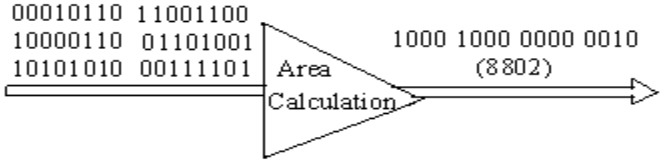


Fig. 10. Illustration of One way property (from 48- bits to 16- bits)

In this way we can find out areas of remaining 48-bit blocks and concatenate the results, then convert into hexadecimal form. So, it shows the one way property of ASH-256 hash function (Fig. 10).

Attacks independent of the algorithm are:

Random attack- The probability of breaking of this algorithm is $1/2^{256}$, the number of trails and the expected values are the key parameters of this attack.

Birthday attack- The length of the Message digest is 256 bits then there are 2^{256} possibilities. The Cryptanalyst generate two samples which are S_1 and S_2 from digest. The approximate probability of two samples is as follows:

$$P = 1 - \frac{1}{e^{\frac{S_1 S_2}{2^{256}}}}$$

Attacks dependent on the algorithm are:

Meet-in-the-middle attack- The probability of generates middle value using first part samples(q_1) and last part samples (q_2) is as follows:

$$P = 1-(1\div(e^{(q1q2/2^n)}))$$

Constrained Meet-in-the-middle attack- This is also same as above but it is using certain constraints.

Generalized Meet-in-the-middle attack- In this attack given message is divided in to 2.10^{p-1} blocks(p-fold iterated scheme).$10^p.2^{256/2}$ operations required to break the scheme.

Correcting block attack-The attacker take a message and it's message digest and he is trying to change blocks number of times then observe the digest remains same.

Differential attack-The priciple behind this attack is that if there is a relation exists between input and output differences,collission occures when the difference is zero.

b. Performance Analysis

These two algorithms ASH-256 and SHA2(256) were tested for comparision based on the execution time and memory space requirements. All algorithms have been implemented in Java and run on Intel®Dual CPU 1.86GHz, 1GB RAM and Windows-XP. By the results of the experiment, it was found that ASH-256 take less execution time for the files of smaller size but more execution time for the files of larger size as compared to SHA2(256) to generate hash code. The results in Fig 12 show time wise comparison of the two algorithms with the input sizes as 1KB, 5KB, 15KB and 20KB, these results are average values which are taken after running each file in five times.

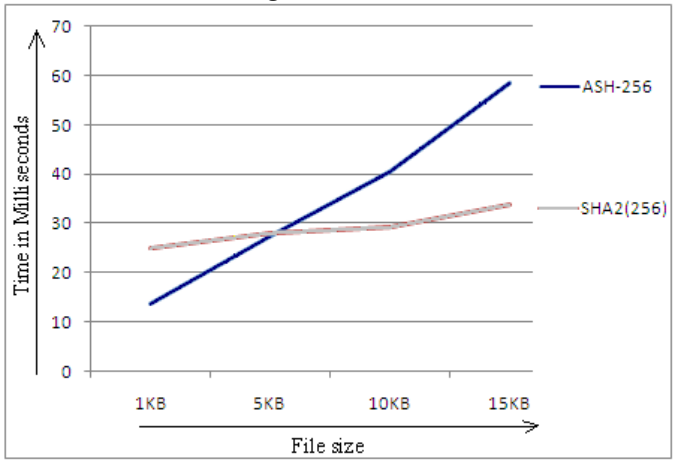


Fig. 12. Relative comparison of ASH-256 and SHA2 (256) with respect to execution time.

The complexity analysis on the operations used in ASH-256 versus those used in SHA2 (256). The total number of operations is compared in TABLE IV.

TABLE IV

Comparision between number of operations of ASH-256 and SHA2(256)

Operation	ASH-256	SHA2(256)
Bitwise operations	704	1024
Addition	32	600
Subtraction	144	-
Multiplication	192	-
Division	16	-

Squar-root	64	-
Shift	-	96
Shift Rotation	-	576
Total	1152	2296

Analysis of the TABLE IV:

The ASH-256 consists of six main functions; the first function is Modification function-1 which contains two stages. In first stage, XOR operations are performed on the temporary array (TEMP8[]) and every 8-bits of 512-bit block, (8x8=64). In the second stage the result of temporary array is XORed with every 8-bits of 512-bit block (8x8=64). So, the total XOR operations in the first function are 128. In step 2 contains only concatenation operations. In step 3 contains (8x8) 64 XOR operations. Rearraging of bits are done in step 4. In step 5 contains (2x16) 32 additions, (9x16) 144 subtractions, (12x16) 192 multiplications, 16 divisions and (4x16) 64 sqaure-root operations. Last step contains (16x16 and 16x16) 512 XOR operations. These results show how much less operations in ASH-256 than SHA2 (256).

CONCLUSION

The demand of online transactions is increasing from day to day in our daily life corresponding security requirements are also increases in secure communications. Based on present scenario we observed that hash functions are important cryptographic primitives in information security applications. This paper proposes a novel secure hash function called *Algorithm for Secure Hashing* (ASH-256) which is designed using geometric concepts. The core strengths of ASH-256 are *XOR operation, E-Table of DES* and *area calculation function*, which result in strong nonlinear avalanche effect, increased diffusion in output and make differential attacks difficult. Based on the experimental results and analysis we conclude that the proposed algorithm is more secure than SHA2 (256) because it exhibits strong avalanche effect but in performance point of view it is suitable for small and modatare length messages. Thus it is more secure and simpler than most of the existing popular hash functions, which may make ASH-256 a substitute. This algorithm is useful for developing secure applications like digital signature and MAC. Further this geometric concept appears to be useful for developing various versions of hash functions like ASH-224, ASH-384 and ASH-512

REFERENCES

[1] Federal Information Processing Standards Publication, Secure Hash Standard (SHS), FIPS PUB 180-4, March 2012.

[2] Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition, November 2012.http://dx.doi.org/10.6028/NIST.IR.7896

[3] Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition, February 2011.
http://csrc.nist.gov/publications/nistir/ir7764/nistir-7764.pdf

[4] Harshvardhan Tiwari and Krishna asawa, **"**Cryptographic Hash Function: An Elevated View**",** *European Journal of Scientific Research*, ISSN 1450-216X Vol.43 No.4 (2010), pp.452-465 © EuroJournals Publishing, Inc.

[5] Sheena Mathew, K. Poulose Jacob, "Performance Evaluation of Popular Hash Functions", *World Academy of Science, Engineering and Technology 61*. 2010.

[6] NeetuSetti, "Cryptanalysis of modern Cryptography Algorithms". International Journal of Compuer Science and Technology, December 2010.

[7] Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition, September 2009.
http://csrc.nist.gov/publications/nistir/ir7620/nistir_7620.pdf

[8] Federal Information Processing Standards Publication," Secure Hash Standard (SHS"), *Information Technology Laboratory, NIST* , MD 20899-8900 October,2008.

[9] Lars R.Knudsenl, Christian Rechberger2 and Soren S. Thomsen, " The Grindahl Hash Function",*A. Biryukov(Ed,):FSE 2007,LNCS 4593,pp.39-57,2007.International Association for Cryptologic research*.

[10] Praveen Gauravaram and Adrian Mccullagh and Ed Dawson, "Attacks on MD5 and SHA-1: Is this the "Sword of Damocles" *for Electronic Commerce*, March 15, 2006.
[11] Atul Kahate, "Cryptography and Network Security" *Tata McGraw-Hill*, 2006.
[12] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribaut, Christophe Lemuet,William Jalby, "Collision in SHA-0 and Reduced SHA-1", *EUROCRYPT05.* 2005.
[13] Ilya Mironov,"Hash functions: Theory, attacks, and applications", November 14. 2005.
[14] Xiaoyunwang, Yiqun Lisa Yin, and Hongbo Yu,"Finding Collisions in the Full SHA-1",*Lecture Notes in Computer Science*, Volume 3621,CRYPTO 2005Proceedings, pp. 17–36. 2005.
[15] Xiaoyun Wang and Hongbo Yu, **"**How to Break MD5 and Other Hash Functions"**,** *R. Cramer (Ed.): EUROCRYPT 2005, LNCS 3494*, pp. 19–35, 2005.International Association for Cryptologic Research05.
[16] X. Wang, X. D. Feng, X. Lai and H.Yu, 2004. "*Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*", rump session, CRYPTO 04.
[17] Henri Gilbert and Helena Handschuh, "Security analysis of SHA-256 and Sisters.Lecture Notes in Computer Science., 3006:175-193, 2004.
[18] Xiaoyunwang1, Dengguo Feng2, Xuejia Lai3, HONGBO YU1, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD***,** revised on August 17, 2004.
[19] WILLIAM STALLINGS, *Cryptography and Network Security: Principles and Practice*. 3/e PH, 2003.
[20] NIST, "*Secure Hash Standars*", FIPS PUB 180-2, 2002.
[21] Praveen S.S. Gauravaram, William L. Millan, and Lauren J. May, *CRUSH: A New Cryptographic Hash Function using Iterated Halving Technique*, Information Security Research Centre, Queensland University of Technology,GPO BOX 2434, Brisbane, QLD, 4001, Australia.
[22] A. Bosselaers, H. Dobbertin, B. Preneel, "The Cryptopraghic Hash Function RIPEMD-160", CryptoBytes, RSA laboratories, 1997.
[23] H. Dobbertin, A. Bosselaers, B. Preneel, "*RIPMEMD-160: A Strengthened Version of RIPMMD*," Fast Software EncrZption, LNCS 1039, D.Gollmann, Ed., Springer-Verlag, 1996.
[24] NIST, "Secure Hash Standard", FIPS PUB, 1995.
[25] B.Preenel,"*Cryptographic hash functions"*, Transactions on Telecommunications, VOL5, 431-448, 1994.
[26] R.L.Rivest,"*The MD4 Message Digest Algorithm*", RFC 1320, 1992.
[27] R.L.Rivest,"*The MD5 Message Digest Algorithm*", RFC 1321, 1992.
[28] R.L.Rivest,"*The MD2 Message Digest Algorithm*", RFC 1319, 1992.