Abstract—Cryptographic hash function assumes a fundamental job in numerous pieces of cryptographic algorithm and conventions, particularly in authentication, non-repudiation and information trustworthiness administrations. A cryptographic hash work takes a commitment of optional tremendous size message and conveys a fixed little size hash code as a yield. In the proposed work SSLHA-160 (a strong and secure lightweight cryptographic hash algorithm), each 512-digit square of a message is first diminished to 256-bit. A cryptographic hash work takes a contribution of discretionary enormous size message and delivers a fixed little size hash code as a yield. In the proposed work SSLHA-160 (A strong and secure lightweight cryptographic hash algorithm), each 512-digit square of a message is first diminished to 256-bit and afterward partitioned into eight equivalent block of 32 pieces each and each 32-cycle block is additionally separated into two sub-block of 16-piece each. These two sub-blocks go about as two purposes of an elliptic curve, which are utilized for computing another point which is of 16 pieces. The new point esteems are thusly handled to produce message digest. SSLHA-160 is easy to develop, simple to actualize and displays solid torrential slide impact (avalanche), when contrasted with SHA1, RIPEMD160 and MD5.

Keywords—Cryptography hash function; message digests; authentication; elliptic curve concepts

I. INTRODUCTION

Authentication is a significant idea in information security, and one method of accomplishing this by using hash function. A Hash function is a mathematical function that maps a message of variable size into a small-length esteem called message digest. Message digest is likewise alluded to as a synopsis of the information or unique mark of the information. One direction hash function is a variation of the message authentication code. A single direction hash function, otherwise called a message digest, is a numerical function that takes a variable-length input string and converts it into a small-length arrangement that is computationally hard to alter—that is, going in back direction is impossible create the information from the hash. Hash code is created from all the pieces of the message any blunder in the message can without much of a stretch be distinguished from hash code as little change made in the message brings about huge modification in the hash code. Message confirmation and gadget verification can be accomplished through this hash code or message digest. Likewise message verification is said to secure the respectability of a message, for example the message that is gotten is showing up in a similar frame as send by the sender with no alteration done by clients with vindictive aim. With digital signature well source of messages can be verified. At the point when responsibility for computerized signature mystery key is bound to a particular client, a substantial digital signature gives an affirmation that the message was send by the right client. Sender validation gets significant in monetary and in light weight scenario. This paper is coordinated as follows. Area 2 gives a diagram of cryptographic hash capacities. In Area 3, proposes a strong and secure lightweight cryptographic hash algorithm SSLHA-160, planned dependent on elliptic curve ideas. Area 4 presents results and conversations. The investigation of SSLHA-160 is talked about in Area 5. Area 6 closes by indicating the straightforwardness of calculation.

II. OVERVIEW OF CRYPTOGRAPHIC HASH FUNCTION

Hash functions are as of now alluring subject of examination. Especially data security area consistently searches for new ways to deal with plan the safe hash capacities. There are endless hash works that have been created like (Venkateswara Rao Pallipamu, K Thammi Reddy, P Suresh Varma 2014 ASH-160 [1], Venkateswara Rao Pallipamu, K Thammi Reddy, P Suresh Varma 2014 : ASH-512 [2], and Venkateswara Rao Pallipamu, K Thammi Reddy, P Suresh Varma 2016 :ASH-256 [3] Some of the renowned hash work are examined beneath:

A. The MDx Family

The MD calculations are generally used to create an advanced mark from a message. MD2 [14] was created in 1989 for 8-digit encoders. It cushions the message to be encoded until it is a various of 16 bytes long, affixes a 16-byte checksum, and computes the hash. MD4 [12] was created in 1990 for 32-digit encoders. The MD5 [13] message-digest calculation which yield a MD of 128 bit, but it also has shortcoming [19] [20] [22]. Starting at 2019, MD5 keeps on being generally utilized, despite its all-around recorded shortcomings and expostulation by security specialists.
B. The RIPEMD Family

RIPEMD is a gathering of hash work which is created by Hans Dobbertin, Antoon Bosselaers and Bart Preneel in 1992. RIPEMD-160 [4] is a 160-bit cryptographic hash function. It was, intended to be used as a secure replacement for the 128-bit hash function MD4, MD5 and RIPEMD. RIPEMD was developed in the frame work of the EU project RIPE (RACE Integrity Primitives Evaluation, 1988-1992). RIPEMD-160 is a strengthened version of RIPEMD with a 160-bit hash result.

C. The HAVAL Algorithm

HAVAL is a cryptographic hash work. In contrast to MD5, yet like most current cryptographic hash capacities, HAVAL can create hashes of various lengths – 128 pieces, 160 pieces, 192 pieces, 224 pieces, and 256 pieces. HAVAL additionally permits clients to indicate the quantity of rounds (3, 4, or 5) to be utilized to create the hash. HAVAL was broken in 2004. HAVAL was imagined by Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry in 1992

D. The SHA Family

The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published in 1993. A revised version was issued as FIPS PUB 180-1 in 1995 [5] [6]. This is generally referred as SHA-1. SHA is based on the MD4 algorithm. SHA works by taking care of a message as a bit string of length under 2^{64} pieces, and creating a 160-piece hash esteem known as a message digest. Various version of SHA are SHA-0 (first form, but had loop hole [21], SHA-1 made by National Security Agency but had cryptographic flaw, SHA-2 and SHA-3 [16],[17],[18].

III. STRONG AND SECURE LIGHTWEIGHT CRYPTOGRAPHIC HASH ALGORITHM USING ELLIPTIC CURVE (SSLHA-160)

The proposed calculation is named as SSLHA-160. This calculation SSLHA-160 (A strong and secure lightweight cryptographic hash algorithm) accepts a message as contribution with a most extreme length of under 2^{64} pieces and creates a 160-piece message digest as yield [10], [11]. First the info message is divided into squares of 512 pieces. This 512 pieces is taken as info, at that point the information is diminished from 512-digit squares to 256-bit blocks. The Hash code creation function acknowledges two sources of info which are 512 pieces square of the message and the initialize MD buffer (fastening variable 160-bits). The cycle comprises of the accompanying advances:

A. Append Cushioning Pieces

The message is cushioned so its length is consistent to 448 modulo 512 (length = 448 mod 512). Cushioning is constantly done, regardless of whether the message is of wanted length. Hence, the quantity of cushioning pieces is in the scope of 1-448. The cushioning comprises of a solitary 1 followed by the important number of 0's.

B. Append Length

A square of 64 pieces which contains the length of the message (prior to cushioning) is affixed to the message. This square is treated as an unsigned 64-bit number (most huge byte first).

C. Initialize MD Cushion

A 160-piece cushion is utilized to hold transitional and end-product of the hash work. The support can be spoken to as five 32-digit registers (S0, S1, S2, S3 and S4) . These registers are instated to the accompanying 32-bit numbers (Hexadecimal qualities):

\[
S0 = 67 45 23 01
\]
\[
S1 = ef cd ab 89
\]
\[
S2 = 98 ba dc fe
\]
\[
S3 = 30 22 54 76
\]
\[
S4 = c3 d2 e1 f0
\]

These qualities are same as the underlying vector estimations of SHA-1 which are normalized by Federal Information Processing Standards Publications (FIPS PUBS). These qualities are put away in big endian design, which is the main byte of a word in the low-address byte position.

D. Processes Message in 512-Bit Blocks

The Hash code generation method comprises of five sub function. This part is named (HSSLHA-160) in Fig. 1 and its rationale is appeared in Fig. 2. Fig. 1 portrays the general preparing of message to create a message review [7]. The result of the initial two stages (after add cushioning pieces and attach size) outputs a message that is a number multiple of 512-piece long. The extended message is spoken to as the grouping of 512-cycle blocks X0, X1,X2 ...X_{l-1}, so the complete size of the extended message is \(L \times 512\) pieces (\(L = \) the quantity of 512 bit blocks), that is the consequence of different of sixteen 32-bit blocks. Here \(K\) speaks to the genuine length of the message in pieces; "IV" is the underlying vector which is utilized to introduce the five 32-cycle registers (S0, S1, S2, S3 and S4). VC1, VC2, ... VC_{q}... and VC_{L-1} speak to instate MD(carry vector) which holds middle of the road and eventual outcome of the Hash work, individually. Each round takes two information sources one 512-cycle block (Xq) of the message and a 160 piece convey vector (VCq). Toward the finish of the Lth stage produces 160 piece message digest. At first the given message is isolated into 512-digit blocks, and each square is passed to Hash Code creating function (HSSLHA) as a contribution alongside the 160-piece vector.

![Fig. 1. Hash Code Creation using SSLHA-160.](www.ijacsa.thesai.org)
The Hash work (H\textsubscript{ASH160}) rationale is:

Alteration function(A\textsubscript{fn}) changes over the given 512-bit block into altered 256-digit block.

Concatenation function(C\textsubscript{fn}) changes over the given 160-piece vector into CONCATENATED 256-bit vector.

32-digit XOR operation(XOR\textsubscript{op}) performs XOR procedure on each 32-pieces of changed 256-bit block and concatenated 256-bit vector.

Segregation and Modification operation(Sop) isolates the 256-bit block into 8 sub-block of 32-bits each and then each 32 bit block is separated into four 8-bit block, after that it modifies each eight bit block..

New Point(on elliptic curve) estimation work (NP\textsubscript{fn}) computes new point on elliptic curve utilizing 32-bit block.

where,

- \(X_q\) = the qth 512-cycle square of the message.
- \(VC_q\) = anchoring variable prepared with the qth square of the message.
- A\textsubscript{fn} = Alteration function.

- C\textsubscript{fn} = Concatenation function.
- XOR\textsubscript{op} = XOR(Exclusive-OR) activity performed on each 32-cycle square of the changed 256-digit block (\(X_q\)) and relating 32-pieces of the extended 256-bit block (\(VC_q\)).
- Sop = 256-bit square can be isolated into 8 sub squares of 32-bits each and then each 32 bit block is separated into four 8-bit block, after that it modifies each eight bit block..
- NP\textsubscript{fn} = first 32-bits further partitioned into 2 sub squares of 16-bits each.

First sub square = 16 pieces, partitioned into two sub squares (8-bits, 8- bits) = (xx1, yy1).

Second sub square = 16 pieces, partitioned into two sub squares (8-bits, 8-bits) = (xx2, yy2).

The qualities (xx1, yy1, xx2, and yy2 as appeared in Fig. 3) are changed over into whole numbers followed by computing a new point on elliptic curve. The above process is redone for remaining seven 32-bit block and the end result is 8 sub block each of 16 bits. Now each of this 8 sub block each of 16 bit is altered ,the first 8-bit of first block is XOR with a 8-bit sub block which are all zeros, the result is stored in first 8-bit sub block. The second 8-bit sub block of first block is added with first 8-bit sub block and result is stored in second 8-bit sub block and the process is repeated for all 8-bit sub block. The result is formatted 128 bit block. Now adjacent sub block each 16 bits are added which results in a sub block of 32 bit and we get four 32 bit sub block. By performing some mathematical operation on above four computed 32 bit sub block and then XOR them results in the fifth 32 bit sub block. This five 32 bit sub block when joined amounts to 160 bit and when added with initialized MD buffer forms the hash code. This 160 bit hash code will be the input to next 512 bit of message i.e. it will act as initialize MD buffer for next 512 bit of the input message. The last 160 bit code generated from last 512 bit of message will be the final hash code which will act as authentication code.
a) Alteration function ($A_a$): Each 512-bit block of information is separated into 64 sub-blocks involving 8-pieces of each sub-block. One brief exhibit of size 8 (Tempx8[ ]) is taken and instated with zeroes. The modification work comprises of two sub functions as demonstrated as follows:

Sub-function-1: Initially, the above aftereffect of Tempx8[ ] is XOR with the initial 8-pieces of the 512-bit block of message to create the initial 8-pieces of adjusted message. In resulting step this Tempx8[ ] is augmented by 1 and is XOR with the following 8-pieces of the message to create the following 8-pieces of altered message as spoken to in Fig. 4.

![Fig. 4. Operation of Sub-Function-1.](image)

Sub-function-2: The above result is again divided into sub block of 8 bits and 1st 8-bit sub-block and middle 8-bit sub block after bitwise complimenting(−) are XOR and the result is again bitwise complimented(−) and stored in a separate array(W2). The above process is repeated for second 8-bit sub block and middle plus one 8-bit sub block and result is stored in second part of separate array(W2). This is repeated for all remaining 8-bit sub block. The result is 512 bit block is reduced to 256 bit block. The modified message as depicted below:

```c
int ModInp(vector<int> m, int n)
{
    int i,j=0, p, q, k1, T=0;
    p=n;
    n*=64;
    for (i=n;i<n+64; i++)
    {
        W[i]=T ^ m[i] ;
        T++;
    }
    j=((i+n)/2);
    q=j;
    for (i=n;i<n+q+n; i++)
    {
        W2[k1] = (~(W[i]) ^ ~(W[j]));
        k1++;
        j++;
    }
    return 0;
}
```

d) Segregation and Modification Operation(Sop): Each 32-digit square of the above outcome is additionally partitioned into 2 sub-squares of 16-bits each. Then each 16-bit block is separated into two 8-bit sub block, each of these eight bits are modified that is First the four bits are taken from the 8-bit value starting from least significant position then it is XOR with a variable1(km) whose value is zero, the result is kept in a variable2(km1) also the value of variable1(km) is updated i.e. it is given the value of variable2(km1).

![Fig. 5. Concatenation of 160-Bit Block (IV) to 256-Bit Block (CV).](image)

32-bit XOR operation (XORop): XOR activity is performed on initial 32-bit sub-squares of altered 256-cycle block and extended 256-bit block. For this first four 8-bit block are expanded and added so that it becomes 32 bit block.

For the rest of the bits of modified 256-bit and expanded 256-bit block the result is 256-bit block as shown below:

```c
for ( t1 = 0; t1 < 8; t1++)
{
    W3[t1] = (W2[4 * t1] << 24) + (W2[4 * t1 + 1] << 16) + (W2[4 * t1 + 2] << 8) + (W2[4 * t1 + 3]) ;
    if(t1==0)
        W3[t1]=((W3[t1]) ^ (A));
    if(t1==1)
        W3[t1]=((W3[t1]) ^ (B));
    if(t1==2)
        W3[t1]=((W3[t1]) ^ (C));
    if(t1==3)
        W3[t1]=((W3[t1]) ^ (D));
    if(t1==4)
        W3[t1]=((W3[t1]) ^ (E));
    if(t1==5)
        W3[t1]=((W3[t1]) ^ (F));
    if(t1==6)
        W3[t1]=((W3[t1]) ^ (G));
    if(t1==7)
        W3[t1]=((W3[t1]) ^ (H));
}
```

b) Concatenation function (Cfs): The 160-piece Initial Vector (IV) is one of the contributions to the hash code creation function ($H_{SSLH}$). It tends to be extended to 256-bits by connecting all underlying vector esteems in roundabout way, which is called Concatenated Vector (VC). The connection cycle is appeared in Fig. 5.
with variable1(km). The input i.e. 8-bit value is recomputed by storing the value of variable3(km2)(four bits starting from least significant position) in four bits starting from most significant position and next four bit of initial eight bit(input) stating from least significant position gets the bits from variable2(km1) (four bits starting from least significant position) so this way the 8-bit of input is modified. The same procedure is adopted for remaining three 8-bit values generated from 32 bit that is

B1 = (W3[t] >> 8) & 0xff

Second eight bit number generated from 32 bit number

A2 = (W3[t] >> 16) & 0xff

Third eight bit number generated from 32 bit number

B2 = (W3[t] >> 24) & 0xff

Fourth eight bit number generated from 32 bit number

A1 \rightarrow y1

return y1;

\text{int mod}_{-} \text{Int2}(\text{int y1})

\{

\text{int i,j,km}=0, \text{km1}=0, \text{km2}=0;

\text{km1}=y1 & 0xf;

\text{km2}=(y1 >> 4) & 0xf;

\text{km1}= \text{km1} \uparrow \text{km};

\text{km2}= \text{km2} \uparrow \text{km};

y1 = (\text{km2}<< 4) + (\text{km1});

\text{return y1};
\}

A similar system is followed for remaining seven 32-digit blocks.

e) New Point (on elliptic curve) Estimation Work (NPfI): Each 32-bit block is separated into two sub blocks of same length. These sub blocks act as two points of a elliptic curve which are used in new point estimation in elliptic curve [8],[9],[15] as shown in Fig. 3. First by using two points slope of line is calculated the this slope value (M) is used in the calculation of new X and Y axis point on elliptic curve. The resultant new point is 16 bit value i.e X axis is 8 bits and Y axis is 8 bits so both taken together is 16 bit sub block. A total of 8 16-bit sub block are generated after new point calculation. Now this new points are modified by performing XOR on new points. Each 8-bit sub block is XOR with its adjacent 8-bit sub block except first 8-bit sub block which is XOR with a 8 bit sub block of all zeros. After above operation adjacent sub blocks are added in such a way that it becomes 32 bit sub block. Repeating this, results in four 32-bit sub block.

Slope of line in elliptic curve (for Real) is calculated as

\text{slope}(\lambda) = \frac{(yy2 - yy1)}{(xx2 - xx1)}

where (xx1,yy1) and (xx2,yy2) are points on elliptic curve. New point on elliptic curve for reals are calculated using the following formula

x3(new point) = (slope(\lambda))2 - xx1 - xx2

and

y3(new point) = slope(\lambda) * (xx1-xx2) – yy1

the above formula is for the case when x1 != (not equal to) x2 (this is assumed that x1 is not equal to x2)

f) Output: The 160-bit hash code is obtained by attaching four 32 bit sub block obtained in above process and the final 32-bit sub block is obtained by using the above four 32-bit sub block as shown below:

\text{E} = (A) \uparrow (B>>5) \uparrow (C) \uparrow (D<<5)

160-bit message digest = A + B + C + D + E

All hexadecimal numbers are replicated into convey vector (or instate MD cushion), which is the message overview of a given message (if the message size is 512-bit block) in any case the above cycle is rehashed until the last 512-digit square of the message.

Table I representation of input message before padding in hexadecimal and Table II representation of input message after padding in hexadecimal.

\begin{table}[h]
\centering
\begin{tabular}{|c|}
\hline
\textbf{TABLE I. REPRESENTATION OF INPUT MESSAGE BEFORE PADDING IN HEXADECIMAL} \\
\hline
\textbf{EX:}: Message “I am from Jabalpur working hard for kits singapur”
\textbf{Characters count 41}.
\textbf{Bit count 328}.

\textbf{Before cushion} (in Hexadecimal representation)
49 20 61 6d 20 66 72 20 77 6f 72 6b 69 6e 67 20 68 61 72 64 20 75 72 20 77 6f 72 6b 69 6e 67 20 68 61 72 64 20 75 72 20 61 6d 20 66 72 6f 6d 20 6a 61 62 61 6c 70 20 61 6d 20 66 72 6f 6d 20 6a 61 62 61 6c 70
\hline
\textbf{TABLE II. REPRESENTATION OF INPUT MESSAGE AFTER PADDING IN HEXADECIMAL} \\
\hline
\textbf{After padding} (in Hexadecimal representation)
49 20 61 6d 20 66 72 20 77 6f 72 6b 69 6e 67 20 68 61 72 64 20 75 72 20 77 6f 72 6b 69 6e 67 20 68 61 72 64 20 75 72 20 61 6d 20 66 72 6f 6d 20 6a 61 62 61 6c 70 20 61 6d 20 66 72 6f 6d 20 6a 61 62 61 6c 70
\hline
\end{tabular}
\end{table}

\text{g) Alteration function (AfI): Tempx8 = 0 implies a brief exhibit instated with zeroes. XOR activity is performed on every 8-digit square of above advance with Tempx8 by one. First letter(I) decimal value 73 and hexadecamal equivalent is 49. Now 73 is XOR with Temp8 as shown below 73 binary form is 01001001 and temp8 is 00000000 so modified first8-bit block is 01001001 \uparrow 00000000 = 01001001}.

Now Temp8 is incremented by 1 so it becomes 1 and next input decimal value is 32 and hexadecamal equivalent is 20. So the second 8-bit block is modified by XOR it with Temp8 i.e. 00100000 \uparrow 00000000 = 00100001(decimal equivalent 33) this is the value of second 8-bit sub block and Temp8 is incremented and XOR with third 8-bit sub block value and the process continues till last 8-bit sub block of 512-bit block.

Once the 512-bit block of message is altered it again subjected to modification i.e. the first 8-bit sub block value is complimented and XOR with complimented (mid+1)-bit sub block value, the result is again complimented and stored in a separate array(W2). Then the second 8-bit sub block value is
complemented and XOR with complemented (middle+2)8-bit sub block value, the result is complemented and stored in the second location of array(W2) and the process continues for rest of 8-bit sub block value till first half of input message(after first step). The result is 512 bit block is reduced to 256 bit block. The modified message as depicted:

The Alteration Step result

| W2[0] = ffffffff0 | W2[1] = ffffffff90 |
| W2[2] = ffffffffcc | W2[3] = ffffffff92 |
| W2[4] = ffffffff94 | W2[5] = ffffffff00 |
| W2[6] = ffffffff9d | W2[7] = ffffffff3c |
| W2[8] = ffffffff92 | W2[9] = fffffff8c |
| W2[10] = ffffffffdc | W2[11] = ffffffff0d |
| W2[12] = ffffffffda | W2[13] = ffffffffdf |
| W2[14] = ffffffffec3 | W2[15] = ffffffffda |
| W2[16] = ffffffff88 | W2[17] = ffffffff2d |
| W2[18] = ffffffff | W2[19] = ffffffff88 |
| W2[20] = ffffffff0b | W2[21] = fffffffadd |
| W2[22] = ffffffff4b | W2[23] = ffffffff6 |
| W2[24] = ffffffffbb1 | W2[25] = fffffff88 |
| W2[26] = ffffffff | W2[27] = fffffffb7 |
| W2[28] = ffffffffbe | W2[29] = fffffffadd |
| W2[30] = fffffffba | W2[31] = fffffff77 |

h) Concatenation function(Cfn): Connecting beginning vector register an incentive in the accompanying design: (link circularly S0 S1 S2 S3 S4 S0 S1 S2)

67 45 23 01
ef cd ab 89
98 ba dc fe
10 32 54 76
c3 d2 e1 f0
67 45 23 01
ef cd ab 89
98 ba dc fe

i) 32- bit XOR operation (XORop): XOR activity is performed on each 32-pieces of altered message with relating 32-pieces of extended beginning vector register esteem.

(Original message) (Expanded vector) (Result)

ef86fcb92 | c3 d2 e1 f0 (S4) | 2c5d2a62 |
93cfd8c3 | 10 32 54 76 (S3) | 83fd8cb5 |
918bdbd0 | 98 ba dc fe (S2) | 931072e |
d9dec2da | ef cd ab 89 (S1) | 36136953 |
d72cfe6a8 | 67 45 23 01 (S0) | b069da9 |
afacb3b6 | c3 d2 e1 f0 (S4) | 6c7e5246 |
b0b7ebe7 | 10 32 54 76 (S3) | a085aac1 |
bdacb977 | 98 ba dc fe (S2) | 25166589 |

j) Segregation and Modification operation(Sop): Separating the above result (256-bits) into 8 sub-block of 32-bits each and then each 32 bit block is separated into four 8-bit block, after that it modifies each eight bit block as shown in Fig. 6. This process is repeated for all 32 bit block.

The Result is

A1[0] is 42
A2[0] is 8d
A1[1] is e5
A2[1] is 2d
A1[2] is ce
A2[2] is 21
A1[3] is 63
A2[3] is 23
A1[4] is 39
A2[4] is f9
A1[5] is 26
A2[5] is 9e
A1[6] is d1
A2[6] is d5
A1[7] is 19
A2[7] is 76

k) New Point (on elliptic curve) calculation function:

Before calculating new point on elliptic curve the 32-bit block is divided into four equal sub-blocks i.e 8-bit block as shown in Fig. 7.

(xx1, yy1) and (xx2, yy2) bits are converted into integers.

for ( t = 0; t < 8; t++)
{
A1 = W3[t] & 0xff; // A1=xx1
B1 = (W3[t] >> 8) & 0xff; //B1=yy1
A2 = (W3[t] >> 16) & 0xff; //A2=xx2
B2 = (W3[t] >> 24) & 0xff; //B2=yy2
M=(B2-B1)/(A2-A1); // Slope
A3[i]=(M*M)-A1-A2; //A3 = New point and method of calculation.
A3[i]=A3[i]^T1; // New point altered by XOR it with it adjacent (previous) new point except first new point which is XOR with a array of size eight with all zeros.
B3[i]=M * (A1-A2)-B1; // B3= New point and method of calculation.
B3[i]=B3[i] ^ A3[i]; //New point altered by XOR it with it
adjacent (previous) new
point except first new point which is XOR with a array of size
eight with all zeros.

T1=B3[i];
i++;}
The Result is
Calculated New Point value A3[0] is ffffff32
Modified New Point value A3[0] is ffffffff32
Calculated New Point value B3[0] is 19
Modified New Point value B3[0] is 19
Calculated New Point value A3[1] is ffffffff67
Modified New Point value A3[1] is ffffffff67
Calculated New Point value B3[1] is ffffffff67
Modified New Point value B3[1] is 19
Calculated New Point value A3[2] is ffffffff89
Modified New Point value B3[2] is 1d
Calculated New Point value A3[2] is ffffffff89
Modified New Point value B3[2] is 1d
Calculated New Point value A3[3] is ffffffff7e
Modified New Point value A3[3] is ffffffff7e
Calculated New Point value B3[3] is ffffffff7e
Modified New Point value B3[3] is 19
Calculated New Point value A3[4] is ffffffff3c
Modified New Point value A3[4] is ffffffff3c
Calculated New Point value B3[4] is ffffffff3c
Modified New Point value B3[4] is 1d
Calculated New Point value A3[5] is ffffffff23
Modified New Point value A3[5] is ffffffff23
Calculated New Point value B3[5] is ffffffff23
Modified New Point value B3[5] is 1d
Calculated New Point value A3[6] is 3b3
Modified New Point value A3[6] is 3b3
Calculated New Point value B3[6] is ffffffff23
Modified New Point value B3[6] is ffffffff23
Calculated New Point value A3[7] is ffffffff71
Modified New Point value A3[7] is ffffffff71
Calculated New Point value B3[7] is ffffffff71
Modified New Point value B3[7] is 1d

for(i=0;i<16;i=i+2)
{
if(i<2)
AA = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1]);
if(i>=2 && i<4)
BB = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1]);
if(i>=4 && i<6)
CC = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1]);
if(i>=6 && i<8)
DD = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1]);
}
EE=~(AA) ^ (BB>>5) ^ ~(CC) ^ ~(DD <<5);

output: The 160-bit hash code is made by joining
four 32-bit sub block obtained in above process and the final
32-bit sub block is obtained by using the above four 32-bit sub
block generated in above process as shown below:

Final hash code for 512 bit block of the message
is generated by adding (S0, S1, S2, S3, S4) to individual 32
bit of above result as shown below:

\[
\begin{align*}
AA &= S0 + AA; \\
BB &= S1 + BB; \\
CC &= S2 + CC; \\
DD &= S3 + DD; \\
EE &= S4 + EE; \\
\text{final hash code: d3ef4e1 0a4cb54f 84d900ab 1a7f09c5 97a1d6d1}
\end{align*}
\]
All hexadecimal numbers are replicated into instant MD buffer, which is the message condensation of a given message in the event that it is a 512-digit block in any case the above cycle is rehashed until the last 512-bit block.

For ex: If Elliptic curve new point is (8, 12), then the binary form is

and its modified form is shown below

8 in its binary form is 00001000
12 in its binary form is 00001100.
8 cycle XOR operation is as per the following:

Step 1: Let AA1 be a array of size 8 and duplicate these qualities into AA1.

\[
AA1[] = \{0, 0, 0, 0; 1, 0, 0, 0\};
\]

Step 2: Let Tempx8 be an array of size 8 and introduce with zeroes.

\[
Tempx8[] = \{0, 0, 0, 0, 0, 0, 0, 0\};
\]

Fig. 8. Operation of Step3.

Step 3: Perform XOR operation on every 8-bits of AA1 and Tempx8:

\[
\{0 0 0 0 0 0 0\} ^\lor \{0 0 0 1 0 0 0\} = \{0 0 0 0 1 0 0\}
\]

Now 1st 8-bit sub block is modified to \{0 0 0 1 0 0 0\} and Tempx8[] \{0 0 0 0 1 0 0\} and AA1[] takes the other value of new point so AA1[] \{0 0 0 0 1 1 0\}. Now XOR operation is done between AA1[] and Temp which is shown below.

\[
\{0 0 0 1 0 0 0\} ^\lor \{0 0 0 0 1 1 0\} = \{0 0 0 0 1 0 0\}
\]

Again the 2nd 8-bit sub block is modified i.e. it becomes \{0 0 0 0 0 0\} and tempx8[] changes its value i.e. Tempx8[] \{0, 0, 0, 0, 1, 0, 0\}; as shown in Fig. 8 and the process continues for all the other seven new point.

Step 4: Above operation results in 128 bits i.e. 8 sub block of 16 bits. Here two adjacent sub block are taken and added in such a way that the result is 32 bit sub block. the process is shown.

```
for(i=0; i<16; i=i+2)
{
    if(i<2)
        AA = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1];
    if(i>=2 & & i<4)
        BB = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1];
    if(i>=4 & & i<6)
        CC = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1];
    if(i>=6 & & i<8)
        DD = (A3[i] << 24) + (B3[i] << 16) + (A3[i+1] << 8) + (B3[i+1];
}
```

The above process results in four 32-bit sub block. By using this four 32-bit sub block fifth 32 bit sub block is computed which is shown below:

\[
EE=\neg(AA) ^\lor (BB>>5) ^\lor (CC) ^\lor (DD <<5)
\]

The result for the message “I am from Jabalpur working hard for kits singapur” is 32171843 (AA) 53d9a622 (BB) ec1e23ad (CC) 1a7f09c6 (DD) 6c89d1e0 (EE)

After computing AA , BB , CC , DD , EE , this values are added with individual 32 bits of initialize MD buffer (S0 , S1 , S2 , S3 , S4 ) to generate the new expended value for next 512 bit of the given message. The process is shown below

\[
AA = S0 = S0 + AA;
BB = S1 = S1 + BB;
CC = S2 = S2 + CC;
DD = S3 = S3 + DD;
EE = S4 = S4 + EE;
\]

AA , BB , CC , DD , EE values generated for last 512 bit of the given message will be the hash code for authentication.

The hash code for the message “I am from Jabalpur working hard for kits singapur” is d3cefe4e1 0a4cb54f 84d900ab 1a7f09c5 97a1d6d1

IV. RESULTS AND DISCUSSION

SSLHA-160, RIPEMD 160, MD5 and SHA-1 when implemented in and run on windows 7 32bit, processor 2.00 Giga Hz with 2 GB of internal memory.

Input String: “The student tried hard but failed in the exam”.

Output (Hash Code): 293967e0 a2141e90 93f12216 b2467106 ed0c49d0.

When the given message is slightly changed, this results in huge alteration in the yield due to the avalanche effect, which is a property of Hash function. For example, when the word exam is changed to rxam i.e. a single letter is changed this produces a hash code which differs from the original hash code by 92 bit out of 160.
Input String: “The student tried hard but failed in the rxam”.

Output (Hash Code): fab3ee1b facc7548 dc39da9e 0afe9ebe be86d00b
293967e0 : 0010 1001 0011 1001 0110 0111 1110 0000 fab3ee1b : 1111 1010 1011 0111 0110 0001 1011
---------------------------------------------------------
3 2 1 1 2 4 3
---------------------------------------------------------
a2141c90 : 1010 0010 0001 0100 0001 1100 1001 0000 facc7548 : 1111 1010 1100 1100 0111 0101 0100 1000
---------------------------------------------------------
2 1 3 1 2 3 1
---------------------------------------------------------
93f12216 : 1001 0011 1111 0001 0010 0001 0011 dc39da9e: 1101 1100 0011 1001 1101 1010 1001 1110
---------------------------------------------------------
1 4 2 1 4 1 1 1
---------------------------------------------------------
b2467106: 1011 0010 0100 0111 0111 0000 0000 0110
0afe9ebe: 0000 1010 1111 1110 1001 1101 1110
---------------------------------------------------------
3 2 3 1 3 1 3 1
---------------------------------------------------------
ed0c49d0: 1110 1101 0000 1100 0101 1001 1001 0000 be86d00b: 1011 1110 1000 0110 1100 0000 0000 1001
---------------------------------------------------------
2 2 1 2 2 2 3 3
---------------------------------------------------------

Total number of bit changed is: 18+14+15+17+17 =81 bit out of 160 bits. When the same experiment was done on SHA-1 it also showed avalanche effect, but only 76 bits changed, in case of MD5 74-bits (57-bits for 128 bits message digest) and in case of RIPEMD-160 82 bits changed as shown in Fig. 9.

Table III shows the message digest produced by four algorithm (SSLHA160, SHA-1, RIPEMD160 and MD5) for the message “The student tried hard but failed in the exam”. The aftereffect of Table III is utilized for computing avalanche impact appeared in above Fig. 8. Table IV shows the adjustment in message digest when the message in Table III is modified for example exam is supplanted with dxam, SSLHA-160 produces a message digest with change of all hexadecimal qualities aside from 9,6 and e(starting from left shown in intense), SHA-1 yield is a hash code with change of all hexadecimal qualities aside from 5,7,e,f and 5(starting from left shown in intense) and RIPEMD 160 produces a message digest with change of all hexadecimal qualities aside from 8 and 5(starting from left shown in strong) and Table V shows the adjustment in hash code when the message in Table III is modified for example exam is supplanted with fxam, SSLHA-160 produces a message digest with change of all hexadecimal qualities aside from 2 and e(starting from left appeared in striking), SHA-1 yield is a hash code with change of all hexadecimal qualities aside from 3(starting from left shown in intense) and RIPEMD 160 produces a message digest with change of all hexadecimal qualities aside from b,0,0 and 5(starting from left shown in strong). So on an average SSLHA-160 shows similar result when any character is changed. The result is better than SHA160 and RIPEMD160 as shown in Fig. 10.

**TABLE III. RESULT**

| Algo/Input | “The student tried hard but failed in the exam” |
|------------|-----------------------------------------------|
| SSLHA160   | 293967e0 : 0010 1001 0011 1001 0110 0111 1110 0000 |
|            | b2467106: 1011 0010 0100 0111 0111 0000 0000 0110 |
|            | 0afe9ebe: 0000 1010 1111 1110 1001 1101 1110 |
| SHA-1      | d6574b20 5edf6cfe 90161af3 b5f335a 2d20ed77 |
| RIPEMD160  | 90c6b9e3 8f01582 3b3bb008 c0f452bf 42f27bc |
| MD5        | 6fa7ff78 11a4814a d4a0699e 7d9f8253 |

**TABLE IV. RESULT (IN INPUT EXAM CHANGED TO DXAM)**

| Algo/Input | “The student tried hard but failed in the dxam” |
|------------|-----------------------------------------------|
| SSLHA160   | d8596faa 5660515c 8a56eca 6692a5d2 9e2c50ea |
| SHA        | 2f9798c0 6e6025a8 a648850e caf12750 474b9bce |
| RIPEMD160  | 51a8d28a 86b7344f b54efa6f a1de5609 8f2216fd |

**TABLE V. RESULT (IN INPUT EXAM CHANGED TO FXAM)**

| Algo/Input | “The student tried hard but failed in the fxam” |
|------------|-----------------------------------------------|
| SSLHA160   | 2a606123 3f316c3d e144d5a9 03f32b3 e478c313 |
| SHA        | 8086f9af 35fcd5da db03cd3d 5496a8h1 e954a0b |
| RIPEMD160  | b6363ac9 49652e09 9c0bd0d7 60a8bd6 c3d5759b |
Above way is adopted for calculating new point of remaining sub blocks and then modify it, after modifying the adjacent new point are added such that they becomes 32-bit sub block. Repeating the process for rest of modified new point’s results in four 32-bit numbers. By using this four 32-bit sub block i.e. performing some mathematical operation and XOR them results in fifth 32-bit sub block. The outcome is linked and changed over to hexadecimal structure. In this way, it shows the single direction(one way) property of SSLHA-160 hash calculation.

Assaults not related with the calculation are:

Irregular assault- The likelihood of breaking this calculation is 1/2160, the quantity of trails and the expected values are the vital boundaries of this assault.

Birthday assault- The idea driving birthday assault came from a well-known issue from probability theory, which is called birthday paradox. By utilizing this idea assault on hash capacity can be outlined. On the off chance that the length of the Message digest is 160 pieces, at that point there are 2160 prospects. The Cryptanalyst produce two examples which are P1 and P2 from digest, the estimated likelihood of two examples is as per the following:

\[
\rho \approx 1 - \frac{1}{e^{2P/160}}
\]

Assaults related on the calculation are:

Meet-in-the-middle assault- is a variant of the birthday assault, here aggressor endeavors to locate any two q1 and q2 with the end goal that their hash esteems are equivalent to y = h (q1) = h (q2). This assault is identified with discovering two people with a similar birthday. Let x be the probability that two person birthday are equal and x! be the probability that the birthdays are not equal. Then the probability is defined as

\[
1 - x! = x
\]

Now suppose that there is only one person, then the probability that his birthday is not same with any one is 1. If there are two person then the probability that they have birthdays on different dates are 1 * 364/365(considering year to be of 365 days). Similarly if three persons are there then the probability that they have birthday on different dates is 1 * 364/365 * 363/365. If we calculate for 9 persons i.e. the probability that nine person have birthday on different dates are:

\[
1 * 364/365 * 363/365 * 362/365 ------9Person
\]

When multiplied we get around 0.9 i.e. 90% which means that the probability that out of nine person two persons having birthday on same date is 10%. Going in the same way when we calculate probability for 23 person having birthday on different dates we get around 49.9% and the probability that two persons out of 23 having birthday on same date will be greater than 50%. One expects lower probability as there are 23*22/2 = 253 pairs of persons.

The birthday paradox can be utilized for assaulting hash capacities. An enemy produces q1 varieties of a sham message and q2 varieties of a veritable message; n is the quantity of
pieces of a hash esteem. At that point, the likelihood of finding a sham message and a certified message as follows:

\[ P \approx 1 - \left(1 + \left( e^{2\pi n}\right)^{2}\right) \]

Remedying block assault - The analyzer takes a message and its hash code on which he attempts to change the block a few times and notices the summary(digest) remaining parts same or not.

Differential assault - The rule of an assault is the investigation of social contrasts among information and yield. An impact happens if the difference is zero.

**B. Performance Analysis**

These three calculations SSLHA-160, SHA1 and MD5 were tried for comparison dependent on the execution time necessities as shown in Fig. 11, Fig. 12 and Fig. 13. All the calculations have been actualized in C/C++ and run on windows 7 32-bit, CPU 2.00 GHz with 2 GB of internal memory. With the aftereffects of the analysis, it was discovered that SHA1 and MD5 requests more execution time than SSLHA160 to create hash code.

The result in above Fig. 14 demonstrates time taken by three algorithm (SSLHA160, SHA1, and MD5) for generating message digest for the message “I am from Jabalpur working hard for kits singapur”.

![Fig. 11. Time Taken by SSLHA160 for Executing “I am from Jabalpur Working Hard for Kits Singapur”](image1)

![Fig. 12. Time Taken by MD5 for Executing ”I am from Jabalpur Working Hard for Kits Singapur”](image2)

![Fig. 13. Time Taken by SHA-1 for Executing “I am from Jabalpur Working Hard for Kits Singapur”](image3)

![Fig. 14. Examination of SSLHA160, SHA1 and MD5 Concerning Time Taken to Execute the Message “I am from Jabalpur Working Hard for Kits Singapaur” i.e. 41 Bytes.](image4)

**VI. CONCLUSION**

The necessity for new hash plans is expanding to make security viewpoints solid, for example, validation, message honesty and secrecy as for present status of web as web conditions are much of the time evolving. The specialists in cryptography should invest solid energy to concoct better plan measures utilizing which long haul and powerful security can be given by hash capacities. The proposed hash work, A strong and secure lightweight cryptographic hash algorithm (SSLHA-160) is created utilizing elliptic curve ideas which is contrasted with different cryptographic hash capacities. The center qualities of SSLHA-160 are XOR activities, bitwise compliment , bitwise left and right shift and new point estimation(for elliptic curve), which bring about solid nonlinear avalanche impact, expanded dispersion in yield and make differential assaults troublesome. Consequently it is safer and simple.

**REFERENCES**

[1] Venkateswara Rao Pallipamu, K Thammi Reddy, Suresh Varma “ASH-160: A novel algorithm for secure hashing using geometric concepts” 2014 Elsevier Ltd.

[2] Venkateswara Rao Pallipamu, K Thammi Reddy, P Suresh Varma “ASH-512: Design and implementation of cryptographic hash algorithm using co-ordinate geometry concepts” 2014 Elsevier Ltd.

[3] Venkateswara Rao Pallipamu, P Suresh Varma “Design and implementation of geometric based cryptographic hash algorithm: ASH-256” 2016 IESRT

[4] Dobbertin H, Bosselaers A, Preneel B. RIPMEMD-160: a strengthened version of RIPMMD. In: Gollmann D, editor. Fast software encryptions, LNCS 1039. Springer-Verlag; 1996. pp. 71 - 82.

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

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

[7] K Bhaskar Prakash, Pasala Sanyasi Naidu2 “Strong and powerful cryptographic scheme for Industrial Internet of Thing using pseudo stream cipher, elliptic curve cryptography and trigonometric techniques (An efficient scheme to detect COVID-19 patient from Quarantine people) Solid State Technology Volume: 63 Issue: 6 Year: 2020

[8] Kahate Atul. Cryptography and network security. Tata McGraw-Hill; 2006.

[9] Swayam Course in Information Security(Swayam is a program initiated by Government of India and designed to achieve the three cardinal principles of Education policy viz, access, equity and quality)
[10] NIST. Secure Hash Standars. FIPS PUB; 2002. pp. 180 - 2.
[11] Preenel B. Cryptographic hash functions. Transactions on Telecommunications 1994;5:431 - 48.
[12] Rivest RL. The MD4 Message Digest Algorithm. RFC 1320; 1992.
[13] Rivest RL. The MD5 Message Digest Algorithm. RFC 1321; 1992.
[14] Rivest RL. The MD2 Message Digest Algorithm. RFC 1319; 1992.
[15] Stallings William. Cryptography and network security: Principles and practice. 3/e PH; 2003.
[16] Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition; September 2009. http://csrc.nist.gov/publications/nistir/7620/nistir_7620.pdf.
[17] Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition.http://csrc.nist.gov/publications/nistir/7764/nistir-7764.pdf; February 2011.
[18] Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition.http://dx.doi.org/10.6028/NIST.IR.7896; November 2012.
[19] Wang X, Yu H. How to break MD5 and Other hash functions. In: Cramer R, editor. EUROCRIPT 2005, LNCS 3494; 2005. pp. 19 - 35.
[20] Wang X, Feng XD, Lai X, Yu H. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. rump session, CRYPTO 04; 2004.
[21] Wang X, Yin Y, Yu H. Finding Collisions in the Full SHA-1. Lecture notes in Computer Science. CRYPTO 2005 Proceedings, Vol. 3621; 2005. pp. 17 - 36.
[22] Wang X, Feng D, Lai X, Yu H. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint archive; 2004.